Uncompromising Performance with Exocompilation

by

Yuka Ikarashi

M.S., Massachusetts Institute of Technology, 2022.
B.S., The University of Tokyo, 2020.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2026

Authored by: Yuka Ikarashi
Department of Electrical Engineering and Computer Science
August 7, 2026

Certified by: Jonathan Ragan-Kelley
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Uncompromising Performance with Exocompilation

by

Yuka Ikarashi

Submitted to the Department of Electrical Engineering and Computer Science
on August 7, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Performance is the currency of modern computing. In AI and scientific HPC, wall-clock time, throughput, and energy per operation translate directly into model scale, scientific resolution, cost, and environmental impact. Today's best results are powered by sophisticated HPC libraries that unlock hardware accelerators without forcing every engineer to write CUDA or assembly. Yet building and maintaining those libraries remains labor-intensive and brittle, particularly as hardware evolves rapidly. This motivates a clear goal: programming languages and compilers that make peak performance readily achievable, delivering lower latency and higher throughput while reducing engineering cost. The sheer number of HPC languages being developed today reflects how actively the field is pursuing this goal.

Achieving this goal, however, has proven difficult, as HPC language design faces a fundamental tradeoff between automation and control: for every optimization, designers must decide whether to automate it for productivity or expose it for maximum performance. Mainstream compiler research has historically leaned toward automation. In practice, however, compiler automation works until it doesn't. When new optimizations turn out to be inexpressible with the provided controls, or when automation is imperfect, engineers are left with stark choices: accept degraded performance, abandon the language for C or assembly, or even modify the compiler itself.

This thesis addresses that impasse through **Exocompilation**, a design philosophy holding that future-proof compilers must give engineers maximum control, with built-in automation minimized to safety analysis alone. We realize this philosophy by externalizing two traditionally hardcoded compiler responsibilities to user code: (1) targeting new hardware and (2) implementing optimization strategies. By shifting them to user code, compilers remain extensible as hardware evolves, while developers gain the flexibility to tailor optimizations to their specific architectural needs. This externalization reconciles the long-standing trade-off between control and automation by keeping the core compiler minimal, auditable, and predictable while letting richer automation emerge through reusable user-defined libraries. **Exo** is a language that embodies this philosophy: programmers optimize via user scheduling—rewriting programs into functionally equivalent but faster versions—with both the object language and scheduling operators kept deliberately low-level, making performance-critical constructs explicit and enabling the exact optimizations needed for peak performance.

Thesis supervisor: Jonathan Ragan-Kelley
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I want to thank my advisor, Jonathan Ragan-Kelley, and Gilbert Bernstein, who has been a second advisor to me in every way. I'm happy with how my PhD turned out, and that is entirely thanks to the two of them. I came in new to PL, and they taught me everything that has shaped me into the researcher I am today. Early on, they gave me the hands-on guidance I needed, and later they gave me the resources and freedom to pursue whatever I wanted. I couldn't have wished for a better advisor and mentor.

I'm also grateful to Saman Amarasinghe and Adam Chlipala, who served on my thesis committee. Saman was the first MIT faculty member I met, back in 2019 when Jonathan was still at Berkeley. I was very nervous at the meeting, but he encouraged me to apply, which meant a great deal, and he has remained generous with his support through every moment of doubt since. Adam, for his part, is always on top of everything, with sharp opinions and advice. I came to know him later in my PhD, yet I still learn so much every time we talk.

The Exo coauthors and collaborators shaped most of my PhD experience, and much of the work in this thesis was done in close collaboration with them. They made not only the work better but also the long hours more fun, and I have fond memories of all the intense discussions and the paper deadlines we pushed through together. I want to thank David Zhao Akeley, Samir Droubi, Hasan Genc, Kevin Qian, Alex Reinking, and Chengpeng Wang. David led the Exo-GPU work in Chapter 7. Samir and Kevin contributed substantially to the implementation and evaluation of the scheduling libraries in Chapter 5. Alex contributed to the design of Cursors (Chapter 5) as well as to the AVX-512 evaluation in Chapter 4. I am also grateful for the undergraduate and master's students I had the pleasure of working with on Exo: Kenneth Moon, Mario Mrowka, Luca Musk, Jason Ng, Jacob Teo, and Andrew Weinfeld. I hope I was a good mentor to you all.

Through Exo, I was fortunate to have many users and collaborators in both industry and academia who supported the project and gave me valuable feedback along the way. This list is not exhaustive, but I want to thank: Rahul Bakshi, Julian Bellavita, Adrián Castelló, Julien de Castelnau, Abhijit Davare, Grace Dinh, Skye Elliot, Tobias Grosser, Asaf Hargil, Rin Iwai, Pritpal Kanhaiya, Sasha Lopoukhine, Aaftab Munshi, Rachit Nigam, Yukinori Sato, Ali Sazegari, and Sophia Shao. Early in my PhD, I also worked on a knitting-semantics project with my wonderful CMU collaborators. Though it isn't part of this thesis, I learned an enormous amount from Jenny Lin, Jim McCann, and Vidya Narayanan, whose lab became a second home to me.

I owe a great deal to my undergraduate mentors: Tsukasa Fukusato and Takeo Igarashi at the University of Tokyo, and Vassil Vassilev at CERN, who started me in research and have continued to support me ever since. Becoming an MIT PhD student as a random Japanese

undergrad was quite nontrivial, and I wouldn't be here at all without them.

Thanks to my labmates, old and new, who made the office such a fun place to work: Rahul Goel, Manya Bansal, Kartik Chandra, Tian Jin, Amanda Liu, Ahmed Mahmoud, William Brandon, Tzu-Mao Li, Karima Ma, Kevin Mu, Anne Ouyang, AJ Root, and Katherine Mohr. I especially want to thank Amanda, with whom I colocated my defense and who hosted so many parties at her place, where I met many great people and enjoyed her delicious homemade bread and cake.

I am deeply grateful for the generous support of the Funai Overseas Scholarship, the Masason Foundation Fellowship, and the Quad Fellowship, which not only funded me for five years but also gave me a wonderful community. A few times each year, these fellowships brought me together with other recipients, some pursuing PhDs and others chasing different ambitions, and the bonds we formed supported me well beyond the professional sphere. I am especially thankful to the friends I met through Funai events, several of whom I was also lucky enough to serve alongside on the Japanese Graduate Student Association in the United States (JGSAU): Riku Arakawa, Akari Asai, Hajime Fujita, Emi Ito, Yuto Katsuyama, Akio Kodaira, Tatsuki Koga, Natsumi Komatsu, and Motoya Ohnishi. My heartfelt thanks go to Daichi Hayakawa and Shion Kubota from Masason for being my roommates during the first few months of my PhD. You two made starting graduate school in the middle of COVID so much better than it could have been.

A PhD is a long journey, and I am thankful to my friends in Cambridge and at MIT, many of whom were also pursuing their own PhDs, for late-night discussions, board games, and cooking. My warm thanks go to Toshiki Aoki, Sarah Gurev, Yuzuru Kanda, Kota Kondo, Mao Miyamoto, Minori Narita, Ryotaro Okabe, Yugo Onishi, Yoshihiro Saito, Tomoya Sasaki, Tomohiro Soejima, Yosuke Tanigawa, Ayako Tsuchiyama, Kakei Yamamoto, and Anna Yui. Some of them I was also fortunate to work with through the Japanese Association of MIT (JAM), which was a great deal of fun. And a special thank you to Kosuke Yoshinaga, who was always there to organize the BBQs that brought us together.

I am also grateful for the many people I met during my time away from MIT. My time at UW, Berkeley, Apple, and Amazon gave me far more than research experience. These research visits and internships were a wonderful chance to build friendships with people from universities across the country. I want to thank Seah Kim, Hongzheng Chen, Yishu Zhou, Aniket Deshmukh, Albert Ou, Haoran Peng, and Ryan Zambrotta for their camaraderie, conversations, and company. A special thanks goes to Philippe Tillet, my most frequent texting friend, who has inspired me both personally and professionally and who probably has a more detailed log of my daily activities than my calendar does.

I want to thank my parents for setting such a great example for me. After spending six years at MIT, I can still confidently say that they are the most hardworking people I have ever met, and both have built successful careers of their own. Watching my mom balance her career and her life made me confident that I can, too, have a successful career and a happy life as a woman. From them I also inherited mental and physical resilience, which may well be the most important trait for completing a PhD. I am very proud to be their daughter.

I am grateful to my grandparents, brother, cousins, and other relatives for the attention, support, and love they have given me. My grandfather was a chemist who spent two years as a postdoc at Lawrence Berkeley National Lab. He told me that there was little East Asian presence in science during his time there, and that he was once mistaken for a beggar at a

bus stop, which left him demoralized. I am grateful to the generations of Japanese people like him who worked so hard to build a country with a reputation for being polite and cool. I have benefited a great deal simply from being Japanese. My grandmother, on the other hand, was a pharmacist who left her career when she married. She was well-educated and could read and write in English, and in an era when papers were written on typewriters, she typed out my grandfather's manuscripts without ever being credited. I am certain there are countless stories like hers, of uncredited women in science and academia. As a woman in academia myself, I am deeply grateful to the women of past generations who fought for the equality that allows me to be a scholar today.

To my husband Masashi—I don't know how to thank you for the last eleven years. You believed in me way before I believed in myself, and that got me through the times I wanted to give up. You gave up so much to be here: your friends, your family, your work in Japan, all to come to the US and stand by me back when I had no idea where any of this would lead. I truly couldn't have finished this without you, and I'm so grateful for everything. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance is the currency of modern computing. Over the past decade, computational demands have grown at a staggering pace, driven above all by artificial intelligence [1,2]. Each new generation of neural networks demands orders of magnitude more floating-point operations (FLOPs) than the last, and inference across billions of devices and queries continues to compound this appetite. In AI and scientific computing alike, wall-clock time, throughput, and energy per operation translate directly into model scale, scientific resolution, cost, and environmental impact; the efficient use of every FLOP is therefore not merely a matter of operational thrift but what sets the ceiling on model quality and scientific reach.

It is well-known that meeting these demands depends on specialized hardware accelerators. GPUs and a growing ecosystem of custom ASICs such as Google's TPUs and AWS's Trainium deliver vastly higher throughput than general-purpose processors [3,4]. The shift follows from the end of Dennard scaling, which halted the growth of single-core performance and pushed designers toward parallel, multicore, and increasingly specialized hardware [5]. These accelerators are hard to program for peak performance (the maximum FLOPs per second they can theoretically achieve), because they leave far more to software than CPUs do. Memory movement and instruction scheduling, handled implicitly on out-of-order CPUs, must instead be orchestrated explicitly in software.

Far less widely known is the software layer that connects models to accelerators. Better algorithms and faster processors have never been enough on their own to realize an accelerator's peak performance. Naive scalar code reaches only a tiny fraction of that peak, far short of what modern AI and scientific workloads demand, so performance engineers must tune each kernel to the specific hardware it runs on. That optimization usually involves hand-writing low-level C and assembly, an engineering cost that is easy to underestimate. OpenBLAS, one of the most widely used open-source linear-algebra libraries, makes that cost concrete: its kernel directory alone holds over 1,200 hand-written assembly files spanning roughly 1.1 million lines of code, with separately tuned implementations for more than a dozen instruction-set families, among them x86-64, ARM64, POWER, RISC-V, LoongArch, and MIPS [6]. This cost persists even in libraries built expressly to relieve it. CUTLASS, NVIDIA's open-source library aimed at raising the level of abstraction in GPU programming, is comparable in scale at roughly a million lines of C++/CUDA code [7]. Moreover, NVIDIA's fastest libraries, cuDNN and cuBLAS, are proprietary and reportedly rely on hand-written PTX assembly, much as OpenBLAS does. The complexity of HPC libraries is pervasive across the industry and only increasing.

A single operation shows why. General matrix multiplication (GEMM) is the computational heart of dense linear algebra and much of modern machine learning, but high-performance GEMM is not a single portable loop nest and is implemented not once but across more than 180 distinct assembly kernels in OpenBLAS, hand-optimized for each microarchitecture. The double-precision case alone exists in more than 20 microarchitecture-specific versions, and the Intel Haswell GEMM kernel runs to over 5,000 lines of assembly, with separate kernels for Sandy Bridge and Skylake-X, while later generations like Cooper Lake and Sapphire Rapids add dedicated bfloat16/AMX kernels of their own. The aforementioned modern workloads are powered by libraries like these, which unlock accelerators without forcing every AI engineer to write CUDA or assembly.

Yet with low-level languages like C or CUDA, building and maintaining these kernels is labor-intensive and brittle: each new chip generation can obsolete carefully tuned code almost overnight, and the expertise to rewrite it is scarce and costly. The scale of industry investment reflects the stakes, with programming languages and compilers including OpenAI's Triton [8], Google's Pallas [9], NVIDIA's CuTe DSL and cuTile [10,11], Modular's Mojo [12], and Meta's Helion [13], each aiming to recover peak performance from concise, higher-level abstractions rather than hand-tuned assembly.

The design of these HPC languages has always faced a fundamental tradeoff between compiler automation and user control. For every optimization, language designers must decide whether to automate it for productivity or leave it under programmer control for maximum performance. Mainstream compiler research and development has historically leaned toward automation [14–24]. Even languages specifically tailored for performance engineers, such as Halide [25] and Triton [8], emphasize automating instruction selection and sub-tile GPU mappings. In practice, however, *compiler automation works until it doesn't.* When optimizations are inexpressible with the provided controls (e.g., warp specialization in Triton) or when automation is imperfect (e.g., suboptimal instruction selection in Halide), engineers face stark choices: accept degraded performance, abandon the language for C or assembly, or even modify the compiler itself.

The Triton team's struggle with Blackwell GPUs illustrates this exact problem: when Triton's performance fell far behind NVIDIA's cuBLAS, the team had to invent Gluon [26], a lower-level language that gives users direct control over tile layouts, memory allocation, data movement, and asynchrony, essentially abandoning the higher-level automation to regain the fine-grained control necessary for peak performance. Halide, in turn, offered the other way out of the same bind: modifying the compiler. When its automated instruction selection yielded suboptimal performance, the only possible fix was to change the Halide compiler itself, which is exactly what Rake and Pitchfork did more than a decade after Halide was initially designed [27,28]. Thus, even in mature and widely used kernel languages like Halide and Triton, the boundary between automation and control is neither clean nor stable. When automation proves too slow or programmers need finer control, the only way forward is to modify the compiler or invent a new language.

Exocompilation Philosophy

Compilers cannot, and should not, attempt to automate everything. Rather than relying on abstraction or automation that eventually breaks down, the Exocompilation philosophy holds that future-proof language design must give performance engineers maximum control to achieve peak performance. Built-in compiler automation should be minimal, limited to safety analysis, while key decisions (such as instruction selection) should be left to the engineer. **Exo** (Chapters 3–7) is the language at the center of this thesis, designed to embody this philosophy. Exo enables optimization through *user scheduling*: programmers rewrite Exo programs into functionally equivalent versions that execute much faster by specifying scheduling operations. By guaranteeing that these rewrites preserve functional equivalence, Exo spares developers from the tedious task of debugging optimized code. While Exo is not the first user-schedulable language (USL), it is distinctive in how far it pushes user control through two key designs: both the object language (describing computation) and the scheduling operators (describing optimization) are deliberately low-level. Performance-critical constructs, including control flow and data copies across memory hierarchies, are made explicit as statements in Exo's imperative object code. At the same time, its fine-grained scheduling operators allow programmers to specify the exact optimizations (e.g., explicit instruction selection) needed to achieve peak performance.

Exocompilation calls for externalizing two traditionally hardcoded compiler responsibilities to user code: (1) targeting new hardware (Chapters 3, 4, and 7) and (2) implementing optimization strategies (Chapters 5 and 6). By shifting them to user code, compilers remain future-proof as hardware evolves, while developers gain the flexibility to tailor optimizations to their specific architectural needs. This approach reconciles the long-standing trade-off between control and automation: the core compiler stays minimal, auditable, and predictable, while richer automation emerges through reusable user-defined libraries. Although Exo's low-level scheduling operators offer maximum performance transparency and fine-grained control, they are cumbersome to write manually. Exocompilation reduces the burden of manual scheduling by reintroducing automation through user extensibility; capturing common scheduling patterns as user-defined functions that can be reused across kernels. This design makes tasks such as vectorization and bound inference—which are typically built into other compilers—entirely modifiable and extensible in the user code.

(1) Externalizing Hardware Definition from Compilers

After the end of Dennard scaling, the industry's shift to multicore processors and specialized accelerators triggered rapid hardware innovation. Accelerators like GPUs, TPUs, Trainium, Arm's SME, and Intel AMX now release new architectures annually, exposing increasingly complex features, such as out-of-order execution, directly to software. This, in turn, demands new, specialized compiler optimizations. However, expecting traditional compiler backends to keep pace is untenable; multiyear compiler-development cycles cannot continuously retune for every fast-moving hardware generation.

In Chapter 4, we demonstrate that diverse and specialized hardware targets can be defined entirely in user or library code, external to the compiler. Programmers specify

their target hardware through APIs that model instructions, memory, and configuration state. These specifications can then be used during optimization via user scheduling and are safety-checked by the compiler. For example, object code can be replaced with calls to user-defined hardware instructions using an explicit instruction-selection scheduling operator (i.e., replace). Correctness is ensured by unification modulo affine equalities, which guarantees the safe use of intrinsics and automatically synthesizes correct argument indexing. This design philosophy extends beyond Exo and could transform compiler design more broadly. By treating hardware targets as standard user libraries rather than hardcoded compiler components, performance engineers can rapidly target new accelerators without modifying the compiler's core. Using this approach, we used Exo to externalize backends for real-world vector and matrix engines—including AVX2, AVX-512, Arm Neon, RVV, Arm SME, UC Berkeley's Gemmini, Apple's outer-product engine, and Intel AMX—while matching or exceeding the performance of existing HPC libraries and significantly reducing code size.

Modern GPUs pose even greater challenges for compilers than vector and matrix accelerators, requiring not only SIMT-style parallelism but also software-managed concurrency between compute and data movement to reach maximum performance. Performance engineers must reason about subdividing work into the hierarchy of computational resources (threads, warps, warpgroups, blocks, clusters) and use asynchronous tensor core and memory copy instructions across the memory hierarchy (registers, tensor core accumulators, shared memory, global memory), which expose explicit out-of-order execution to software. GPU programming languages generally offer either direct low-level control without safety guarantees (e.g., CUDA C++ inline assembly or intrinsics) or easier-to-analyze, high-level abstractions (e.g., Triton's tile-based model) that hide asynchrony in the compiler backend and prevent engineers from maximizing performance through direct control.

Following the Exocompilation philosophy, we once again emphasize that giving users precise control is essential for achieving optimal GPU performance. In Chapter 7, we extend Exo's backend and programming model with concurrency support, providing a minimal abstraction over CUDA. Asynchronous instructions are exposed to users in the object language, with synchronization safety guaranteed through *sequential-parallel equivalence*—meaning the parallel interpretation of the code remains functionally equivalent to its sequential interpretation, stripped of parallel loops and synchronous instructions. This guarantee is enforced through synchronization checking on an abstract machine model, which tracks effects on buffer elements and verifies the absence of hazards and races. We implemented GEMM kernel variants on Hopper GPUs, achieving performance on par with cuBLAS and Triton while statically catching and eliminating elusive synchronization bugs that would typically be extremely difficult to diagnose.

(2) Externalizing Optimization from Compilers

Exo gives this low-level control through a set of *scheduling primitives* that the compiler provides natively: the explicit instruction selection described above, along with more than forty other loop- and statement-modifying primitives. These primitives are designed to give performance engineers maximum control at the cost of productivity. Exo embraces this tradeoff: automation is achieved by users building higher-level scheduling operations as

functions composed of these primitives, enabling complex optimizations like vectorization and bound inference directly in user code without modifying the compiler. Because user-defined functions are built from safety-checked primitives, they compose trivially and preserve functional equivalence.

In Chapter 5, we identify three properties as essential for any user-extensible scheduling-language design: actions (ways of modifying code), inspection (ways of querying code), and references (ways of pointing to code). These features converge in a time-stable reference mechanism we developed called *Cursors*, which parameterizes scheduling metaprogramming and enables reusable, user-defined scheduling functions. With Cursors, optimization effort can be amortized across families of kernels, making it possible to build scheduling libraries in user code. For instance, we used this approach to develop more than 80 high-performance kernels, including different levels of BLAS and GEMM, achieving an order-of-magnitude reduction in code size through reuse of higher-level scheduling operators (e.g., vectorize), while matching or surpassing the performance of MKL, OpenBLAS, and BLIS.

While this design makes scheduling operators extensible, the expressivity of scheduling optimizations in Exo, as in other USLs, is fundamentally constrained by the analyses used to guarantee functional equivalence. Dependency analysis is a common tool for reasoning about array-based loops, but such conservative overapproximation fails to model the safety of many of the optimizations we need, such as sliding-window optimizations or the removal of statements with mutations. Ensuring the safety of these operations requires analyzing the values written by the statements being modified. This approach, which we term *value-sensitive analysis*, broadens the expressive power of imperative USLs beyond what dependency and polyhedral methods allow. As with any static analysis, the challenge is striking a balance between accuracy and computational cost by designing analyses that are simultaneously sound, precise, and efficient. In Chapter 6, we show that a cylindrical algebraic decomposition (CAD) tree abstract domain achieves the right balance, yielding both precision and efficiency for inferring hypotheses that are required to verify scheduling operations. The CAD-tree abstract domain not only scales more efficiently than alternatives in abstract interpretation but also provides the scheduling expressivity needed for unconventional architectures requiring value-sensitive methods, such as emerging variable-length vector architectures.

Chapter Overview

The main chapters of this thesis are as follows:

- Chapter 2 provides the necessary background for the rest of this thesis, introducing the key considerations in performance engineering through two examples: vector machines (AVX2) and the Gemmini accelerator.

- Chapter 3 introduces Exo, covering its object language and scheduling language, including our notion of program equivalence and the full set of scheduling primitives. Chapter 4 then shows how hardware targets are externalized from the compiler, defining the Gemmini accelerator in user code through Exo's instruction, memory, and configuration APIs.

- Chapter 5 distills the essence of a user-extensible scheduling language into three core mechanisms: Action, Inspection, and Reference. It then presents our scheduling libraries, built from these three mechanisms, which externalize optimization from the compiler.
- Chapter 6 motivates the need for value-sensitive analysis, derives Prexo IR (a space–time SSA analysis IR) from Exo IR, and develops a value-sensitive analysis over the CAD-tree abstract domain.
- Chapter 7 extends the Exo IR to the GPU IR for tensor cores and defines its syntax. It then guarantees sequential-parallel equivalence for asynchronous GPU code via abstract-machine checking and analyzes its compile-time overhead.

Prior Publication

This thesis contains work that also appeared in the following publications.

- *Exocompilation for Productive Programming of Hardware Accelerators* [29], at PLDI'22. This paper contributes the design of Exo's object and scheduling languages (Chapter 3), the externalization of hardware targets from the compiler through Exo's instruction, memory, and configuration APIs and an empirical evaluation (Chapter 4).
- *Exo 2: Growing a Scheduling Language* [30], at ASPLOS'25. This paper introduces the core mechanisms of a user-extensible scheduling language—Action, Inspection, and Reference—and scheduling libraries that externalize optimization from the compiler (Chapter 5). It also contributes the scheduling primitives used throughout the thesis (Chapter 4) and an empirical evaluation (Chapter 5).
- *Value-Sensitive Analysis for Looping Array Code* [31], in submission. This paper presents the extension of the Exo IR to the Prexo IR for value-sensitive analysis and a value-sensitive analysis over the CAD-tree abstract domain, with an empirical evaluation (Chapter 6).
- *Exo-GPU: Safe, Imperative, User-schedulable Programming for Tensor Cores* [32], in submission. This paper contributes the extension of the Exo IR to the GPU IR for tensor cores, the abstract-machine checking that guarantees sequential-parallel equivalence for asynchronous GPU code, and an empirical evaluation (Chapter 7).

Chapter 2

Performance-Optimization Preliminaries

This chapter provides the background for the rest of this dissertation, introducing the key considerations in performance optimization through two hardware targets: a general-purpose CPU with AVX2 vector instructions and the Gemmini accelerator.

Reaching the hardware's peak performance is a software problem. The hardware fixes a ceiling on throughput, but attaining it falls to performance engineers, who restructure each program by hand to fit the machine it runs on. This chapter uses the two machines to expose the considerations that govern high-performance code and the challenges they create, which motivate the design of Exo and Exocompilation in the chapters that follow. These examples show that even a simple computation demands intricate optimization, that the same high-level strategies recur across machines, that each strategy must be realized in hardware-specific code, and that keeping such code correct as it is aggressively rewritten is a genuine difficulty in its own right.

We use a single running example program throughout the chapter: the no-transpose, unit-scalar general matrix–matrix multiplication (GEMM), which computes

$$C \coloneqq C + AB, \qquad A \in \mathbb{R}^{m \times k},\ B \in \mathbb{R}^{k \times n},\ C \in \mathbb{R}^{m \times n}, \tag{2.1}$$

performing $2mnk$ floating-point operations over $\mathcal{O}(mn + mk + kn)$ matrix elements. GEMM is an apt choice for three reasons. It is the computational workhorse of dense linear algebra and much of modern machine learning, so its performance matters in practice; it is simple enough to state in three lines yet rich enough to expose nearly every important optimization; and, because it is so fundamental, it is among the first programs implemented for any new architecture. Optimizing this program for two different machines shows that the same high-level strategy must be realized through entirely different optimizations on each.

The remainder of this chapter is organized as follows. Section 2.1 introduces the relationship between the two quantities that bound performance—floating-point throughput and memory bandwidth—through the roofline model. Section 2.2 then optimizes GEMM for a general-purpose CPU with AVX2 vector instructions, where the memory hierarchy is managed *implicitly* by hardware caches. Section 2.3 optimizes the same program for the Gemmini accelerator, where the memory hierarchy is managed *explicitly* in software. Finally, Section 2.4 draws out the properties that make performance engineering challenging, connecting them to the case for a new approach.

Level	Typical capacity	Access time
Registers	4000 B	200 ps
L1 cache	64 KB	1 ns
L2 cache	256 KB	3–10 ns
L3 cache	16–64 MB	10–20 ns
Main memory	32–256 GB	50–100 ns

Table 2.1: Representative characteristics of the memory hierarchy of a server-class processor [33]. Sizes and access times are approximate and vary by implementation.

2.1 Compute and Memory

A processor has two resources that a program can run short of: *compute* (the rate at which it performs floating-point and other operations) and *memory* (the rate at which it moves data). Floating-point operations, such as addition and multiplication, are by far the cheaper of the two. A fused multiply–add completes in a few cycles, whereas a value fetched from main memory (DRAM) arrives hundreds of cycles later [33]. Consequently, almost every optimization in this chapter has a single underlying purpose, *to keep the floating-point units fed*. Because operands arrive slowly, this means both reducing the volume of data moved and hiding the latency of the movement that remains, so that the compute units rarely stall waiting for data. The techniques differ from machine to machine, but this goal does not.

The memory hierarchy. Fast memory is small, and large memory is slow. Hardware therefore interposes a sequence of progressively larger, slower memories between the computation core and DRAM, running from a few kilobytes of registers through L1, L2, and L3 caches of increasing size and latency, as summarized in Table 2.1. The hierarchy speeds up a program only to the extent that the program exhibits locality: *temporal* locality, reusing a datum while it is still resident in a fast level; and *spatial* locality, using data adjacent to data already fetched (hardware moves memory in fixed-size cache lines, so neighboring elements come for free). A program with poor locality stalls, repeatedly waiting on DRAM no matter how fast its floating-point units are. The crux is that locality is a property of how a computation touches memory (both its access order and its data layout), not of the result it computes, so it must be engineered without changing what the program computes.

The Roofline model. Whether a floating-point program is limited by floating-point throughput or by memory bandwidth is captured by the *Roofline* model [34], shown in Figure 2.1. The model relates two quantities that belong to the hardware, peak floating-point performance and peak memory bandwidth, to one quantity that characterizes the running program: its *operational intensity*, the number of operations per byte of DRAM traffic (flop/DRAM byte for floating-point programs). The byte count is not the traffic between the processor and the caches; it is the traffic that reaches main memory after being filtered by the cache hierarchy. Thus, operational intensity is not just an algorithmic operation count divided by the nominal input size. It also depends on how the implementation and the memory hierarchy translate the program's access pattern into DRAM traffic.

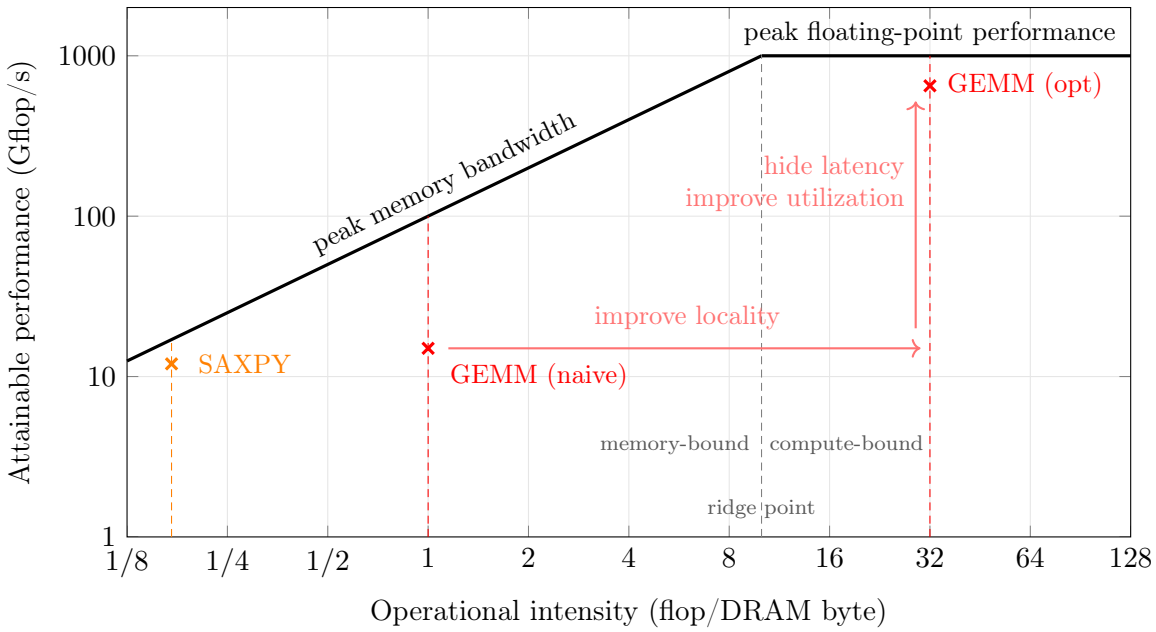


Figure 2.1: The Roofline model [34]. The two hardware limits are peak floating-point performance and peak memory bandwidth. Dashed vertical lines indicate a program's operational intensity, measured as floating-point operations per byte of DRAM traffic; the marks on those lines indicate schematic attained performance. SAXPY has little temporal reuse, so its operational intensity remains low and, on machines whose ridge point lies to its right, its attainable performance is bounded by the bandwidth roof. GEMM can reuse $\mathcal{O}(n^2)$ data across $\mathcal{O}(n^3)$ operations, but a naive implementation generates excess DRAM traffic from poor locality and therefore has low operational intensity. Blocking and packing improve locality, reducing DRAM traffic and moving GEMM rightward by increasing operational intensity. Register blocking, unrolling, and instruction scheduling hide remaining latency and improve utilization, moving the implementation upward toward the floating-point roof.

For a program with operational intensity I, the Roofline upper bound is

$$P \le \min\{P_{\text{peak}},\, B_{\text{peak}} I\}, \tag{2.2}$$

where P_{peak} is peak floating-point performance (the maximum rate at which the machine can execute floating-point operations, in flop/s) and B_{peak} is peak memory bandwidth (in bytes/s). In Figure 2.1, this bound appears as a sloped bandwidth roof at low operational intensity and a horizontal floating-point-performance roof at high operational intensity. The two roofs meet at the ridge point, whose x-coordinate is the minimum operational intensity required to reach peak floating-point performance. A program whose operational intensity lies to the left of the ridge point is ultimately memory-bound; one whose operational intensity lies to the right can, in principle, become compute-bound. The Roofline therefore gives a theoretical upper bound and an optimization target: the performance of a program must lie on or below the roof above its operational intensity.

Some programs, however, have too little data reuse to raise their operational intensity far enough to reach the ridge point. The BLAS level-1 operation SAXPY, $y \leftarrow ax + y$, performs

```
// C is m-by-n, A is m-by-k, B is k-by-n; all stored row-major.
for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    for (int p = 0; p < k; ++p)
      C[i*n + j] += A[i*k + p] * B[p*n + j];
```

Listing 1: Naive GEMM, written in C. Correct, but far from the Roofline.

two floating-point operations per element while streaming the input vector x, reading the old value of y, and writing the new value of y. In single precision, this simple traffic count gives roughly $2/12 \approx 0.17$ flop/DRAM byte. On machines whose ridge point is larger than this value, SAXPY remains on the sloped part of the Roofline. Optimization can help it approach the available bandwidth, but the program has little temporal reuse with which to substantially improve its operational intensity.

GEMM has much greater reuse potential. It performs $\mathcal{O}(n^3)$ floating-point operations on $\mathcal{O}(n^2)$ data, so a well-structured implementation can reuse matrix elements many times after they have been brought on-chip. A naive implementation fails to exploit this reuse: it repeatedly brings data through the memory hierarchy, increasing DRAM traffic and lowering the operational intensity of the program. Blocking and packing are the transformations that reduce this traffic by keeping reused operands in faster storage. In the Roofline view, reducing this traffic raises GEMM's operational intensity and shifts the program rightward, out of the memory-bound region. Hiding the memory latency that remains does not change the operational intensity; it instead keeps the compute units from stalling so the program can approach the floating-point-performance roof above its current intensity (Figure 2.1).

2.2 Optimizing for a CPU with AVX2 vector instructions

Our first target is a general-purpose CPU equipped with the AVX2 instruction set, which provides 256-bit vector registers and fused multiply–add (FMA) instructions. In performance engineering, a small, heavily optimized routine like the one we are about to build is called a *kernel*, and we use the term throughout the dissertation. In this chapter, kernels are written in ordinary C with compiler intrinsics. We begin from the textbook statement of GEMM and progressively transform it, noting at each step which resource the transformation relieves. Listing 1 shows the starting code.

This implementation is correct but slow, for reasons that map directly onto the two resources above. On the memory side, the innermost loop indexes `B[p*n + j]`, which strides through memory by a full row (n elements) each iteration. In row-major storage the elements of a column are far apart, so each access falls in a different cache line and, for large matrices, misses cache almost every time (Figure 2.2). On the compute side, each iteration performs a single scalar multiply–add, leaving the vector units idle and the arithmetic pipeline starved. The transformations below address these problems in turn.

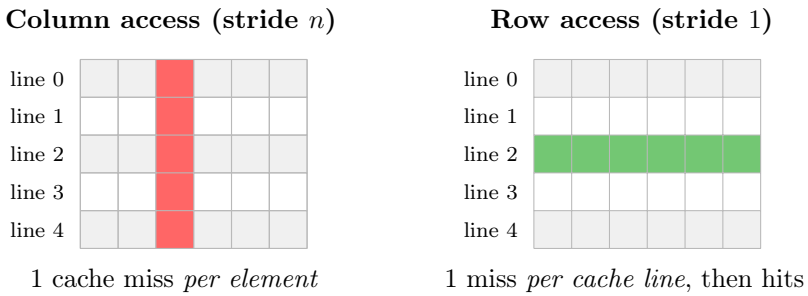


Figure 2.2: Why access order matters, for a matrix stored row-major (one row drawn as one cache line, a simplification). Walking down a *column* (left, the naive inner loop's access to B) touches a different cache line at every step, missing cache on each element. Walking along a *row* (right) stays within a cache line, paying a single miss and then hitting for the rest of the line. Loop reordering converts the former into the latter.

```
for (int i = 0; i < m; ++i)
  for (int p = 0; p < k; ++p)
    for (int j = 0; j < n; ++j)
      C[i*n + j] += A[i*k + p] * B[p*n + j];
```

Listing 2: GEMM after loop interchange. The inner loop now has unit-stride accesses.

Loop order. The cheapest improvement is to reorder the loops. Interchanging the j and p loops yields the i, p, j order of Listing 2. Now the innermost loop over j accesses `C[i*n + j]` and `B[p*n + j]` with unit stride (walking along rows rather than down columns; see Figure 2.2) while `A[i*k + p]` is loop-invariant. This single change restores spatial locality in the inner loop and, as a bonus, exposes a vectorizable pattern. No arithmetic has changed, only the order in which it is performed.

Cache blocking. Reordering improves spatial locality, but it does not create temporal reuse. For large matrices, an operand evicted from cache must be refetched from DRAM before it is used again. *Cache blocking* (or tiling) restores reuse by partitioning the iteration space into tiles small enough that the data touched by a tile fits in a given cache level and is reused there before being evicted [35]. Because the hierarchy has several levels, high-performance implementations block at several levels at once, forming a nest of progressively smaller tiles [36,37]. A typical scheme keeps a large panel of B resident in L3, a panel of A in L2, and a smaller block in L1, with the register-level tile of the micro-kernel below (discussed next) as the innermost level. Each level exists to create reuse at its memory: a block of operands, once loaded, is reused for many flops before being evicted, which spares the DRAM traffic that refetching it would cost and so raises GEMM's operational intensity. None of the block sizes is a free parameter, since each is bounded by the capacity of the level it targets, so choosing them well requires knowing the exact size of every cache.

```
// Accumulators for a 6*16 tile of C:
// c[i][0] holds columns 0..7 of row i;
// c[i][1] holds columns 8..15 of row i.
__m256 c[6][2];

for (int i = 0; i < 6; ++i) {
  c[i][0] = _mm256_loadu_ps(&C[i*ldc + 0]);
  c[i][1] = _mm256_loadu_ps(&C[i*ldc + 8]);
}

for (int p = 0; p < Kc; ++p) {
  __m256 b[2];
  b[0] = _mm256_load_ps(&B_pack[p*16 + 0]);   // contiguous, aligned
  b[1] = _mm256_load_ps(&B_pack[p*16 + 8]);

  for (int i = 0; i < 6; ++i) {
    __m256 a = _mm256_broadcast_ss(&A_pack[p*6 + i]);
    c[i][0] = _mm256_fmadd_ps(a, b[0], c[i][0]);
    c[i][1] = _mm256_fmadd_ps(a, b[1], c[i][1]);
  }
}

// Store c[0][0]..c[5][1] back to the corresponding 6*16 block of C.
```

Listing 3: Micro-kernel sketch (single precision, 6×16 tile). The accumulator array holds the tile as twelve vector values, two `__m256` vectors per row. Each iteration of the K-loop performs a rank-1 update of the tile: it loads a 16-float row of packed B, broadcasts the six scalars (each copied into all eight lanes of a vector, one per row) from the corresponding column of packed A, and updates all six rows.

Vectorization. With the inner loop made unit-stride, the computation can be mapped onto AVX2's vector units, replacing eight scalar single-precision FMAs with a single 256-bit vector FMA. This exposes a practical requirement of efficient vectorization: the values consumed by a vector instruction should be available through unit-stride loads or through inexpensive broadcasts and register rearrangements. A row-major layout makes only one dimension contiguous, so a microkernel's operands are not necessarily laid out in the order the vector unit consumes them. This motivates the packing step below, which rearranges matrix panels into contiguous, microkernel-friendly order. Vectorization is also the first optimization that is fundamentally instruction-set-specific, tied to AVX2's register width and FMA semantics.

Register blocking and the micro-kernel. Beneath the cache blocks sits the innermost tier of the hierarchy: a small *micro-kernel* that computes a fixed-size $M_r \times N_r$ tile of C while holding that tile entirely in vector registers across the whole reduction over p. Keeping the accumulators in registers means each is read and written once per p step rather than streamed to and from cache, maximizing reuse at the very top of the hierarchy. Listing 3 sketches such a kernel for a 6×16 tile in single precision.

The size and shape of the tile are dictated by two microarchitectural facts at once. First, the accumulators must fit in the architectural register file. The whole point of the micro-

kernel is to hold the output tile in registers across the reduction; if the tile is too large, the compiler is forced to *spill* some accumulators to the stack, reintroducing the memory traffic the micro-kernel exists to avoid. A 6×16 single-precision tile occupies twelve of AVX2's sixteen 256-bit registers, leaving just enough for the operands being multiplied in, and no more. Second, the FMA unit is *pipelined*: an individual FMA has a latency of several cycles before its result is available, yet the unit can begin a new, independent FMA every cycle (two per cycle on machines with two FMA ports) [38]. To keep the pipeline full, the kernel must have enough independent accumulator chains in flight to cover the latency–throughput product; with a latency of roughly five cycles and two FMAs issued per cycle, on the order of ten independent accumulators are needed. The twelve accumulators of the 6×16 tile are chosen to clear this threshold. A kernel with too few would stall on FMA latency despite issuing the same number of instructions.

Unrolling. Unrolling the reduction loop replicates its body for several values of p and advances the counter in larger steps. This removes loop-control overhead and, more importantly, gives the processor's scheduler a larger window of independent instructions to interleave, helping to sustain the FMA throughput targeted by the register-blocking step.

Packing. The final ingredient ties several of the above together. Before invoking the micro-kernel, the relevant blocks of A and B are copied into small, contiguous buffers (`A_pack` and `B_pack` in Listing 3), laid out in exactly the order the kernel will stream them [36,37]. Packing makes the inner-loop accesses unit-stride and alignment-friendly and improves the behavior of the caches. Although packing copies data, the cost is paid once and amortized over the $\mathcal{O}(n^3)$ arithmetic of the block, so it is a net win for all but the smallest problems.

The cost of all this. The progression from Listing 1 to a packed, vectorized, multi-level blocked kernel can improve performance by orders of magnitude, but it leaves behind a program that bears little resemblance to Equation (2.1). Worse, the choices are deeply entangled. The micro-kernel dimensions M_r and N_r are bounded by the register count but also set the ratio of memory operations to FMAs, which decides whether the kernel is limited by the load units or the floating-point units; the cache block sizes depend on all three cache capacities; the unroll factor interacts with both. None of these parameters can be chosen in isolation, and the optimum shifts with every change in microarchitecture—a new cache size, a different FMA latency, an extra vector port.

2.3 Optimizing for a Specialized Accelerator (Gemmini)

Gemmini is an open-source generator of *systolic-array* accelerators for deep learning [39], integrated into a full RISC-V system. It merits close attention for two reasons. First, systolic arrays are widely adopted in machine-learning hardware. Google's TPU is built around a systolic matrix unit [3], and AWS's Trainium accelerators likewise center their Tensor Engine on a systolic array [40]. Second, the traits that make Gemmini hard to program are not unique to systolic designs: even a non-systolic accelerator such as a GPU reaches peak performance only when software explicitly overlaps data movement with computation.

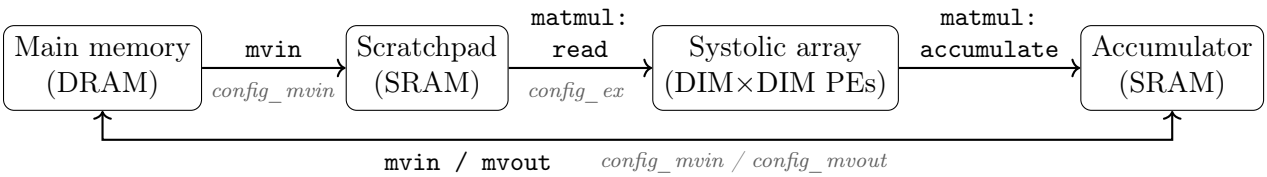


Figure 2.3: Simplified structure of the Gemmini accelerator [39]. Each data movement is driven by an explicit instruction: `mvin` moves operand tiles from DRAM into the scratchpad and may also seed the accumulator from the old C tile, while `mvout` reads the accumulator and writes finished results back to DRAM. The `matmul` instruction reads operands from the scratchpad and accumulates partial sums into the accumulator. `config_*` instructions (italic) set the mutable state that governs subsequent `mvin`, `matmul`, and `mvout` operations, including strides, dataflow, output scaling, and activation.

Where the CPU offered general-purpose vector instructions over a hardware-managed cache, Gemmini offers a fixed-function matrix engine over a memory hierarchy that software must orchestrate explicitly (Figure 2.3). Gemmini's compute primitive is a small matrix multiply: a two-dimensional array of DIM × DIM processing elements (here 16×16) that multiplies 16×16 operand tiles in a systolic fashion. The on-chip memory is a software-managed *scratchpad* (256 KB, holding operands) and a separate *accumulator* (64 KB, holding 32-bit partial sums); unlike a cache, nothing enters or leaves them except by explicit instruction. And the engine is *stateful*: configuration registers, set by their own instructions, determine operand strides, the dataflow, the quantization scaling, and the activation function applied on output. The optimizations below turn the naive GEMM of Listing 1 into a program expressed entirely in these terms. An illustrative optimized code is sketched in Listing 4.

Tiling to the on-chip memories. The first task is to decompose the multiplication so its working set fits the fixed on-chip SRAMs. The loop nest is tiled so a tile of the output occupies the 64 KB accumulator, and the operand tiles occupy the 256 KB scratchpad (A and B take half apiece, roughly 128 KB each). This is the counterpart of cache blocking from Section 2.2: it creates reuse for the identical purpose, but the engineer sizes the tiles against hard SRAM capacities rather than coaxing a cache, and, because the array consumes 16×16 tiles, the innermost dimension of every on-chip buffer must be a multiple of sixteen.

Staging data explicitly and selecting hardware instructions. With tiles chosen, the data must actually be moved. Operands are brought from DRAM into the scratchpad with `mvin`, the old C tile is brought from DRAM into the accumulator with another `mvin`, partial sums are kept in the accumulator, and finished results are written back with `mvout`. The arithmetic is then mapped onto Gemmini's instruction set: seed the accumulator from C, load operands, issue accumulating `matmul` operations on 16×16 tiles, and finally `mvout` the result. Choosing the right instructions is itself an optimization. Although the array multiplies 16×16 tiles, Gemmini also provides a blocked `mvin` form that brings a $4 \times 16 \times 16$ block of operands on-chip as a unit; exploiting it requires further dividing the contraction and column loops by four so the loads line up with the instruction.

```
// int8 in-place GEMM on Gemmini: C <- act(requant(C + A*B)).
enum { ACC_TILES = 64 };       // 64 * 16 * 16 * sizeof(int32_t) = 64 KB
enum { MVIN_TILES = 4 };       // blocked operand mvin covers 4 adjacent 16*16 tiles

config_ex(WEIGHT_STATIONARY, /*accumulate=*/true);
config_mvout(stride_C, act, scale);                      // mvout: acc -> DRAM, then scale/act
config_mvin(stride_C, /*id=*/0, /*dst=*/ACCUMULATOR);    // C mvin config
config_mvin(stride_A, /*id=*/1, /*dst=*/SCRATCHPAD);     // A mvin config
config_mvin(stride_B, /*id=*/2, /*dst=*/SCRATCHPAD);     // B mvin config

for (int io = 0; io < m; io += ROW_TILE)
  for (int jo = 0; jo < n; jo += COL_TILE)
    for (int i = 0; i < ROW_TILE; i += DIM)
      for (int j = 0; j < COL_TILE; j += ACC_TILES*DIM) {

        // Seed 64 KB of accumulator state from the old C panel.
        mvin(&C[(io + i) * stride_C + (jo + j)], acc_tile[0..ACC_TILES-1], /*id=*/0);

        for (int ko = 0; ko < k; ko += DIM) {

          if (jo == 0 && j == 0)
            mvin(&A[(io + i) * stride_A + ko], A_sp[i][ko], /*id=*/1);          // reused across cols

          for (int jj = 0; jj < ACC_TILES; jj += MVIN_TILES) {

            if (i == 0)
              mvin(&B[ko * stride_B + (jo + j + jj*DIM)],
                          B_sp[ko][j/DIM+jj..j/DIM+jj+MVIN_TILES-1], /*id=*/2); // reused across rows

            matmul_panel(A_sp[i][ko], B_sp[ko][j/DIM+jj..j/DIM+jj+MVIN_TILES-1],
                                 acc_tile[jj..jj+MVIN_TILES-1]);                // expands to 4 16×16 matmuls
          }
        }
        mvout(acc_tile[0..ACC_TILES-1], &C[(io + i) * stride_C + (jo + j)]);
      }
fence();
```

Listing 4: Pseudocode for in-place quantized GEMM on Gemmini. A and B panels are loaded into the scratchpad with `mvin` and reused across independent column and row subtiles. Each output panel first seeds 64 KB of accumulator state by moving the old C panel into the accumulator with `mvin`; the weight-stationary `matmul` sequence then writes products to the same accumulator addresses with accumulation enabled. The final `mvout` reads the accumulator, requantizes to int8, applies the configured activation, and writes C in-place.

Managing configuration state. Gemmini is a stateful accelerator, and managing that state is a concern with no counterpart on a CPU. Operand strides, the compute mode, the quantization scale, and the activation function all live in configuration registers, set by `config_*` instructions, and the behavior of every `mvin`, `matmul`, and `mvout` depends on them. A naive lowering reissues the configuration before each operation (a `config_mvin` before every `mvin`, a `config_mvout` before every `mvout`, and so on), which is correct but expensive: reconfiguring the accelerator can force its pipelines and queues to drain, so a redundant configuration instruction costs a full stall rather than a single wasted issue slot. The configuration instructions must instead be *hoisted* out of the inner loops so that the state is established once and reused, as shown at the top of Listing 4. Because configuration is shared mutable state that every instruction reads implicitly, hoisting these instructions without introducing bugs is difficult: the dependency never appears in an instruction's operands, so nothing in the program text prevents a hoist that leaves some operation running

under the wrong configuration. This thesis treats configuration as a first-class part of the hardware model and shows how such hoisting is expressed and checked to preserve the kernel's behavior (Chapter 4).

Quantization and fused post-processing. Because Gemmini targets machine-learning inference, it operates on 8-bit integer operands with 32-bit accumulation, and as each output tile leaves the accumulator through `mvout` it is rescaled to 8 bits, clamped, and optionally passed through a ReLU activation.

Eliminating redundant data movement. Across the tiled loop nest, the same operand tiles are required by many matrix multiplies, and reloading them on every iteration would squander the scarce bandwidth between DRAM and the scratchpad. Guards are introduced so that each tile is loaded only when it actually changes, realizing on-chip reuse explicitly, the hand-managed analogue of a cache hit.

Balancing the instruction mix. Finally, the order of instructions matters for throughput. If the issued stream is, say, a long run of `mvin` operations, the processor's reorder buffer fills with loads, and the accelerator stalls for lack of executable work; the loops are therefore fused to interleave `mvin`, `matmul`, and `mvout` operations into a balanced mix, and the innermost loops are unrolled to expose instruction-level parallelism and overlap data movement with computation. This is the accelerator's counterpart to the latency hiding that register blocking and software pipelining provide on the CPU. And, as in a CPU-only kernel, it is essential rather than optional, since without it the expensive matrix unit spends much of its time idle.

2.4 What Makes Performance Engineering Hard

The two case studies optimized one three-line kernel for two machines, producing two long, intricate programs that share almost no code. Yet, as we just saw, they share a great deal at the level of strategy: both create reuse by blocking, both stage operands through the memory hierarchy to keep the compute units fed, and both rely on selecting the right hardware instructions. The high-level strategies are general and reusable, but every concrete realization is bound tightly to one machine, so today the same strategy must be reimplemented by hand for each new target. In what follows, we explain why high-performance kernels are written the way they are today.

A vast, interacting, hardware-specific space. Each kernel exposed many parameters, including tile sizes at several levels, micro-kernel dimensions, unroll factors, instruction choices, configuration, and buffering, whose combinations form an enormous space. The parameters are not separable: on the CPU the micro-kernel dimensions were at once bounded by the register file and decisive for the load-to-arithmetic balance, while the block sizes were pinned to cache capacities; on Gemmini the tile sizes were pinned to SRAM capacities and entangled with instruction selection. A choice that is locally optimal can be globally poor. And every one of these choices is tied to specifics of the hardware (e.g., register counts, cache sizes, FMA

latency, array dimension, and scratchpad capacity) so the optimum shifts with each new chip. This is why a production library such as OpenBLAS carries scores of separately hand-tuned GEMM kernels, one per microarchitecture, amounting to over a million lines of code [6]: the machine-specific tuning does not transfer and must be redone for each target.

The abstraction–performance gap and the limits of automation. The optimized kernels no longer resemble Equation (2.1); the very transformations that deliver performance obscure the specification beneath them, so that speed is purchased at the cost of readability and maintainability. The natural response is to hide these transformations inside a compiler and let it optimize automatically. But automation necessarily embeds a particular set of optimizations into the compiler: it takes *responsibility* for optimization away from the programmer, and unfortunately, simultaneously takes away *control* over it. This is especially problematic for new or unusual hardware. As the examples in this chapter showed, useful optimizations depend on hardware details such as register counts, cache sizes, FMA latency, vector width, scratchpad capacity, matrix-instruction shape, configuration state, and the ability to overlap data movement with computation. A compiler cannot reliably foresee every optimization that a future target will require. The cost of automation failure is that the optimization needed to achieve peak performance is simply not available through the programming model [8,25,26].

Fragility of correctness. There is a subtler difficulty underlying all of this. Every optimization in this chapter rewrote the program while it was supposed to preserve the result of Equation (2.1). Each rewrite is an opportunity to introduce a subtle bug: a missized tile, a transposed pack, an off-by-one in an `mvin`, or, on a stateful accelerator like Gemmini, a configuration register left holding the wrong value after being hoisted. The cumulative correctness of dozens of such steps is difficult to establish by testing alone, and worse on GPUs, where asynchronous operators let a bug surface only under a particular interleaving (Chapter 7). As optimizations grow more aggressive, ensuring that the fast program still computes what the slow one did becomes a first-order concern and one that consumes much of the effort of writing high-performance code.

Toward a different approach. These are the structural reasons that hand-written high-performance code is expensive to produce and costly to port. The fragility of correctness, in particular, is the burden that a user-schedulable language is built to remove: by guaranteeing that every transformation preserves the program's result, correctness ceases to be something the engineer must reestablish by hand after each change. Rather than chasing ever-more automation that eventually fails, Exo (Chapters 3–7) is a language that gives the performance engineer explicit, low-level control over exactly the decisions this chapter made, namely instruction selection, data movement across the memory hierarchy, and configuration. It externalizes from the compiler the two responsibilities that make these kernels hardware-specific and compiler-bound: the definition of the hardware itself (its instructions, memories, and configuration state, as we will define Gemmini in user code in Chapter 4); and the optimization strategies, captured as reusable libraries rather than baked into the compiler (Chapter 5).

Chapter 3

Exo Foundations

Exo is a domain-specific language for writing and optimizing high-performance kernels.[1] From the user's point of view, the Exo compiler consists of three main parts: an imperative *object language* for writing programs, libraries for describing hardware targets, and a rewrite-based *scheduling language* for transforming object programs into optimized implementations. The object language is the source language of the programs being optimized; hardware libraries describe target-specific memories and instructions; and schedules are metaprograms that rewrite object programs while preserving their meanings.

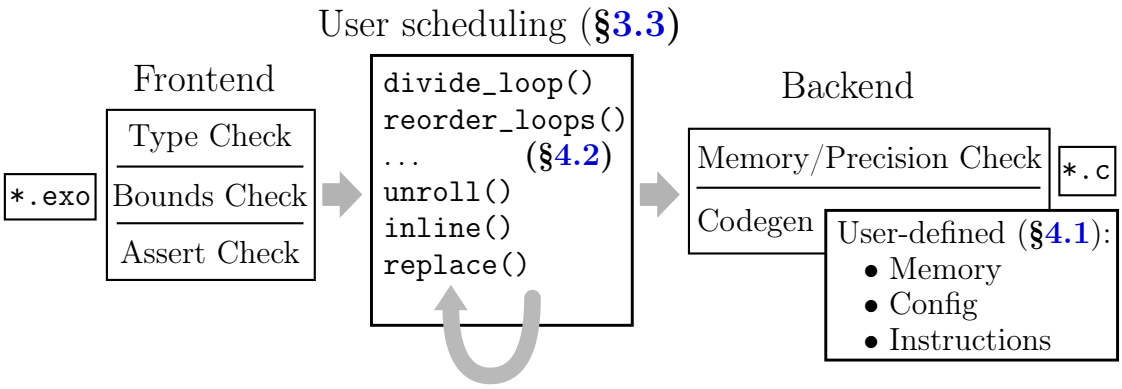


Figure 3.1: Exo compiler overview.

Figure 3.1 shows the compiler system from the standpoint of a single program being compiled. Exo object code (Exo IR) first passes type, bounds, and assertion checks (Sections 3.1 and 3.2); the user then rewrites the Exo IR through a series of scheduling transformations, such as `reorder_loops()` (Section 3.3); finally, the backend performs precision and memory analyses and emits a C program. Deferring precision and memory analyses until the very end is deliberate: the validity checks that justify scheduling rewrites are insensitive to those details, so they need not be settled until code generation.

This chapter introduces the Exo IR object language, both through examples and through its formal definition (Sections 3.1 and 3.2), provides an overview of the rewrite-based scheduling system (Section 3.3), and presents the effect analysis used to reason about equivalence during program transformations (Section 3.4).

[1] Exo is open source under the MIT license and available at https://github.com/exo-lang/exo.

```
@proc
def gemm(M: size, N: size, K: size,
         A: f32[M, K] @ DRAM,
         B: f32[K, N] @ DRAM,
         C: f32[M, N] @ DRAM):
    assert M % 8 == 0
    assert N % 8 == 0
    assert K % 8 == 0
    for i in seq(0, M):
        for j in seq(0, N):
            for k in seq(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

Figure 3.2: An Exo object-language procedure implementing matrix-matrix multiplication, used as the running example throughout this chapter and the next.

3.1 Exo IR Object Language

Exo IR is a familiar imperative language in the mold of the static control program model [41]. It supports for-loops, if-conditions, arrays, and procedures, but neither while-loops nor recursion. Its surface syntax is embedded in Python, and a BNF grammar for its formal core appears in Figure 3.3; beyond that core, the full language additionally supports `stride` values and expressions as well as memory annotations. We introduce the language through the matrix-matrix multiplication procedure `gemm` (Figure 3.2), which serves as a running example.

The `@proc` decorator marks the beginning of an Exo procedure, and its body is written using Exo statements. Both procedure arguments and variable declarations follow the form

$$\langle name \rangle : \langle type \rangle \texttt{[} \langle size \rangle \texttt{]} \texttt{ @ } \langle memory \rangle .$$

For example, `A: f32[M, K] @ DRAM` declares `A` to be a two-dimensional array of 32-bit floating-point values residing in the memory space `DRAM`. Here `f32` is a concrete numeric type; Exo also provides an abstract numeric type `R` that scheduling can specialize to a concrete precision such as `f32` or `i8`. The $\langle size \rangle$s may be constants or may refer dependently to other arguments, as `[M, K]` does, so the same procedure can be specialized to many matrix dimensions while remaining statically checked. The `@` symbol, followed by a memory-space identifier, specifies where the argument or variable resides (e.g. `@DRAM`). The `assert`s state preconditions on the sizes (here, that every dimension is a multiple of 8), a fact that later scheduling transformations may exploit. Finally, `for i in seq(0, M)` is a *sequential* for-loop iterating from 0 to $M - 1$, inclusive.

Exo IR compiles to `C` source code in the expected way:

```
void gemm(int M, int N, int K, float *A, float *B, float *C) {
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      for (int k = 0; k < K; k++) {
        C[N*i + j] += A[K*i + k] * B[N*k + j];
} } } }
```

Six relatively standard (but not universally adopted) features of Exo are worth discussing further: (1) control/data separation, (2) mutable global control state, (3) dependently typed arrays [42], (4) array windowing/slicing, (5) explicit `+=` reduction primitives, and (6) static assertions. Two of these are already visible in `gemm`: the sizes of `A`, `B`, and `C` are given dependently, and `+=` appears in the inner loop.

(1) Control/data separation. Exo is built around a distinction between control and data values. Control values (types `int`, `bool`, `size`, etc.) are constrained so that they may be analyzed precisely: arithmetic on integer control values must be quasi-affine, meaning that values may only be multiplied, divided, or modulo-ed by integer literals. Expressions appearing in loop bounds and branch conditions must be control values. This restriction allows Exo to translate control reasoning into satisfiability queries modulo linear integer arithmetic (LIA).

Data values include numerical types such as `R`, `f32`, and `i8`. They are stored in scalars, buffers, or windows and may be combined using ordinary numerical operations. Unlike control expressions, data computations are not required to be affine. However, Exo prohibits data-to-control dependences: control values may not depend on data values, whether the control values are local variables or mutable global configuration state. This restriction is what allows Exo to decidably reason about control flow, bounds, and preconditions independently of the particular numerical values stored in buffers.

(2) Mutable global control state. Configuration state (discussed in more detail in Chapter 4) is introduced via structs of variables declared with `@config` and is modeled formally as global variables (Section 3.2). Unlike the other sources of control values, configuration state is mutable and may be read or written at any point in a program. Consistent with the static-control program model, however, Exo prohibits any dependence of control values on data values, whether those control variables are local or global. Consequently, the values of configuration variables can be reasoned about independently of the contents of any buffer. Exo's analyses treat this state value-insensitively, reasoning about where configuration variables are written rather than the values they hold (Section 3.4.3); Chapter 6 lifts this to a value-sensitive analysis that also extends from scalar configuration state to arbitrary arrays.

(3) Dependently typed arrays. Array sizes may be specified by control-value expressions of strictly positive value. This design permits shape-parametric programs such as `gemm` while still allowing Exo to perform static bounds checking. If a program passes Exo's front-end checks, then all accesses to buffers and windows are guaranteed to be in-bounds, without inserting dynamic bounds checks into the generated code. This, too, is made possible by control/data separation.

(4) Windowing. In addition to ordinary array accesses, Exo supports array windowing, or slicing, using syntax such as `x[lo:hi]`. Creating a window does not copy data. Instead, the window aliases the underlying buffer: if `y = x[3:8]`, then reading `y[2]` reads the same location as `x[3+2]`. Windows may also have lower dimension than their underlying buffers, because some indices may be sliced while others are fixed. For example, `x[0:n, j]` creates

a one-dimensional window over column `j` of the matrix `x`. Since windows may refer to non-contiguous regions of memory, their runtime representation includes stride information, not only a base pointer and static size.

(5) Reduction. Exo treats reduction updates explicitly through the `+=` statement. In ordinary imperative code, `x += e` could be desugared into a read followed by a write. Exo instead treats reduction as a primitive operation, because reductions are ubiquitous in high-performance numerical kernels and because knowing that an update is associative and commutative gives the scheduler more freedom to reorder and restructure computations safely. In the `gemm` example, the statement

```
C[i, j] += A[i, k] * B[k, j]
```

makes the accumulation into `C[i, j]` explicit.

(6) Static assertions. Finally, assertions about control values may be stated at the beginning of a procedure. These act as *preconditions*, not as dynamic tests. Program analysis within a procedure may assume its asserted preconditions, whereas a calling procedure is valid only if it establishes that the callee's pre-conditions hold. Assertions thereby let us model the constraints of hardware instructions and the specialization of subroutines to particular sizes.

3.1.1 Frontend and Backend Checks

Exo performs several front-end checks before scheduling. Type checking ensures that variables, expressions, and procedure calls are well-formed. Bounds checking ensures that every buffer or window access is statically in-bounds. Assertion checking ensures that each procedure call satisfies the callee's preconditions. These checks guarantee that a well-typed, well-bounded, assertion-satisfying Exo program cannot crash because of an out-of-bounds access or failed static precondition.

Precision and memory annotations are checked late in the backend, immediately before code generation, after scheduling. Exo requires all data expressions to have consistent precision (multiplying an `f32` by an `i8` is forbidden) but inserts type casts as necessary just before writing or reducing data values. Because precision and memory are themselves scheduling choices, set by the `set_precision` and `set_memory` directives, they are ignored by the rewrite equivalence checks and are not validated until scheduling has finished. Chapter 7 extends the backend checks to asynchronous GPU programs. The sequential rewrite checks remain unchanged, while the collective, distributed-memory, and abstract-machine analyses verify that the added parallel annotations preserve the semantics of the sequential program.

3.1.2 Code Generation

Exo is designed to generate human-readable C code that is more or less a syntactic translation of the corresponding Exo code, which allows the programmer to integrate Exo with existing C-based tools and workflows. Any processing beyond what the scheduling directives explicitly request is left to the user's choice of C compiler. Three non-obvious details of this translation

merit discussion. First, all data values (scalars, buffers, and windows alike) are passed by pointer rather than by value; this is necessary even for scalars, in order to "return" modified scalar values to a caller. Second, windows are compiled to structs each containing both a data pointer and stride values, since the static size of a window is insufficient to compute a linear address into the underlying buffer. Lastly, static assertions are translated into compiler-specific optimization hints, which facilitate constant propagation and consequent dead-code elimination downstream. Since the Exo backend has already discharged these assertions, it is safe to hand them to the C compiler as assumptions.

3.2 The Formal Core of Exo

In order to define our program analysis, we now give a formal definition of the core of Exo, including a denotational semantics. The central idea is that statements denote store-transforming functions of type $\Sigma \to \Sigma$. With these semantics in hand, in Section 3.3 we define two Exo programs to be equivalent when their denotations are equal.

3.2.1 Mathematical Model of Exo Programs

We begin by defining the values and stores over which Exo programs operate.

Definition 3.1 (Names)**.** The namespace Name is partitioned into three disjoint sets:

$$\mathsf{Name} = \mathsf{Name}_{data} \uplus \mathsf{Name}_{local} \uplus \mathsf{Name}_{global}.$$

Names in Name_{data} identify buffers, names in Name_{local} identify procedure-local variables, and names in Name_{global} identify global control variables.

We refer to booleans $\mathbb{B}$ and integers $\mathbb{Z}$ as *control values*, and to real numbers[2] $\mathbb{R}$ as *data values*. For simplicity, we assume that built-in operations on data, including basic arithmetic and the mathematical library, are total; for example, $0/0$ does not cause the program to crash.

Definition 3.2 (Exceptional values)**.** We distinguish three kinds of exceptional values: an unknown or uninitialized value $\bot$, a memory error ϵ_m, and a type error ϵ_τ. Unknowns are typed, giving $\bot_{\mathbb{B}}$, $\bot_{\mathbb{Z}}$, and $\bot_{\mathbb{R}}$, although we write $\bot$ when the type is clear.

These values are equipped with an information order $\sqsubseteq$. Errors contain less information than unknowns, and unknowns contain less information than ordinary values of the corresponding types. In particular,

$$\epsilon_\tau \sqsubseteq \epsilon_m \sqsubseteq \bot_t \sqsubseteq v$$

for an ordinary value v of basic type t. Distinct ordinary values are otherwise incomparable. A well-typed program does not produce ϵ_τ, and a well-bounded program does not produce ϵ_m.

[2] As will be discussed in Section 4.1, our semantics is insensitive to how data values are approximated in finite precision. One may replace $\mathbb{R}$ with $\mathbb{Q}$ without changing the relevant meaning of the semantics.

Definition 3.3 (Buffers)**.** The heap component of a store contains buffers of arbitrary dimensionality:

$$\mathsf{Buf} = \bigcup_{m=0}^{\infty} \left(\mathbb{Z}^m \rightharpoonup (\mathbb{R} \uplus \{\bot_{\mathbb{R}}, \epsilon_m\})\right).$$

We treat these partial functions as having default value ϵ_m. Thus, reading a coordinate outside the allocated region of a buffer produces a memory error, while reading an allocated but uninitialized coordinate produces $\bot_{\mathbb{R}}$.

Definition 3.4 (Windows)**.** An n-dimensional window identifies an underlying buffer together with an indexing transformation:

$$\mathsf{Win}_n = \mathsf{Name}_{data} \times \bigcup_{m=n}^{\infty} \left(\mathbb{Z}^n \to \mathbb{Z}^m\right).$$

A window $(\ell, \phi) \in \mathsf{Win}_n$ refers to the buffer stored at address $\ell \in \mathsf{Name}_{data}$. Its indexing function ϕ is an injective translation: every output coordinate has the form

$$\phi_i(\hat{\imath}) = \hat{\imath}_j + c \qquad \text{or} \qquad \phi_i(\hat{\imath}) = c,$$

and no two output coordinates depend on the same input coordinate. Consequently, reading the window at coordinates $\hat{\imath} \in \mathbb{Z}^n$ reads the underlying buffer at $\phi(\hat{\imath})$.

Definition 3.5 (Values)**.** The set of control values is

$$\mathsf{Val}_c = \mathbb{B} \uplus \mathbb{Z} \uplus \{\bot_{\mathbb{B}}, \bot_{\mathbb{Z}}\}.$$

Argument values additionally include windows:

$$\mathsf{Val}_a = \mathsf{Val}_c \uplus \bigcup_{n=0}^{\infty} \mathsf{Win}_n.$$

Finally, the set of all values is

$$\mathsf{Val} = \mathsf{Val}_a \uplus \mathbb{R} \uplus \{\bot_{\mathbb{R}}, \epsilon_\tau, \epsilon_m\}.$$

Extending the information order to these values yields a meet-semilattice, whose meet is written $\sqcap$.

Definition 3.6 (Functions on values)**.** Let

$$f : D_1 \times \cdots \times D_k \to \mathsf{Val}, \qquad D_i \subseteq \mathsf{Val},$$

be a function defined on values of the appropriate types. Its extension to all values, also written f, agrees with the original function on $D_1 \times \cdots \times D_k$ and otherwise returns

$$x_1 \sqcap \cdots \sqcap x_k.$$

Thus, exceptional values propagate according to the information order, while applying an operation to ordinary values of inappropriate types produces ϵ_τ. These extended functions are monotone with respect to $\sqsubseteq$.

Definition 3.7 (Stores)**.** A *store*, or program state, is either an error or a tuple containing the data, local, and global components:

$$\Sigma = \{\epsilon_\tau, \epsilon_m\} \uplus (\Sigma_{data} \times \Sigma_{local} \times \Sigma_{global}),$$

where

$$\begin{aligned} \Sigma_{data} &= \mathsf{Name}_{data} \rightharpoonup \mathsf{Buf}, \\ \Sigma_{local} &= \mathsf{Name}_{local} \rightharpoonup \mathsf{Val}_a, \\ \Sigma_{global} &= \mathsf{Name}_{global} \rightharpoonup \mathsf{Val}_c. \end{aligned}$$

Undefined names in these partial functions have default value ϵ_τ. We extend the information order pointwise to non-error stores and write $\sigma(x)$ for the appropriate component of σ when that component is clear from context.

Definition 3.8 (Heap and stack)**.** A procedure call receives the nonlocal portion of its caller's store. For a non-error store $\sigma = (\sigma_{data}, \sigma_{local}, \sigma_{global})$, define

$$\mathsf{heap}(\sigma) = (\sigma_{data}, \emptyset, \sigma_{global}).$$

When a callee returns a store $\sigma' = (\sigma'_{data}, \sigma'_{local}, \sigma'_{global})$, its heap and global components are copied back while the caller's local component is retained:

$$\sigma[\mathsf{heap} \mapsto \sigma'] = (\sigma'_{data}, \sigma_{local}, \sigma'_{global}).$$

Definition 3.9 (Functions on stores)**.** Any function defined on non-error stores,

$$f : \Sigma \setminus \{\epsilon_\tau, \epsilon_m\} \to \Sigma,$$

is lifted to all of Σ by propagating ϵ_τ and ϵ_m. Hence, once an execution has entered an error state, all subsequent computation remains in that state. The resulting store functions are monotone with respect to the information order.

3.2.2 Syntax, Semantics, and Well-Typed Programs

The syntax and denotational semantics of the formal core of Exo are given in Figures 3.3, 3.4, 3.5, and 3.6. The denotation of a statement or procedure s is written $\mathrm{S}[\![s]\!]$ and is a store-transforming function

$$\mathrm{S}[\![s]\!] : \Sigma \to \Sigma.$$

The expression semantics use the extensions of value-level operations from Definition 3.6, while the statement and procedure semantics use the error-propagating extension from Definition 3.9.

This core language makes no reference to user-defined instructions or memories. Those features affect code generation but are deliberately opaque to the core program analysis. This separation allows the same analysis to be used with new hardware backends.

Our focus is not on basic type-checking, which is standard, nor on bounds- and assertion-checking, which can be implemented using established techniques (Section 3.1.1). It is nevertheless useful to restate the guarantees provided by these front-end checks. First,

τ_a : ArgType	::= `bool` | `int` | `R[`e^*`]`	
τ_s : SigType	::= `(`x `:` τ_a`)` $\to$ τ_s | `unit`	
τ : Type	::= τ_a | `R`	
note: we use $\cdot^*$ *to mean 0 or more*		
e : Expr	::= x	variables
	| op`(`e^*`)`	built-in operations
	| e`[`e^*`]`	array read
	| `win(`e, w^*`)`	window expression
w : WinCoord	::= e	point-access
	| e `..` e	interval-access
$op \in$	`+, -, *, /, mod, and, or, not, ==, <, <=, >, >=` $\cup$ *Literals*	
s : Stmt	::= s`;`s	sequencing
	| `if` e `then` s	guards
	| `for` x `in` e `..` e `do` s	sequential loops
	| `alloc` x`(`e^*`)`	array allocation
	| e`[`e^*`]=` e	array write
	| e`[`e^*`]+=` e	array reduce
	| x `=` e	global write
	| p`(`e^*`)`	subprocedure call
$pdef$: Proc	::= `proc` p `:` τ_s `assert` e `do` s	
L : Lib	::= `globals` $(x : \tau)^*$ $pdef^*$	

Figure 3.3: Abstract syntax of the Exo core language.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathrm{E} : \mathsf{Expr} \to \Sigma \to \mathsf{Val}} \\
\mathrm{E}\,[\![x]\!]\,\sigma & = & \sigma(x) \\
\mathrm{E}\,[\![op(e_1,\ldots,e_k)]\!]\,\sigma & = & \widehat{op}(\mathrm{E}\,[\![e_1]\!]\,\sigma,\ldots,\mathrm{E}\,[\![e_k]\!]\,\sigma) \\
\mathrm{E}\,[\![e_0[e_1,\ldots,e_n]]\!]\,\sigma & = & \sigma(\ell,\varphi(e'_1,\ldots,e'_n)) \\
\mathrm{E}\,[\![\mathtt{win}(e_0,w_1,\ldots,w_m)]\!]\,\sigma & = & (\ell,\varphi\circ\phi_{win}(w'_1,\ldots,w'_m)) \\
\mathrm{E}\,[\![e_{lo}\,..\,e_{hi}]\!]\,\sigma & = & (\mathrm{E}\,[\![e_{lo}]\!]\,\sigma,\mathrm{E}\,[\![e_{hi}]\!]\,\sigma) \\
\text{where} & & \ell,\varphi = \mathrm{E}\,[\![e_0]\!]\,\sigma \\
e'_i = \mathrm{E}\,[\![e_i]\!]\,\sigma & & w'_i = \mathrm{E}\,[\![w_i]\!]\,\sigma \\
\hline
\phi_{win}() & = & () \\
\phi_{win}(i,u_2,\ldots) & = & p(i)\circ\phi_{win}(u_2,\ldots) \\
\phi_{win}((i_{lo},i_{hi}),u_2,\ldots) & = & \lambda x.cons(x+i_{lo})\circ\phi_{win}(u_2,\ldots) \\
cons(i,(y_1,\ldots,y_n)) & = & (i,y_1,\ldots,y_n)
\end{array}
$$

Figure 3.4: Expression denotations.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathrm{S} : \mathsf{Stmt} \to \Sigma \to \Sigma} \\
\mathrm{S}\,[\![s_1\,\texttt{;}\,s_2]\!]\,\sigma & = & (\mathrm{S}\,[\![s_2]\!] \circ \mathrm{S}\,[\![s_1]\!])\,\sigma \\
\mathrm{S}\,[\![\texttt{if}\ e\ \texttt{then}\ s]\!]\,\sigma & = & \begin{cases} \mathrm{S}\,[\![s]\!]\,\sigma, & \text{if } \ \mathrm{E}\,[\![e]\!]\,\sigma = \mathsf{true} \\ \sigma, & \text{otherwise} \end{cases} \\
\mathrm{S}\,[\![\texttt{for}\ x\ \texttt{in}\ e_{lo}\,..\,e_{hi}\ \texttt{do}\ s]\!]\,\sigma & = & \phi_{m-1} \circ \cdots \circ \phi_n \\
\multicolumn{3}{r}{\text{where } n, m = \mathrm{E}\,[\![e_{lo}]\!]\,\sigma, \mathrm{E}\,[\![e_{hi}]\!]\,\sigma} \\
\multicolumn{3}{r}{\phi_k = \lambda\sigma'.\mathrm{S}\,[\![s]\!]\,(\sigma'[i \mapsto k])} \\
\mathrm{S}\,[\![\texttt{alloc}\ \ x\texttt{(}e_1, \ldots, e_k\texttt{)}]\!]\,\sigma & = & \sigma \begin{bmatrix} \ell \mapsto \mathsf{buf}(\vec{e'}) \\ x \mapsto (\ell, id) \end{bmatrix} \\
\multicolumn{3}{r}{\text{where } \ell \text{ is a fresh name}} \\
\multicolumn{3}{r}{e'_i = \mathrm{E}\,[\![e_i]\!]\,\sigma} \\
\mathrm{S}\,[\![x\ \texttt{=}\ e]\!]\,\sigma & = & \sigma[x \mapsto \mathrm{E}\,[\![e]\!]\,\sigma] \\
\mathrm{S}\,[\![e_0\texttt{[}e_1, \ldots, e_k\texttt{]=}\ e_{rhs}]\!]\,\sigma & = & \sigma\,[(\ell, \hat{\imath}) \mapsto e'_{rhs}] \\
\mathrm{S}\,[\![e_0\texttt{[}e_1, \ldots, e_k\texttt{]+=}\ e_{rhs}]\!]\,\sigma & = & \sigma\,[(\ell, \hat{\imath}) \oplus\!\!\to e'_{rhs}] \\
\multicolumn{3}{r}{\text{where } \ell, \varphi = \mathrm{E}\,[\![e_0]\!]\,\sigma} \\
\multicolumn{3}{r}{\hat{\imath} = \varphi(e'_1, \ldots, e'_k)} \\
\mathrm{S}\,[\![p\texttt{(}e_1, \ldots, e_n\texttt{)}]\!]\,\sigma & = & \sigma[\mathsf{heap} \mapsto \sigma'] \\
\multicolumn{3}{r}{\text{where } \sigma' = \mathsf{call}(p, \vec{e'}, \sigma)}
\end{array}
$$

for $\texttt{proc}\ p : (x_1 : \tau_1) \to \cdots (x_n : \tau_n) \to ()\ \texttt{assert}\ e\ \texttt{do}\ s$,

$$
\begin{array}{rcl}
\mathsf{arg}_i(p) & = & x_i \\
\mathsf{call}(p, \vec{e'}, \sigma) & = & \mathrm{P}\,[\![p]\!]\,(\mathsf{heap}(\sigma)[\mathsf{arg}_i(p) \mapsto e'_i]) \\
\mathsf{buf}(\vec{e'}) & = & [\vec{u} \mapsto \bot \mid 0 \le u_i < e'_i]
\end{array}
$$

Figure 3.5: Statement denotations.

$$
\mathrm{P} : \mathsf{Proc} \to \Sigma \to \Sigma
$$

$$
\mathrm{P}\left[\!\!\left[\begin{array}{l} \texttt{proc}\ \ p : \tau_s \\ \quad \texttt{assert}\ e \\ \quad \texttt{do}\ s \end{array}\right]\!\!\right]\sigma = \begin{cases} \epsilon_m, & \text{if } \ \mathrm{E}\,[\![e]\!]\,\sigma \neq \mathsf{true} \\ \mathrm{S}\,[\![s]\!]\,\sigma, & \text{otherwise} \end{cases}
$$

Figure 3.6: Procedure denotations.

every integer-valued control expression is restricted to be quasi-affine. Second, every access to, or windowing of, a buffer or window is statically guaranteed to be in-bounds. Third, every procedure call is guaranteed to satisfy the asserted preconditions of its callee. Finally, mutation of nonglobal control values is prohibited.

Consequently, a top-level procedure invoked with arguments satisfying its assertions cannot produce a type or memory error. In particular, the quasi-affine restriction lets us translate control expressions into satisfiability queries modulo linear integer arithmetic (LIA), which can then be discharged by an SMT solver.

3.3 Scheduling by Program Rewriting

Rather than directly writing code for a hardware target, Exo users start from a simple, obviously correct program and transform it into an equivalent but more complex, high-performance version targeted at a specific hardware accelerator. This transformation is carried out by successively rewriting the application—a process called *scheduling*.

This section provides a high-level overview of scheduling; a complete list of Exo's scheduling primitives, together with the code transformation and safety condition of each, appears in Section 4.2.

3.3.1 Program Equivalence

The basic correctness contract for Exo scheduling is semantic equivalence. Each scheduling operator takes a procedure and returns a rewritten procedure whose behavior is equivalent to the original, subject to the side conditions of that operator.

Definition 3.10 (program equivalence)**.** Let s_1 and s_2 both be either statements, written Stmt, or procedures, written Proc. We say that s_1 and s_2 are equivalent, written $s_1 \cong s_2$, when the store-transforming functions they denote are equal on valid input stores:

$$\mathrm{S}\,[\![s_1]\!] = \mathrm{S}\,[\![s_2]\!]\,.$$

Here, a valid input store is one that is not already in an error state and that satisfies the relevant precondition assertions. When s_1 and s_2 are procedures, their preconditions must also be equivalent.

Definition 3.10 is, however, stricter than we can afford in practice. As will be discussed later (Section 4.1), many useful transformations—notably those that manipulate configuration state, such as `bind_config()`—preserve everything the program actually computes while leaving behind writes that pollute global configuration variables. We therefore want to reason about programs that are equivalent "up to," or excluding, a set of globals $\mathcal{L}$, and so we define a family of weaker equivalence relations:

Definition 3.11 (program equivalence modulo globals)**.** Let s_1 and s_2 both be either Stmt or Proc, and let $\mathcal{L} \subseteq \mathsf{Name}_{global}$ be a set of global names to ignore. We say that s_1 and s_2 are equivalent modulo $\mathcal{L}$, written $s_1 \cong_{\mathcal{L}} s_2$, when for every valid input store σ and every global $x \notin \mathcal{L}$,

$$\mathrm{S}\,[\![s_1]\!]\,\sigma\ x = \mathrm{S}\,[\![s_2]\!]\,\sigma\ x.$$

Thus, the two programs may differ only in the ignored globals.

Since $\mathcal{L} \subseteq \mathcal{L}'$ implies that $\cong_{\mathcal{L}}$ refines $\cong_{\mathcal{L}'}$, these relations form a lattice of progressively weaker equivalences, with $\cong = \cong_{\emptyset}$ at the top. Exo tracks where each scheduled procedure sits in this lattice, which is what allows configuration-polluting rewrites to be composed safely with the rest of the scheduling language.

3.3.2 Scheduling Operations

Because Exo is an embedded DSL, schedules are written as metaprograms in the host language (Python). Each scheduling operator takes a procedure `p` (e.g., `gemm`) plus some other arguments as input and returns a functionally equivalent, rewritten procedure as output. Operators fall into two categories: primitive operators, the built-ins provided by Exo (Section 4.2), and user-defined operators, new optimizations implemented by users (Chapter 5).

For example, the `divide_loop(p, loop, factor, new_vars, ...)` scheduling primitive divides a single loop of n iterations into a pair of outer and inner loops of n/factor and factor iterations. It takes the procedure, the loop to be divided, the division factor, new iterator variable names, and optional "tail strategy" arguments for handling cases where n does not divide evenly by factor, and it returns the modified procedure.

This raises a natural question: how can we specify which loop we want to divide? As we will explore in Chapter 5, the problem of *reference* is a fundamental issue in scheduling languages. To get us off the ground, we will introduce two mechanisms for identifying object code. First, we can refer to parts of the object code by *name*. For example, we can divide the `'i'` loop of `gemm` simply by naming it: `divide_loop(gemm, 'i', ...)`. Second, we can refer to more complex or specific pieces of code structurally using more general *patterns*. For example, we could refer to that same `'i'` loop in `gemm` with the pattern `'for i in _: _'`. This pattern matches a loop with the iteration variable `i`. The wildcards (`_`) match any range specification and any loop body. Finally, we reify these references in objects we call *Cursors*, drawing an analogy to cursors in text editors. Procedures expose `find_loop` and `find` methods that take loop names or patterns and return matching cursors:

```
cur_0 = gemm.find_loop('i')          # loop name
cur_1 = gemm.find('for i in _:_')    # pattern
assert(cur_0 == cur_1)               # point to the same loop
```

Most scheduling operators take cursors for any reference arguments, though they often also accept pattern strings as an optional convenience. For example, `divide_loop(p, 'i', ...)` is shorthand for `divide_loop(p, p.find_loop('i'), ...)`.

Every scheduling operator has the type $\mathsf{Op} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc}$, taking a procedure, a cursor, and other arguments and returning a procedure. This design contrasts with the more common method-chaining style used in Halide [43] and TVM [44], where scheduling operators are methods on a program object. Section 4.2 lists Exo's full set of scheduling primitives, which are transformations built into the compiler, and Chapter 5 unpacks Cursors in greater detail.

3.3.3 Rewrite-Based Scheduling

The scheduling flow can be summarized as follows:

$$p_1 \xrightarrow{R_1} p_2 \xrightarrow{R_2} \cdots \xrightarrow{R_{N-1}} p_N.$$

For ordinary Exo rewrites, each step preserves sequential behavior; that is, consecutive procedures are equivalent in the sense of Definition 3.10:

$$p_1 \cong p_2 \cong \cdots \cong p_N,$$

or, unfolding the definition,

$$\mathrm{S}\,[\![p_1]\!] = \mathrm{S}\,[\![p_2]\!] = \cdots = \mathrm{S}\,[\![p_N]\!]\,.$$

Here, $\mathrm{S}\,[\![p]\!]$ denotes the store-transforming function as in Definition 3.10. Chapter 7 inherits this sequential rewrite scheme unchanged for GPU programs. It then separately checks, in the backend before code generation, that the final program's parallel interpretation agrees with its sequential one.

Like Lift and ELEVATE [45,46], but unlike Halide and TVM, Exo's scheduling operators are rewrites of programs rather than arguments to a monolithic lowering process. Consequently, the implementation and correctness of each scheduling primitive is independent of every other primitive, which makes the Exo implementation substantially simpler and easier to maintain. Importantly, Exo rewrites imperative rather than functional programs. This makes checking the correctness of the primitive rewrites more complex, as we'll see in the next section (Section 3.4), but it is what allows the technique to target the imperative kernels that hardware accelerators expect.

3.4 Effect Analysis & Transformation of Programs

Our analysis of Exo programs is based on an *effect* analysis. An effect a extracted from a statement s characterizes which functions $f : \Sigma \to \Sigma$ the statement s could possibly denote $\mathrm{S}\,[\![s]\!]$. This effect analysis allows us to determine when code transformations like $s_1\,;s_2 \rightsquigarrow s_2\,;s_1$ and $s_1\,;s_2 \rightsquigarrow s_2$ are valid.

This analysis will require us to define (1) effect expressions and environments, (2) a global symbolic data-flow analysis, (3) location sets as a symbolic abstraction of store locations, and finally (4) effects as an abstraction of programs. We can then state safety conditions for various program rewrites using these building blocks.

3.4.1 Ternary Logic

When extended with $\bot$, $\mathbb{B}$ becomes a ternary logic with the values true (true or T), false (false or F), and unknown ($\bot$). Intuitively, this ternary logic will allow us to distinguish between statements that are *definitely* true and statements that *may be* true. This logic can be encoded in classical logic for the purposes of targeting SMT solvers, as we detail at the end of this subsection.

We define two additional operators for collapsing back down from ternary to classical logic. $D\ p$ ("definitely p") is defined by $DT = T$, $D\bot = F$, and $DF = F$; $M\ p$ ("maybe p") is defined by $MT = T$, $M\bot = T$, and $MF = F$.

Encoding ternary logic We may encode effect expressions (Definition 3.12), whose operators are

$$op \in \{+, -, *, /, \mathsf{mod}, \wedge, \vee, \neg, =, <, >, \leq, \geq\},$$

for standard SMT solvers as follows. Represent a value in $\mathbb{Z} \cup \{\bot\}$ by a value in $\mathbb{Z} \times \mathbb{B}$ and a value in $\mathbb{B} \cup \{\bot\}$ by a value in $\mathbb{B} \times \mathbb{B}$, such that the meaning of the encoding is defined by $\mathsf{Val}(x, \mathsf{true}) = x$ and $\mathsf{Val}(x, \mathsf{false}) = \bot$. Accordingly, we can pull back the definitions of the various operators as follows. For $op \in \{+, -, *, /, \mathsf{mod}, =, <, >, \leq, \geq\}$ with inputs of int sort, the rule is simple and illustrated by $+$:

$$(x_1, d_1) + (x_2, d_2) \mapsto (x_1 + x_2, d_1 \wedge d_2)$$

For operators on inputs of bool sort, we may take advantage of Boolean short-circuiting to produce known values even when some input is unknown. Such definitions are key to extracting useful information from the ternary logic. These operators are

$$\begin{aligned}
(x_1, d_1) \wedge (x_2, d_2) &\mapsto \left(\begin{array}{ll} & (d_1 \wedge d_2) \vee \\ x_1 \wedge x_2, & (\neg x_1 \wedge d_1) \vee \\ & (\neg x_2 \wedge d_2) \end{array}\right) \\
(x_1, d_1) \vee (x_2, d_2) &\mapsto \left(\begin{array}{ll} & (d_1 \wedge d_2) \vee \\ x_1 \vee x_2, & (x_1 \wedge d_1) \vee \\ & (x_2 \wedge d_2) \end{array}\right) \\
\neg(x, d) &\mapsto (\neg x, d) \\
\forall x_1.(x_2, d_2) &\mapsto \left(\begin{array}{ll} (\forall x_1.x_2), & (\forall x_1.d_2) \vee \\ & (\exists x_1.(\neg x_2 \wedge d_2)) \end{array}\right) \\
(x_1, d_1)?\ (x_2, d_2)\ \mathsf{else}\ (x_3, d_3) &\mapsto \\
& (x_1?\ x_2\ \mathsf{else}\ x_3, d_1 \wedge (x_1?\ d_2\ \mathsf{else}\ d_3))
\end{aligned}$$

3.4.2 Effect Expressions

Effect expressions both give us a way of expressing symbolic values and of encoding sentences in a first-order logic, for discharging to an SMT solver.

Definition 3.12 (Effect Expressions)**.** We define the following grammar of effect-expressions

$$ee : \mathsf{EffExpr} ::= x \mid c \mid \bot \mid op(ee^*) \mid ee?\ ee\ \mathsf{else}\ ee \mid \forall x.ee$$

where every expression either has sort bool or sort int. The operators are the same as the bool and int operators from Figure 3.3. Recall that in the case of int operators, the pseudo-affine condition means that the quotient for $/$ and mod must be a constant, and one side of $*$ must be a constant.

Definition 3.13 (Effect Environments)**.**

$$\gamma : \mathsf{EffEnv} = (\mathsf{Name}_{global} \uplus \mathsf{Name}_{local}) \to \mathsf{EffExpr}$$

are partial functions that default to mapping x to x, not $\bot$.

Effect environments abstract functions $\Sigma \to \Sigma$ with respect to control values, not stores Σ. This is why they may appear to be impredicative (mapping x to x by default). We define substitution $\gamma(ee)$ in the usual way. Using this, we can define composition of two effect environments $(\gamma \cdot \gamma')x = \gamma(\gamma'(x))$, which may also be resolved by substituting with γ inside the expressions bound by γ'. This definition of substitution extends naturally up to our later definitions of location sets LocSet and effects a.

3.4.3 Global Dataflow

The main complication in our analyses is mutable, global control state, which makes precise analysis of control logic undecidable in general. Our dataflow analysis is therefore symbolic, yielding effect environments, and control-sensitive, so that symbolic values reflect the guards enclosing a statement. We write $\mathsf{ValG}[\![s]\!]$ for the effect environment (Definition 3.13) that the global analysis assigns to a statement s: it records how s updates symbolic control values and is applied to the effects of subsequent statements (Definition 3.17).

However, we must make some kind of approximation to force convergence on loops. We use a very simple heuristic, expressed symbolically: If every loop iteration does not change the value of a global variable x, then the loop behaves as an identity function. Otherwise, the loop drives x to the uncertain value $\bot$. This usually suffices because configuration state that depends on the loop iteration is usually meaningless outside of the loop. When it is *not*—for example, RISC-V's vector-length register `rvv.vl`, whose loop-varying value must be tracked to schedule vector-length-agnostic code safely—this coarse heuristic is insufficient, and a value-sensitive analysis is required; we return to this case in Chapter 6.

3.4.4 Location Sets

Definition 3.14 (Location Set)**.**

$$\mathcal{L} : \mathsf{LocSet} ::= \emptyset \mid \{x, ee^*\} \mid \mathcal{L} \cup \mathcal{L} \mid \bigcup_x \mathcal{L} \mid \mathcal{L} \cap \mathcal{L} \mid \mathcal{L} - \mathcal{L} \mid \mathsf{filter}(ee, \mathcal{L})$$

Location sets symbolically abstract sets of global and heap locations in the store.

These sets support a set-membership predicate $(\_ \in \_) : (\mathsf{Name} \times \mathsf{EffExpr}^n) \to \mathsf{LocSet} \to \mathsf{EffExpr}$ and an is-empty predicate $(\_ = \emptyset) : \mathsf{LocSet} \to \mathsf{EffExpr}$, both defined in the expected way as follows:

$$
\begin{aligned}
\in &: \mathsf{Name} \times \mathsf{EffExpr}^n \to \mathsf{LocSet} \to \mathsf{EffExpr} \\
xs \in \emptyset &= \mathsf{false} \\
xs \in \{y, ee_1, \ldots, ee_n\} &= x = y \wedge \textstyle\bigwedge_{i=0}^{n} ee'_i = ee_i \\
&\quad \text{where } xs = (x, ee'_1, \ldots, ee'_n) \\
xs \in \mathcal{L}_1 \cup \mathcal{L}_2 &= (xs \in \mathcal{L}_1) \vee (xs \in \mathcal{L}_2) \\
xs \in \textstyle\bigcup_x \mathcal{L} &= \exists_x (xs \in \mathcal{L}) \\
xs \in \mathcal{L}_1 \cap \mathcal{L}_2 &= (xs \in \mathcal{L}_1) \wedge (xs \in \mathcal{L}_2) \\
xs \in \mathcal{L}_1 - \mathcal{L}_2 &= (xs \in \mathcal{L}_1) \wedge (xs \notin \mathcal{L}_2) \\
xs \in \mathsf{filter}(ee, \mathcal{L}) &= ee \wedge (xs \in \mathcal{L})
\end{aligned}
$$

$$
\begin{aligned}
(\_ = \emptyset) &: \mathsf{LocSet} \to \mathsf{EffExpr} \\
(\mathcal{L} = \emptyset) &= \forall xs. (xs \notin \mathcal{L})
\end{aligned}
$$

Note that because effect expressions are a ternary logic, these location sets express upper and lower bounds on a set of locations: points definitely not in the set, points definitely in the set, and a penumbra of points ambiguously in the set. We collapse these sets down to "classical sets" using the aforementioned operators: $D\mathcal{L}$ meaning points definitely in the set, and $M\mathcal{L}$ meaning points that might be in the set. Thus $x \in D\mathcal{L}$ means $D(x \in Ls)$ and $x \notin M\mathcal{L}$ means $D(x \notin \mathcal{L})$.

3.4.5 Effects

Definition 3.15 (Effects)**.**

$$
\begin{aligned}
a : \mathsf{Effect} \quad ::= \quad & a\,;a \mid \emptyset \mid \mathsf{Guard}(ee, a) \mid \mathsf{Loop}(x, a) \\
\mid \quad & \mathsf{GlobalRead}(x) \mid \mathsf{GlobalWrite}(x) \\
\mid \quad & \mathsf{Read}(x, ee^*) \mid \mathsf{Write}(x, ee^*) \\
\mid \quad & \mathsf{Reduce}(x, ee^*) \mid \mathsf{Alloc}(x)
\end{aligned}
$$

This definition allows us to define the obvious translation of expressions ($\mathit{Eff}_e : \mathsf{Expr} \to \mathsf{Effect}$) and statements ($\mathit{Eff} : \mathsf{Stmt} \to \mathsf{Effect}$) into effects, as defined below.

Definition 3.16 (Effect of an Expression)**.** In the following, we sometimes combine effects using $\cup$ in place of ; to indicate that since we are strictly combining read-effects, the order of composition is irrelevant.

$$
\begin{aligned}
\mathit{Eff}_e [\![x]\!] &= \begin{cases} \mathsf{GlobalRead}(x), & \text{if } x \in \mathsf{Name}_{glob} \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathit{Eff}_e [\![op(e_1, \ldots, e_n)]\!] &= \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e [\![e_i]\!] \\
\mathit{Eff}_e [\![e_0[e_1, \ldots, e_n]]\!] &= \left[\begin{array}{l} \mathsf{let}\ x, \varphi = \mathit{Lift}\ [\![e_0]\!] \\ \qquad ee_i = \mathit{Lift}\ [\![e_i]\!] \\ \qquad a' = \bigcup_{i=0}^{n} \mathit{Eff}_e [\![e_i]\!] \\ \qquad \varphi_{ee} = \varphi(ee_1, \ldots, ee_n) \\ \qquad\quad \mathsf{in}\ a' \cup \mathit{Read}(x, \varphi_{ee}) \end{array}\right. \\
\mathit{Eff}_e [\![\mathtt{win}(e_0, w_1, \ldots, w_n)]\!] &= \mathit{Eff}_e [\![e_0]\!] \cup \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e [\![w_i]\!]
\end{aligned}
$$

Definition 3.17 (Effect of a Statement)**.**

$$
\begin{aligned}
&\mathit{Eff}\ [\![s_1\,;s_2]\!] && = \mathit{Eff}\ [\![s_1]\!]\ ;\mathbf{ValG}\ [\![s_1]\!]\ (\mathit{Eff}\ [\![s_2]\!])\\
&\mathit{Eff}\ [\![\texttt{if}\ e\ \texttt{then}\ s]\!] && = \mathit{Eff}_e\ [\![e]\!]\ ;\mathsf{Guard}(\mathit{Lift}\ [\![e]\!], \mathit{Eff}\ [\![s]\!])\\
&\mathit{Eff}\ [\![\texttt{for}\ x\ \texttt{in}\ e_{lo}\texttt{..}e_{hi}s\ \texttt{do}\ ]\!] && = \left[\begin{array}{l}(\mathit{Eff}_e\ [\![e_{lo}]\!] \cup \mathit{Eff}_e\ [\![e_{hi}]\!])\,;\\ \mathsf{Loop}(x, \mathsf{Guard}(bds, G(\mathit{Eff}\ [\![s]\!])))\end{array}\right.\\
& && \quad \text{where } bds = \mathit{Lift}\ [\![e_{lo}]\!] \le x < \mathit{Lift}\ [\![e_{hi}]\!]\\
& && \quad G = \mathbf{ValG}\ [\![\texttt{for}\ x\ \texttt{in}\ e_{lo}\texttt{..}e_{hi}s\ \texttt{do}\ ]\!]\\
&\mathit{Eff}\ [\![\texttt{alloc}\ \ x(e_1,\ldots,e_n)]\!] && = \textstyle\bigcup_{i=1}^{n} \mathit{Eff}_e\ [\![e_i]\!]\\
&\mathit{Eff}\ [\![e_0\texttt{[}e_1,\ldots,e_n\texttt{]=}\ e_r]\!] && = \left[\begin{array}{l}\mathsf{let}\ x, \varphi = \mathit{Lift}\ [\![e_0]\!]\\ \qquad ee_i = \mathit{Lift}\ [\![e_i]\!]\\ \qquad\quad \mathsf{in}\ (\mathit{Eff}_e\ [\![e_r]\!] \cup \bigcup_{i=0}^{n} \mathit{Eff}_e\ [\![e_i]\!])\ ;\\ \qquad\qquad \mathsf{Write}(x, \varphi(ee_1,\ldots,ee_n))\end{array}\right.\\
&\mathit{Eff}\ [\![e_0\texttt{[}e_1,\ldots,e_n\texttt{]+=}\ e_r]\!] && = \left[\begin{array}{l}\mathsf{let}\ x, \varphi = \mathit{Lift}\ [\![e_0]\!]\\ \qquad ee_i = \mathit{Lift}\ [\![e_i]\!]\\ \qquad\quad \mathsf{in}\ (\mathit{Eff}_e\ [\![e_r]\!] \cup \bigcup_{i=0}^{n} \mathit{Eff}_e\ [\![e_i]\!])\ ;\\ \qquad\qquad \mathsf{Reduce}(x, \varphi(ee_1,\ldots,ee_n))\end{array}\right.\\
&\mathit{Eff}\ [\![x\ \texttt{=}\ e]\!] && = \mathit{Eff}_e\ [\![e]\!]\ ;\mathsf{GlobalWrite}(x)
\end{aligned}
$$

Effects then allow us to define read, write, and reduce location sets.

To start, we define the set of buffers allocated by and visible to subsequent statements/effects:

$$
\begin{aligned}
&\mathbf{A} : \mathsf{Effect} \to \mathsf{LocSet}\\
&\mathbf{A}\ \mathsf{Alloc}(x) = \{x\}\\
&\mathbf{A}\ (a_1\,;a_2) = \mathbf{A}(a_1) \cup \mathbf{A}(a_2)\\
&\mathbf{A}\ \_ = \emptyset
\end{aligned}
$$

Definition 3.18 (Locations of an Effect)**.** Let $\mathbf{Rd_G}$, $\mathbf{Wr_G}$, $\mathbf{Rd}$, $\mathbf{Wr}$, and $\mathbf{R}+$, be functions $\mathsf{Effect} \to \mathsf{LocSet}$. To avoid redundancy, define common cases for all such functions $\mathcal{F}$:

$$
\begin{aligned}
&\mathcal{F} : \mathsf{Effect} \to \mathsf{LocSet}\\
&\mathcal{F}\ \mathsf{Guard}(ee, a) = \mathsf{filter}(ee, \mathcal{F}\ a)\\
&\mathcal{F}\ \mathsf{Loop}(x, a) = \textstyle\bigcup_x \mathcal{F}\ a'
\end{aligned}
$$

Sequencing is defined differently for read and write sets:

$$
\begin{aligned}
&\mathbf{Rd_G}\ (a_1\,;a_2) = \mathbf{Rd_G}(a_1) \cup (\mathbf{Rd_G}(a_2') - \mathbf{Wr_G}(a_1) - \mathbf{A}(a_1))\\
&\mathbf{Wr_G}\ (a_1\,;a_2) = \mathbf{Wr_G}(a_1) \cup (\mathbf{Wr_G}(a_2') - \mathbf{A}(a_1))\\
&\mathbf{Rd}\ (a_1\,;a_2) = \mathbf{Rd}(a_1) \cup (\mathbf{Rd}(a_2') - \mathbf{Wr}(a_1) - \mathbf{A}(a_1))\\
&\mathbf{Wr}\ (a_1\,;a_2) = \mathbf{Wr}(a_1) \cup (\mathbf{Wr}(a_2') - \mathbf{A}(a_1))\\
&\mathbf{R}+\ (a_1\,;a_2) = \mathbf{R}+(a_1) \cup (\mathbf{R}+(a_2') - \mathbf{A}(a_1))
\end{aligned}
$$

Each function detects its corresponding leaf-node:

$$
\begin{aligned}
&\mathbf{Rd_G}\ \mathsf{GlobalRead}(x) = \{x\}\\
&\mathbf{Wr_G}\ \mathsf{GlobalWrite}(x) = \{x\}\\
&\mathbf{Rd}\ \mathsf{Read}(x, ee_1,\ldots,ee_n) = \{x, ee_1,\ldots,ee_n\}\\
&\mathbf{Wr}\ \mathsf{Write}(x, ee_1,\ldots,ee_n) = \{x, ee_1,\ldots,ee_n\}\\
&\mathbf{R}+\ \mathsf{Reduce}(x, ee_1,\ldots,ee_n) = \{x, ee_1,\ldots,ee_n\}\\
&\mathcal{F}\ \_ = \emptyset
\end{aligned}
$$

From these five primitive sets we can define six other useful sets:

$$\begin{aligned}
\mathbf{Rd}\ a &= \mathbf{Rd_G}\ a \cup \mathbf{Rd}\ a \\
\mathbf{Wr}\ a &= \mathbf{Wr_G}\ a \cup \mathbf{Wr}\ a \\
\mathbf{R+}\ a &= \mathbf{R+}\ a - \mathbf{Wr}\ a \\
\mathbf{All}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a \cup \mathbf{R+}\ a \\
\mathbf{Mod}\ a &= \mathbf{Wr}\ a \cup \mathbf{R+}\ a \\
\mathbf{RW}\ a &= \mathbf{Rd}\ a \cup \mathbf{Wr}\ a
\end{aligned}$$

3.4.6 Effects as Abstraction

The different objects we have talked about so far each abstract some part of the program. For instance, the dataflow analysis of a statement $\mathbf{ValG}\,[\![s]\!]$ is an abstraction of its denotation $\mathrm{S}\,[\![s]\!]$ with respect to global values. Similarly, the effect extracted from an expression $\mathit{Eff}_e\,[\![e]\!]$ abstracts its denotation $\mathrm{E}\,[\![e]\!]$, and the effect extracted from a statement $\mathit{Eff}\,[\![s]\!]$ abstracts its denotation $\mathrm{S}\,[\![s]\!]$. But what do we mean by this?

The effect abstraction a for a statement s with denotation f guarantees a few properties. First, it provides an analogue of the "frame axiom" from separation logic. If a location x lies outside of $M\mathbf{Mod}(a)$, then it is unmodified: $f\sigma x = \sigma x$. Second, if a location is in the write set $x \in D\mathbf{Wr}(a)$, then the post-hoc value at that location $f\sigma x$ is determined solely by the values at read locations $y \in M\mathbf{Rd}(a)$. Third, if a location is reduced to $x \in D\mathbf{R+}(a)$, then the difference between the initial and final value at that location $f\sigma x - \sigma x$ is determined solely by values at read locations $y \in M\mathbf{Rd}(a)$. Finally, so long as the values at read locations $y \in M\mathbf{Rd}(a)$ are determined, then one of the three previous cases applies to every store location, even if we can't be certain which set(s) the location is in.

Even more simply in the case of expression abstraction, the effect a of an expression e with denotation $f : \Sigma \to \mathsf{Val}$ guarantees one property: The value $f\sigma$ is solely determined by the values at read locations $y \in M\mathbf{Rd}(a)$.

3.4.7 Basic Program Rewrites

The preceding analysis objects allow us to turn program equivalence checks into SMT queries.

Reorder statements The rewrite $s_1\,;s_2 \rightsquigarrow s_2\,;s_1$ is safe when $\mathsf{Commutes}\ \mathit{Eff}\,[\![s_1]\!]\ \ \mathit{Eff}\,[\![s_2]\!]$ holds. Commutativity of statements is defined as noninterference of effects. A special exception must be made for locations that are reduced.

Definition 3.19 (Commutativity)**.**

$$\begin{aligned}
&\mathsf{Commutes}\ a_1\ a_2 = \\
&D\left(\begin{array}{l}
\mathbf{Wr}(a_1) \cap \mathbf{All}(a_2) = \emptyset\ \ \wedge\ \ \mathbf{Wr}(a_2) \cap \mathbf{All}(a_1) = \emptyset \\
\mathbf{R+}(a_1) \cap \mathbf{Rd}(a_2) = \emptyset\ \ \wedge\ \ \mathbf{R+}(a_2) \cap \mathbf{Rd}(a_1) = \emptyset
\end{array}\right)
\end{aligned}$$

Shadow statement The rewrite $s_1\,;s_2 \rightsquigarrow s_2$ is safe when $\mathsf{Shadows}\ \mathit{Eff}\,[\![s_1]\!]\ \ \mathit{Eff}\,[\![s_2]\!]$ holds. Whereas commutativity requires reasoning about what definitely doesn't intersect (and hence what memory might be touched), shadowing requires reasoning positively about what definitely is overwritten—which is why a one-sided approximation sets is insufficient.

Definition 3.20 (Shadowing)**.**

$$\begin{array}{l}\mathsf{Shadows}\ a_1\ a_2 = \\ \forall x \in M\mathbf{Mod}(a_1) \implies (x \notin M\mathbf{Rd}(a_2) \ \wedge \ x \in D\mathbf{Wr}(a_2))\end{array}$$

New config write The rewrite $s \rightsquigarrow s;x_g{=}e$ is always safe but only results in code that is equivalent modulo $\{x_g\}$. This rule underlies `bind_config`, `write_config`, and `delete_config` in Section 4.2.6: they preserve equivalence only modulo the affected configuration field. They are distinct from Prexo's `bind` operation in Section 6.6.2, which uses inferred value relations to justify the introduced binding.

3.4.8 Loop Rewrites

When working with rewrites of loops, it is convenient to abbreviate notation for an iteration variable being in-bounds. If the variable x occurs in `for` x `in` e_{lo} `..` e_{hi} `do` , then let $\mathsf{Bd}(x) = \mathit{Lift}\ [\![e_{lo}]\!] \leq x < \mathit{Lift}\ [\![e_{hi}]\!]$ in the following.

Loop reordering One of the most basic nontrivial loop transformations is loop reordering. When can we rewrite `for` x `do for` y `do` s into `for` y `do for` x `do` s? This transformation is valid when the loop bounds commute with the body, and when any loop iterations that are moved past each other commute. To formulate these conditions, let a_x be the effect of the x-loop's bound expressions and a_y similarly for the y-loop. Let x', y' be copies of these iteration variables s.t. $s' = [x \mapsto x'][y \mapsto y']s$. Let $a = \mathit{Eff}\ [\![s]\!]$ and $a' = \mathit{Eff}\ [\![s']\!]$. Then the reordering condition may be precisely stated as

$$\begin{array}{l}(\forall x, y.\ M\mathsf{Bd}(x, y) \implies \mathsf{Commutes}((a_x ; a_y), a)) \\ \wedge \left(\begin{array}{r} \forall x, y, x', y'.\ M(\mathsf{Bd}(x, y, x', y') \wedge x < x' \wedge y' < y) \\ \implies \mathsf{Commutes}(a, a') \end{array} \right)\end{array}$$

Loop fusion & fission Another basic loop transformation is to fuse two loops together or in reverse to fission one loop in two. When can we rewrite `for` x `do` s_1;s_2 into (`for` x `do` s_1);`for` x `do` s_2? This is possible when the loop bound commutes with the first statement and when the statements that get reordered commute with each other. Letting a_x be the effect of the loop bounds, $a_1 = \mathit{Eff}\ [\![s_1]\!]$, $s_2' = [x \mapsto x']s_2$, and $a_2' = \mathit{Eff}\ [\![s_2']\!]$, we can state fission conditions precisely as

$$\begin{array}{l}(\forall x.\ M\mathsf{Bd}(x) \implies \mathsf{Commutes}(a_x, a_1)) \ \wedge \\ (\forall x, x'.\ M(\mathsf{Bd}(x, x') \wedge x' < x) \implies \mathsf{Commutes}(a_1, a_2'))\end{array}$$

Loop removal In order for the rewrite `for` x `do` $s \rightsquigarrow s$ to be safe, the variable x must not be free in s, s must be idempotent, and the loop must run for at least one iteration. If $a = \mathit{Eff}\ [\![s]\!]$, then these conditions are precisely

$$(\exists x.\ D\mathsf{Bd}(x)) \ \wedge \ \mathsf{Shadows}(a, a)$$

Chapter 4

Externalizing Hardware Targets from the Compiler

In this chapter, we unpack one of the two pillars of *Exocompilation*, a new compiler approach for developing hardware-accelerated, high-performance libraries: externalizing as much accelerator-specific code-generation and optimization logic as possible out of the compiler, exposing it instead to high-performance library writers at the user level. In Exo, custom hardware instructions are user-defined and abstracted as procedures, while specialized memories and configuration state are defined in user code, all without modifying the core compiler. We introduce these mechanisms through the running GEMM example in Section 4.1.

Building on the rewrite-based user scheduling introduced in Section 3.3, we now turn to *scheduling primitives* (Section 4.2). Scheduling primitives are compiler built-in transformations. In Exo, they are designed to be as low-level as possible, giving performance engineers the fine-grained control needed to achieve peak performance. Because they are low-level and granular, we provide over forty primitives spanning a wide range of optimization scenarios. These primitives externalize hardware-mapping and optimization decisions to the performance engineer. They are both concise and safe: they elide details such as loop and array re-indexing, which can be inferred automatically, while guaranteeing functional equivalence and memory safety. This safety rests on the foundations established in Section 3.4, where effect analysis guarantees program equivalence and memory safety under scheduling.

We further discuss the `.replace()` primitive and its *unification* procedure in Section 4.2.7. Crucially, unification lets the engineer replace arbitrary program fragments with equivalent user-defined accelerator instructions or specialized subroutines, automatically resolving the argument and array-indexing details that the substitution requires. A single source algorithm can thus be specialized to different hardware, or even different parameter values, simply by applying different schedules.

Finally, we put Exo to work in a series of case studies that optimize high-performance kernels for specialized hardware. We develop user-level backends for the Gemmini neural network accelerator [39], a software-controlled systolic array resembling many TPU-like architectures (details in Section 2.3), and for x86-64 with AVX-512. In each case we focus on matrix multiplication and convolutional neural network layers, which are among the most heavily optimized kernels in common libraries. With Exo, we were able to develop implementations competitive with state-of-the-art libraries in only a few dozen lines of code.

4.1 Hardware Targets as Libraries

Having introduced the Exo object language with the `gemm` running example (Figure 3.2), we now use that same example to show how a kernel is optimized for a sample hardware accelerator whose ISA is a simplified variant of Gemmini's, centered on the efficient execution of small (e.g. 16×16), dense matrix-matrix multiplication instructions.

Optimizing these kernels is primarily an exercise in orchestrating data movement and only secondarily a matter of selecting compute instructions such as the matrix-multiplication primitive: a programmer must explicitly schedule loads and stores between `DRAM` and the accelerator's custom, explicitly managed memories, such as a `SCRATCHPAD` for inputs and an `ACCUMULATOR` for partial sums. Much of an accelerator's behavior is further governed by infrequently changing *configuration state*, and the instructions that update it typically flush the pipeline. These three features—custom memories, instructions, and configuration state—are all a hardware developer must define to model an accelerator. This work is done once per accelerator and packaged as a reusable *hardware library*, rather than repeated per application or baked into a compiler backend as Halide, TVM, and LLVM do. The library approach has two advantages: hardware vendors need not maintain compiler forks to protect proprietary details of their hardware, and the cost of adding support for new hardware is significantly reduced—our experience adding support to both Exo and Halide suggests adding hardware target as a library to Exo requires at least an order of magnitude less development time than modifying Halide's backend.

Given this hardware definition, an Exo programmer uses scheduling to rewrite a simple program into one that targets the accelerator. Throughout the example we tag each code listing to indicate where it lives: a listing marked *in app.py* is application code, the GEMM kernel a user writes and schedules, while a listing marked *in hw_lib.py* is hardware library code, the reusable definitions that abstract the accelerator.

4.1.1 Tiling GEMM to Expose 16×16 Multiplication

We start from the running `gemm` procedure (Figure 3.2), here taken at the concrete instance $M = N = K = 128$. To target the accelerator, we must expose a 16×16 matrix multiplication as the innermost loop nest. We do this entirely through *scheduling primitives* that rewrite the procedure. Recall from Section 3.3.2 that the scheduling operators had type $\mathsf{Op} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc}$: each one takes the procedure being scheduled as its first argument, a cursor identifying the object code to rewrite, and any remaining parameters and returns a new procedure. Accordingly, we `divide_loop(gemm, i, 16, io, ii)` (and similarly for `j` and `k`) to break each 128-iteration loop into an outer loop over 8 tiles and an inner loop of 16, and then `reorder_loops()` (see Section 4.2) so that the three inner indices are nested together, threading the rewritten procedure returned by each operator into the next. The result is the following tiled matrix multiplication:

```
def gemm(A: f32[128, 128] @ DRAM, B: f32[128, 128] @ DRAM, C: f32[128, 128] @ DRAM):
  for io in seq(0, 8):
    for jo in seq(0, 8):
      for ko in seq(0, 8):
        for ii in seq(0, 16):
          for ji in seq(0, 16):
            for ki in seq(0, 16):
              C[16*io+ii, 16*jo+ji] += A[16*io+ii, 16*ko+ki] * B[16*ko+ki, 16*jo+ji]
```

in app.py

4.1.2 Memories

Many accelerators—including ours in this example—have explicitly managed memories. By default, all Exo buffers reside in system `DRAM` and are managed using standard `malloc` and `free`. Accelerators, however, often require modeling buffers that are resident in special accelerator memories or pinned to special address ranges in the global address space. Performance critically depends on how data movement to and from these memories is interleaved with other computation. Therefore Exo puts scheduling of data movement in the hands of the programmer. The first step in doing this is to define *custom memories* on a per-accelerator basis by subclassing a `Memory` base class and overloading its methods. For example,

```
class ACCUMULATOR(Memory):
    def alloc(...):
        return f"{prim_type} {name} = hw_malloc({sz});"
    def free(...):
        return f"hw_free({name});"
    def read(...):  # also write, reduce
        raise MemGenError('memory is not addressable')
```

in hw_lib.py

If a buffer or window type is annotated with `ACCUMULATOR` instead of `DRAM` (e.g., `x : f32[n] @ ACCUMULATOR` says that the vector `x` lives in `ACCUMULATOR`), then these `alloc` and `free` *macros* will determine the `C` code that is generated when that buffer is allocated or freed via string interpolation. With `read(...)`, the author of a custom memory specifies whether to allow standard reading and writing of the buffer (appropriate when the memory merely changes the memory-management policy) or to disable all ordinary accesses. `ACCUMULATOR` and `SCRATCHPAD` memories take the latter option and explicitly disable code generation for reading, writing, and accumulating into individual locations, preventing direct access from `C`. Instead, we will only allow custom instructions (see below) to access this custom memory; improper accesses are prevented by backend checks (Section 3.1.1). In general, memory annotations are ignored during scheduling.

Supposing we have written custom `ACCUMULATOR` and `SCRATCHPAD` memories, we use the `stage_mem` scheduling primitive to stage `C`, `A`, and `B` into these memories:

```
def gemm(...):
    res: R[...] @ ACCUMULATOR
    a  : R[...] @ SCRATCHPAD
    b  : R[...] @ SCRATCHPAD
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Load C to res
            for ko in seq(0, 8):
                # Load A to a
                for ii in seq(0, 16):
                    for ki in seq(0, 16):
                        a[...] = A[...]
                ... # Load B to b
                # Matmul of a and b
                for ii in seq(0, 16):
                    for ji in seq(0, 16):
                        for ki in seq(0, 16):
                            res[..]+=a[..]*b[..]
            ... # Store res to C
```

in app.py

4.1.3 Instructions

We can clearly see opportunities in the above code to map loops to semantically equivalent accelerator instructions. However, to do this soundly, the compiler needs definitions of our accelerator instructions in terms of Exo's semantics. The key idea of exocompilation is to provide users with a framework for defining these instructions in libraries, without modifying the compiler itself. An instruction is simply an Exo procedure annotated with a macro—a C code template—rather than with `@proc`. When code-generating a call to the instruction, Exo emits this template in place of a procedure call, interpolating the arguments into the template's placeholders as strings. This works as well for scheduling fine-grained intrinsics as for coarse-grained calls to existing microkernels or library routines. Below, we show an example of such a definition for the scratchpad load.

```
@instr("config_ld({src}.strides[0]);\n"
       "mvin({src}.data, {dst}.data, {m}, {n});")
def ld_data(n: size, m: size,
            src: [R][n, m] @ DRAM,
            dst: [R][n, 16] @ SCRATCHPAD):
    assert m <= 16
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```

in hw_lib.py

Notice that this function has been annotated with `@instr` rather than `@proc`. This indicates that the declaration *asserts* equivalence between the Exo code in the body and the C code template (i.e. macro) in the annotation. The resulting `ld_data` function may be scheduled and called like any other function, but Exo's C code generator will instead emit the C code "`config_ld({src}.strides...)`", with argument placeholders `{src}` and `{dst}` substituted

appropriately. Exo provides a `replace()` scheduling primitive (Section 4.2.7) for matching code in one procedure with the body of another procedure (including an `@instr` like `ld_data`), then *replacing* the matched code with an appropriate procedure call. Because the annotation alone drives code generation, the body of an instruction has no effect on the generated code; it instead serves as a semantic specification of the instruction for the purposes of checking correctness and program equivalence during scheduling. This approach carries several benefits and tradeoffs. First, programmers need not learn any specification language beyond Exo itself. Second, Exo entrusts programmers with the responsibility of verifying the link between the Exo procedure and its annotation. Third, programmers can use instructions in clever ways, including as an escape hatch—a prefetch instruction, for instance, can be modeled with a no-op procedure and thereby inserted anywhere.

4.1.4 Configuration State

We could issue this `replace()` primitive now to schedule the accelerator instructions. However, the C code has fused the expensive `config_ld` instruction to the `mvin` instruction we are really interested in scheduling. Since the stride does not actually change during the kernel, this will cause the accelerator pipeline to repeatedly flush and stall. We must somehow schedule the configuration instruction independently of the actual load. Therefore, we need a way to define hardware state. Exo models hardware configuration state as global structs of control variables annotated with `@config`. When defining a configuration, the programmer chooses whether to realize it as DRAM-resident data or to disable direct access to it, much as with a custom memory; in the latter case no global struct is generated. The following code models the stride configuration state in Exo.

in hw_lib.py

```
@config
class ConfigLoad:
    src_stride : stride
@instr("config_ld({s});")
def config_ld_def(s : stride):
    ConfigLoad.src_stride = s
```

Here, `ConfigLoad` defines a global struct of configuration variables, here containing a single `src_stride` field that models the state of the stride hardware parameter. We also write an instruction definition, `config_ld_def`, that updates the `src_stride` field. Now we can write a new instruction for the 16×16 load without the `config_ld` setup:

in hw_lib.py

```
@instr("mvin({src}.data, {dst}.data, {m}, {n});")
def real_ld_data(...):
    assert ConfigLoad.src_stride == stride(src, 0)
    # same as ld_data
```

Using scheduling instructions, we will rewrite the body of `ld_data` into a call to `config_ld_def()`, followed by a call to `real_ld_data()`. First, we use the `write_config()` scheduling primitive to rewrite `ld_data` into the following:

```
def ld_data(...):
    assert m <= 16
    ConfigLoad.src_stride = stride(src, 0)
    for i in seq(0, n):
        for j in seq(0, m):
            dst[i,j] = src[i,j]
```
in hw_lib.py

Unlike previous scheduling primitives, `write_config()` only partially preserves procedure equivalence—the new `ld_data()` is only equivalent *up to* the configuration state `ConfigLoad.src_stride` (details in Section 4.2.6). In general, Exo needs to reason about this kind of program equivalence modulo configuration state (see definition 3.11). Since the statement `ConfigLoad.src_stride = ...` is equivalent to the body of `config_ld_def`, and the statement `for i in seq(...):...` is equivalent to the body of `real_ld_data`, we can now `replace()` the body of `ld_data` with the two calls, as desired:

```
def ld_data(...):
    assert m <= 16
    config_ld_def(stride(src, 0))
    real_ld_data(n, m, src, dst)
```
in hw_lib.py

By following this same procedure, we can create instruction abstractions for our 16x16 matmul and store instructions. At last, we can replace the code in `gemm` with calls to `ld_data` and inline its definition.

```
def gemm(...):
    res: R[...] ...
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Loading C to res
            for ko in seq(0, 8):
                config_ld_def(stride(A, 0))
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```
in app.py

We will then hoist the call to `config_ld_def` using scheduling primitives `reorder_stmts()`, `fission()`, as well as `remove_loop()`. Doing so will require Exo's program analysis to both reason about when different statements *commute* (can be reordered) as well as when they are *idempotent* (allowing the loop to be removed). To further complicate matters, the presence of global, mutable *configuration* state means that fully precise analyses are undecidable and thus impossible in Exo. By using a ternary logic (Section 3.4), Exo can distinguish between memory locations that are *definitely* written to (a necessary condition for idempotency) and locations that are *maybe* written to (the relevant condition for commutativity).

```
def gemm(...):
    config_ld(stride(A, 0))
    res: R[...] ...
    for io in seq(0, 8):
        for jo in seq(0, 8):
            ... # Loading C to res
            for ko in seq(0, 8):
                real_ld_data(16, 16, A[...], a[...])
            ... # etc. etc.
```

in app.py

The code above is schematic rather than the final optimized implementation; the complete schedule and the resulting object code used in our evaluation appear in Appendix A.1. All of the above code transformations are achievable using the scheduling primitives discussed in the next section.

4.2 Scheduling Primitives

This section provides a complete list of Exo's scheduling primitives: for each, we give the code transformation it performs and the safety condition it must discharge. Primitives are the transformations built into the compiler, and they are deliberately as low-level as possible, giving performance engineers the fine-grained control needed to reach peak performance. Granularity comes at the cost of quantity: covering a wide range of optimization scenarios takes 46 of them. Their number is not a burden on the user, however, because primitives are not the only vocabulary available. In Chapter 5 we introduce *user-defined scheduling operators*, which live outside the compiler and are composed from the primitives presented here. It is this composition from primitives that lets them inherit the same safety guarantees.

Throughout this section we follow a uniform naming convention for the arguments of each primitive. Recall from Section 3.3.2 that every scheduling operator has type $\mathsf{Op} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc}$. We write `p` for the Proc argument, the procedure being scheduled, which is always the first argument and whose rewritten copy is always the result. The remaining arguments name the object code to be rewritten and parameterize the rewrite: we write `loop` for a cursor to a loop, `s` for a cursor to a statement, and `e` for a cursor to an expression, using `s1`, `s2`, and so on when a primitive takes several. Cursors are explained in full in Chapter 5; for the purposes of this section it suffices to think of them as pointers into the object code of `p`, naming the loop, statement, or expression the primitive should act on.

4.2.1 Loop Transformations

Loops are the dominant structure in the kernels Exo targets, and consequently loop rewrites make up the largest group of scheduling primitives (Tables 4.1 and 4.2). They fall into three families: those that change the *shape* of the iteration space, those that change its *extent*, and those that change the *nesting* of statements within it.

The shape-changing primitives are `reorder_loops()`, `divide_loop()`, `mult_loops()`, and `divide_with_recompute()`. `reorder_loops()` interchanges a perfect two-loop nest; it is safe when the inner loop's bounds are independent of the outer index and the body commutes

Scheduling operation	Code transformation	Safety conditions
`reorder_loops(p, loops)`	`for i:` `  for j:` `    s` ⇝ `for j:` `  for i:` `    s`	The `j` loop's bounds cannot depend on `i`. The loop body `s` commutes for different (i, j) and (i', j') pairs.
`divide_loop(` `  p,` `  loop,` `  c,` `  [io, ii],` `  tail_strategy` `)`	`# tail_strategy=perfect` `for i < I:` `  s` ⇝ `for io < I/c:` `  for ii < c:` `  s[i ↦ c*io+ii]`	For perfect `tail_strategy`, the loop bound is perfectly divisible by `c`.
	`# tail_strategy=guard` `for i < I:` `  s` ⇝ `for io < I/c:` `  for ii < c:` `  if c*io+ii < I:` `  s[i ↦ c*io+ii]`	
	`# tail_strategy=cut` `for i < I:` `  s` ⇝ `for io < I/c:` `  for ii < c:` `  s[i ↦ c*io+ii]` `for ii < I%c:` `  s[i ↦ c*I/c+ii]`	
	`# tail_strategy=cut_and_guard` `for i < I:` `  s` ⇝ `for io < I/c:` `  for ii<c:` `  s[i ↦ c*io+ii]` `if I%c > 0:` `  for ii < I%c:` `  s[i ↦ c*I/c+ii]`	
`divide_with_recompute(` `  p, loop, N, c, [io, ii]` `)`	`for i < I:` `  s` ⇝ `for io < N:` `  for ii < c+I-N*c:` `  s[i ↦ c*io+ii]`	`s` is idempotent and `N*c` $\leq$ `I`.

Table 4.1: Loop-transformation primitives, part I: loop interchange and loop division.

across iteration pairs. `divide_loop()` splits a single loop of extent `I` into an outer loop over tiles and an inner loop of length `c`, and it is the workhorse for tiling and vectorization. Because `I` need not be divisible by `c`, the primitive is parameterized by a `tail_strategy`: `perfect` enforces divisibility as a safety condition, `guard` predicates the body on an in-bounds test, `cut` peels the remainder into a separate loop, and `cut_and_guard` peels the remainder and additionally guards it against being empty. `mult_loops()` is the inverse: it flattens a two-loop nest whose inner extent is a constant `c` into a single loop of extent `I*c`, provided the inner loop is the only statement in the outer body. `divide_with_recompute()` is a variant of `divide_loop()` in which the outer loop count `N` is chosen independently of the inner extent, so that adjacent tiles *overlap* and boundary iterations are executed more than once. This is the pattern needed for stencil-style redundant computation, and it is sound only because the body is required to be idempotent.

The extent-changing primitives—`cut_loop()`, `join_loops()`, and `shift_loop()`—operate on the iteration bounds rather than the body. `cut_loop()` splits the range `[l, h)` at a cutoff `e` into two loops running over `[l, e)` and `[e, h)`, which is how one exposes a prologue or epilogue for separate treatment; `join_loops()` is its inverse, merging two adjacent loops

Scheduling operation	Code transformation	Safety conditions

`mult_loops(p, loops, k)`	`for i < I:`
	`  for j < c:`
	`  s` ⇝ `for k < I*c:`
	`  s[i ↦ k/c, j ↦ k%c]`
		The `j` loop is the only statement in the `i` loop's body. Also, `c` is constant.

`cut_loop(p,loop,e)`	`for i in l, h:`
	`  s` ⇝ `for i in l, e:`
	`  s`
	`for i in e, h:`
	`  s`
		The cutoff `e` lies between the loop bounds `l` and `h`.

`join_loops(`
`  p, loop, loop2`
`)`
	`for i in l1, h1:`
	`  s`
	`for i2 in l2, h2:`
	`  s` ⇝ `for i in l1, h2:`
	`  s`
		The loops are adjacent, have identical bodies, and `h1 = l2`.

`shift_loop(p, loop, e)`	`for i in l, h:`
	`  s` ⇝ `for i in e, h+e-l:`
	`  s`
		The new lower bound `e` ≥ 0.

`fission(p, gap)`	`for i:`
	`  s1`
	`─────`
	`  s2` ⇝ `for i:`
	`  s1`
	`─────`
	`for i:`
	`  s2`
		`s2` cannot depend on allocations in `s1`, and effects of `s1` and `s2` commute.

`remove_loop(p, loop)`	`for i:` `  s` ⇝ `s`	`s` is idempotent and cannot depend on `i`. Loop executes at least once.

`add_loop(`
`  p, s, i, hi, guard`
`)`
	`# guard = False`
	`s` ⇝ `for i < hi:`
	`  s`
	`─────`
	`# guard = True`
	`for i < hi:`
	`s` ⇝ `  if i == 0:`
	`  s`
		`s` is idempotent, and `hi` is positive.

`unroll_loop(p, loop)`	`for i in lo, hi:`
	`  s` ⇝ `s[i ↦ lo]`
	`...`
	`s[i ↦ hi - 1]`
		`hi` and `lo` are constants, `hi - lo` > 0.

Table 4.2: Loop-transformation primitives, part II: flattening, bound manipulation, fission, loop insertion and removal, and unrolling. `mult_loops()` inverts `divide_loop()`, and `join_loops()` inverts `cut_loop()`.

with identical bodies and abutting bounds. `shift_loop()` rebases a loop to start at a new lower bound `e`, normalizing bounds so that subsequent primitives (or unification against an instruction) can apply.

Finally, `fission()`, `remove_loop()`, `add_loop()`, and `unroll_loop()` change how statements are nested inside loops. `fission()` splits a loop at a designated gap in its body, distributing the loop over the two halves; it requires that the second half neither depend on allocations in the first nor fail to commute with it. `remove_loop()` deletes a loop whose body is idempotent and independent of the loop index, which is safe only when the loop is known to execute at least once; `add_loop()` is its inverse, and may optionally guard the body so that it executes on a single iteration rather than relying on idempotence of repetition. `unroll_loop()` fully unrolls a loop with constant bounds.

4.2.2 Scope Transformations

Scheduling operation	Code transformation	Safety conditions
`specialize(p, s, conds)`	`s` ⇝ `if conds[0]: s else: if conds[1]: s else: ...`	`conds` only contains valid boolean expressions.
`fuse(p, scope, scope2)`	`for i<I: s` / `for i<I: s2` ⇝ `for i<I: s s2`	Equivalent upper loop bound `I` in its context. Iterations of `s` commute with any iteration of `s2`.
	`if e: s` / `if e: s2` ⇝ `if e: s s2`	Equivalent if conditional expression `e` in its context.
`lift_scope(p, scope)`		For all cases, `scope` is the only statement in its parent's body.
	`if e: if e2: # scope s else: s2 else: s3` ⇝ `if e2: if e: s else: s3 else: if e: s2 else: s3`	N/A
	`if e: # scope for i: ...` ⇝ `for i: if e: ...`	`if` statement cannot have an `else` clause.
	`for i: if e: # scope s else: s2` ⇝ `if e: for i: s else: for i: s2`	`e` cannot depend on `i`.

Table 4.3: Scope-transformation primitives.

Scope transformations restructure the control-flow nesting of a procedure (Table 4.3). `specialize()` replaces a statement `s` with a chain of guarded copies of `s`, one per condition in `conds`, with the original as the final `else` branch. Since every branch has the same body, the rewrite is unconditionally sound; its purpose is to create contexts in which a condition is known to hold, so that later primitives can exploit it—for instance, to specialize a tail case away from a fast path. `fuse()` merges two adjacent scopes into one. For loops, it requires that the two loops have equivalent bounds in their context and that the bodies commute across iterations; for conditionals, it requires that the two guards be equivalent in context. `lift_scope()` hoists a scope out of its enclosing scope and handles three cases:

Scheduling operation	Code transformation	Safety conditions
`reorder_stmts(p, s1, s2)`	`s1` ⇝ `s2` `s2` ⇝ `s1`	`s1` and `s2` commute.
`commute_expr(p, e)`	`x+y` ⇝ `y+x` `x*y` ⇝ `y*x`	N/A

Table 4.4: Code-rearrangement primitives.

lifting a conditional out of a conditional (which duplicates the outer `else` branch), lifting a conditional out of a loop, and lifting a loop out of a conditional. In every case the lifted scope must be the only statement in its parent's body. Lifting an `if` out of a `for` additionally forbids an `else` clause, and lifting a `for` out of an `if` requires that the guard not depend on the loop index—otherwise the two forms would not agree on which iterations execute.

4.2.3 Code Rearrangement

Table 4.4 presents two local rearrangement primitives. `reorder_stmts()` exchanges two statements only when their effects commute. `commute_expr()` exchanges the operands of an addition or multiplication using the corresponding commutative identity.

4.2.4 Buffer Transformations

Where loop transformations reshape the iteration space, buffer transformations reshape memory (Tables 4.5 and 4.6). These primitives are what let a schedule introduce accelerator-local staging buffers, size them to fit a scratchpad, and lay them out in the shape an instruction expects.

Table 4.5 collects primitives that move allocations, manage buffer lifetimes, and change buffer dimensionality. `lift_alloc()` hoists an allocation out of a loop, so that a temporary is allocated once rather than per iteration; it requires that the buffer's dimensions not depend on the loop index and that hoisting introduce no loop-carried dependency. `sink_alloc()` is the inverse. `delete_buffer()` removes a dead buffer, while `reuse_buffer()` rebinds all uses of `b` to an identically typed and sized buffer `a` that is dead by the time `b` is allocated—the standard way to reduce scratchpad pressure. `expand_dim()` adds a dimension to a buffer and indexes it by an expression `e`, which must be provably in-bounds; combined with `lift_alloc()`, this is how a scalar accumulator becomes a per-iteration vector. `rearrange_dim()` permutes the dimensions of a buffer according to `p_vec`, and `unroll_buffer()` explodes a constant-size dimension into separate scalar buffers, which is how one names individual vector registers. `bind_expr()` stages an expression into a fresh temporary and substitutes the temporary for its occurrences, optionally performing common-subexpression elimination.

Table 4.6 collects the reshaping and staging primitives. `resize_dim()` narrows a buffer dimension to the interval that is actually accessed, shifting accesses by an offset `off`; with `fold=True` it instead wraps accesses modulo the new size `sz`, turning the buffer into a circular buffer. The folded case carries the additional condition that no access reaches more than `sz` elements behind the largest access so far, which is what guarantees that a live value is never overwritten. `divide_dim()` and `mult_dim()` are the memory analogues of `divide_loop()`

Scheduling operation	Code transformation	Safety conditions
`lift_alloc(p, a)`	`for i:` / `  a: T[sz]` / `  s` ⇝ `a: T[sz]` / `for i:` / `  s`	Dimensions `sz` don't depend on `i`. No loop-carried dependencies.
`sink_alloc(p, a)`	`a: T` / `for i:` / `  s` ⇝ `for i:` / `  a: T` / `  s`	`a` is only accessed in the `i` loop. No loop-carried dependencies.
`delete_buffer(p, a)`	`a: T` / `s` ⇝ `s`	`a` should be dead.
`reuse_buffer(p, a, b)`	`a : T[sz]` / `...` / `b : T[sz]` / `s` ⇝ `a : T[sz]` / `...` / `s[b ↦ a]`	`a` and `b` have the same type and size. `a` is dead after `b`'s allocation.
`expand_dim(p, a, sz, e)`	`a: T[_]` / `s` ⇝ `a: T[_]` / `s[a[_] ↦ a[e, _]]`	`sz` is positive, `e` only uses existing variables, and `e < sz` in all contexts.
`rearrange_dim(p, a, p_vec)`	`# p_vec = [2, 0, 1]` / `a: T[N, M, K]` ⇝ `a: T[K, N, M]`	`a` cannot be windowed or passed in function calls.
`unroll_buffer(p, a, dim)`	`a: T[c]` ⇝ `a1: T` / `...` / `ac: T`	For this dimension, `a` has constant size and index accesses. `a` cannot be windowed along this dimension.
`bind_expr(p, e, a, cse)`	`s` ⇝ `a: T` / `a = e` / `s[e ↦ a]`	N/A

Table 4.5: Buffer-transformation primitives, part I.

and `mult_loops()`, splitting a constant-size dimension into two or collapsing two dimensions into one. Finally, `stage_mem()` is the most substantial of the buffer primitives: given a statement block `s` and a window `w` into a buffer `a`, it allocates a temporary `tmp` shaped like `w`, copies `w` into `tmp` before `s`, rewrites `s` to operate on `tmp`, and copies back afterwards. Its safety condition is that `s` touch `a` only through `w`. This single rewrite is what introduces the load-compute-store structure that accelerator memory hierarchies require; the surrounding copy loops are ordinary Exo loops and are typically then scheduled and `replace()`d with DMA or load instructions.

4.2.5 Simplification

Scheduling rewrites tend to leave behind arithmetic clutter, dead branches, and redundant writes. `simplify()` applies arithmetic simplification and trivial branch elimination across the whole procedure, and it is unconditionally sound. `eliminate_dead_code()` removes a loop whose bounds prove it never executes or collapses a conditional whose guard is equivalent to `True` or `False` in its context—the contextual qualification is essential, since a guard such as `c*io + ii < I` is only decidable given the bounds of the enclosing loops. `rewrite_expr()` replaces an expression with a contextually equivalent one, discharging the obligation to the SMT solver. `merge_writes()` combines two consecutive writes or reduces to the same destination into a single statement, handling all four combinations of write and reduce, and

Scheduling	Code transformation		Safety conditions

`resize_dim(`
`  p, a`
`  dim, sz,`
`  off, fold`
`)`
	`## fold = False`
	`a: T[_]`
	`s`
		⇝ `a: T[_]`
		`s[a[e] ↦ a[e-off]]`
			`sz` > 0, every `a[e]` in `dim`-th dimension has `off <= e < off+sz`.

	`## fold = True`
	`a: T[sz]`
	`s`
		⇝ `a: T[sz]`
		`s[a[e] ↦ a[(e-off)%sz]]`
			`sz` > 0; each access window of `a` in the `dim`-th dimension is at most `sz` wide. Also, `a` accesses never go more than `sz` before the largest so far.

`divide_dim(`
`  p, a,`
`  dim, c`
`)`
	`# dim = 0, c = 4`
	`a: R[12, _]`
	`s`
		⇝ `a: R[3, 4, _]`
		`s[a[i, _] ↦ a[i/4, i%4, _]]`
			`a`'s `dim`-th dimension size is constant and divisible by `c`.

`mult_dim(`
`  p, a,`
`  dim, dim2`
`)`
	`# dim = 0, dim2 = 2`
	`a: R[n, _, 4]`
	`s`
		⇝ `a: R[4*n,_]`
		`s[a[i, _, j] ↦ a[4*i+j, _]]`
			`a` cannot be windowed or passed in function call. One of the dimensions is constant size.

`stage_mem(`
`  p, s, a,`
`  w, tmp`
`)`
	`# w = a[0:n, j-1:j]`
	`for i < n-1: # s`
	`  a[i,j] = a[i+1,j-1]`
		⇝ `tmp: T[n, 2]`
		`for k0 < n:`
		`  for k1 < 2:`
		`  tmp[k0, k1] = a[k0,j-1+k1]`
		`for i < n-1:`
		`  tmp[i, 1] = tmp[i+1, 0]`
		`for k0 < n:`
		`  for k1 < 2:`
		`  a[k0, j-1+k1] = tmp[k0, k1]`
			The code block `s` does not access buffer `a` outside of the given `window`.

Table 4.6: Buffer-transformation primitives, part II.

requires only that the second right-hand side not read the destination. `inline_window()` substitutes a window expression for its name, and `inline_assign()` substitutes an assigned expression for the variable, provided the variable is not subsequently written.

4.2.6 Configuration State

Many accelerators are controlled not only by their instructions but by persistent configuration state: mode registers, stride registers, or data-type selectors that are written once and implicitly read by subsequent instructions. Exo models this state as a set of global variables (Section 4.1.4) and provides three primitives for manipulating it (Table 4.8). `bind_config()` hoists an expression `e` into a configuration field and replaces its occurrences with reads of that field; `write_config()` inserts a configuration write at a designated gap; and `delete_config()` removes one. Each of the three carries the same safety condition, namely that the affected field not be read by any code that executes afterwards.

This locational condition is value-insensitive. It can reject a hoist whenever a later read exists, even when that read would observe an identical value. Chapter 6 revisits configuration

Scheduling operation	Code transformation	Safety conditions
`simplify(p)`	does arithmetic simplifications and trivial branch elimination on the entire procedure	N/A

`eliminate_dead_code(`
`  p, scope`
`)`
	`for i in l, h:` ⇝ `pass`	The loop runs 0 times, e.g. `l`≥`h`.

	`if e:`
	`  s`
	`else`
	`  s2` ⇝ `s` or `s2`
		`e` is equivalent to `True` or `False` in its context.

`rewrite_expr(p, e1, e2)`	`e1` ⇝ `e2`	`e1` and `e2` are equivalent in the context in which `e1` appears.
`merge_writes(p, s1, s2)`	`x = e1` `x = e2` ⇝ `x = e2`	The two statements write to the same destination `x`. `e2` cannot read from `x`.
	`x += e1` `x = e2` ⇝ `x = e2`	
	`x = e1` `x += e2` ⇝ `x = e1 + e2`	
	`x += e1` `x += e2` ⇝ `x += e1 + e2`	
`inline_window(p, w)`	`w = a[_]` `s` ⇝ `s[w ↦ a[_]]`	N/A
`inline_assign(p, x = e)`	`x = e` `s` ⇝ `s[x ↦ e]`	`x` cannot be written to in the block `s` after `s`.

Table 4.7: Simplification primitives.

binding with a value-sensitive analysis, enabling scheduling operations that remain safe even when the bound field is reread with an unchanged value across loop iterations.

4.2.7 Code Replacement and Instruction Selection

The primitives considered so far rewrite code within a single procedure. Those in Table 4.9 instead change the boundaries between procedures, and it is through them that a schedule finally reaches the hardware. `inline()` inlines at a call site, and `extract_subproc()` is its inverse, isolating a statement block into a named procedure and leaving a call in its place; both are unconditionally sound. `call_eqv()` replaces a call to a subprocedure `foo` with a call to a sub-procedure `bar` known to be equivalent, typically because `bar` was scheduled from `foo`. This equivalence is tracked by *provenance*: the Exo system records the sequence of transformations by which `foo` was turned into `bar`. The notion of an equivalent subprocedure is complicated by exactly those scheduling primitives which pollute configuration state (e.g. `bind_config()`); to handle these, Exo tracks the lattice of equivalence relations modulo different parts of the configuration state introduced in Definition 3.11.

The `replace()` scheduling primitive takes a designated statement block `s` and *replaces* it with a call to a designated subprocedure `foo`. In particular, when `foo` is an `@instr`, this rewriting performs instruction selection. In other cases, it allows Exo programmers to manage code size trade-offs, as well as to more neatly abstract and organize their code.

Our implementation of `replace()` is based on a form of unification modulo linear equalities.

Scheduling operation	Code transformation	Safety conditions
`bind_config(p, e, cfg, field)`	`s` $\rightsquigarrow$ `cfg.field = e` `s[e ↦ cfg.field]`	`cfg.field` is not read by code that executes afterwards.

`delete_config(p, cfg.field = _)`	`s`
	`cfg.field = _` $\rightsquigarrow$ `s`
	`s2`
		`cfg.field` is not read by code that executes afterwards.

`write_config(`
`  p, gap, cfg, field, e`
`)`
	$\frac{\texttt{s}}{\texttt{s2}} \rightsquigarrow$ `s`
	`cfg.field = e`
	`s2`
		`cfg.field` is not read by code that executes afterwards.

Table 4.8: Configuration state primitives. Unlike the other scheduling primitives, these preserve equivalence only *modulo* the configuration fields they write (Definition 3.11).

Scheduling operation	Code transformation	Safety conditions
`inline(p, foo)`	Inline a callsite of `foo`	N/A
`replace(p,s,instr)`	Replace `s` with a call to an instruction `instr`	`s` and the `instr` function are unifiable.
`call_eqv(p,foo,bar)`	Replace `foo` with a call to an equivalent procedure `bar`	The two procedures `foo` and `bar` are equivalent, e.g. scheduled from the same procedure.
`extract_subproc(p,s,foo)`	Isolates `s` into a function named `foo` and replaces `s` with a call to `foo`.	N/A

Table 4.9: Code-replacement and instruction-selection primitives. `replace()` is the mechanism by which a scheduled statement block is matched against a hardware instruction; `inline()` is its inverse.

First, we attempt to unify (i.e. pattern match) the body of the subprocedure `foo` with the designated statement block `s`. When doing this, the arguments of `foo` are designated as unknowns, the free variables of `s` as known symbols, and any symbols introduced or bound in the body of `foo` or within `s` are unified. The ASTs are required to match exactly with respect to statements and with respect to all expressions that are not simply integer-typed control. Equivalences between integer-typed control expressions are instead recorded as a system of linear equations, to be solved in a second step.

If Exo did not support windowing, then we could determine expressions for the unknown argument variables by symbolically solving the resulting linear system. However, the possibility of windowing expressions appearing as arguments forces us to make categorical choices between different possible windowing expressions, resulting in disjunctions as well as conjunctions of linear equalities. For example, if `replace()` is asked to infer a 1-dimensional window onto a 2-dimensional buffer `x`, it could infer an expression of the form `x[i,jlo:jhi]` or one of the form `x[ilo:ihi,j]`. To handle this complication, we observe that all inferred integer expressions must be affine combinations of the known, free variables. Therefore, we can transform our symbolic-linear-system problem into a linear system in the unknown *coefficients* of these affine expressions. Once encoded in this way, the problem falls into a decidable theory and can be discharged to an SMT solver (in our implementation, Z3).

As a concrete example, consider unifying the body of `bar` against the body of `foo`:

Scheduling operation	Code transformation	Compilation safety check
`set_memory(p, a, MEM')`	`a @ MEM` ⇝ `a @ MEM'`	Ensures that the accesses to the buffer obey the custom memory read/write/window definition, and the caller and callee have the same memory types
`set_precision(p, a, T')`	`a: T` ⇝ `a: T'`	Checks the caller, callee, and the both sides of binary operation have the same precision types
`parallelize_loop(p, loop)`	Annotate loop as parallel	Ensures that the loop iterations do not have RAW or WAW dependencies.
`set_window(p, a)`	`a: T[_]` ⇝ `a: [T][_]`	Checks the caller and callee have the same window shapes

Table 4.10: Backend-checked annotation primitives. Unlike the primitives in the preceding tables, these are not checked at rewrite time; their side conditions are discharged by the compiler backend immediately prior to code generation.

```
@proc
def bar(n : size, src : [R][n], dst : [R][n]):
    for i in seq(0, n):
        dst[i] += src[i]

@proc
def foo(x : R[50, 2]):
    for j in seq(0, 50):
        x[j,1] += x[j,0]
```

Substituting `bar` into `foo` requires matching `dst[i]` with `x[j,1]`, and it is not *a priori* clear which dimension of `x` should be windowed and which should be point-accessed. Here the constraints are sufficient to determine that the body of `foo` may be replaced with the call `bar(50, x[0:50, 0], x[0:50, 1])`.

4.2.8 Backend-Checked Annotations

All the scheduling primitives that we have discussed so far are safety-checked within their rewrite process. By contrast, consistency of precision types, memory annotations, and window annotations is checked in the backend, after all scheduling is complete and immediately prior to code generation (Table 4.10).

These properties are deferred to the backend check because the corresponding annotations are not local properties of a single rewrite. `set_memory()` retags a buffer with a different memory space, but whether that tagging is legal depends on every access to the buffer, including accesses in procedures the buffer is passed to; the backend checks that all accesses obey the custom memory's read/write/window definitions and that caller and callee agree on memory type. Likewise, `set_precision()` retypes a buffer, and the backend checks that callers, callees, and both operands of every binary operation agree on precision; `set_window()` converts a buffer argument into a window argument, and the backend checks that caller and callee windows have matching shapes.

`parallelize_loop()` annotates a loop as parallel, and the backend checks that its iterations carry no RAW or WAW dependences. Chapter 7 generalizes this deferred-validation pattern to GPUs, where thread assignment, distributed-memory ownership, instruction convergence, and

asynchronous synchronization must be checked altogether. It adds collective and distributed-memory analyses together with an abstract-machine synchronization check of sequential-parallel equivalence. Deferring these checks allows a schedule to pass through intermediate states with temporarily inconsistent annotations and postpones validation until the final target mapping is fixed.

4.3 Case Studies

This chapter showed how a hardware target can be externalized: instructions, memories, and configuration state are defined in user or library code rather than baked into the compiler. We put that design to the test on two different targets. Gemmini [39] is a research accelerator whose ISA was still in flux and whose baselines its designers wrote by hand; x86 with AVX-512, by contrast, runs baselines near theoretical peak after decades of tuning. Of both we ask the same question: does externalizing the target cost performance, code size, or maintainability?

4.3.1 Gemmini

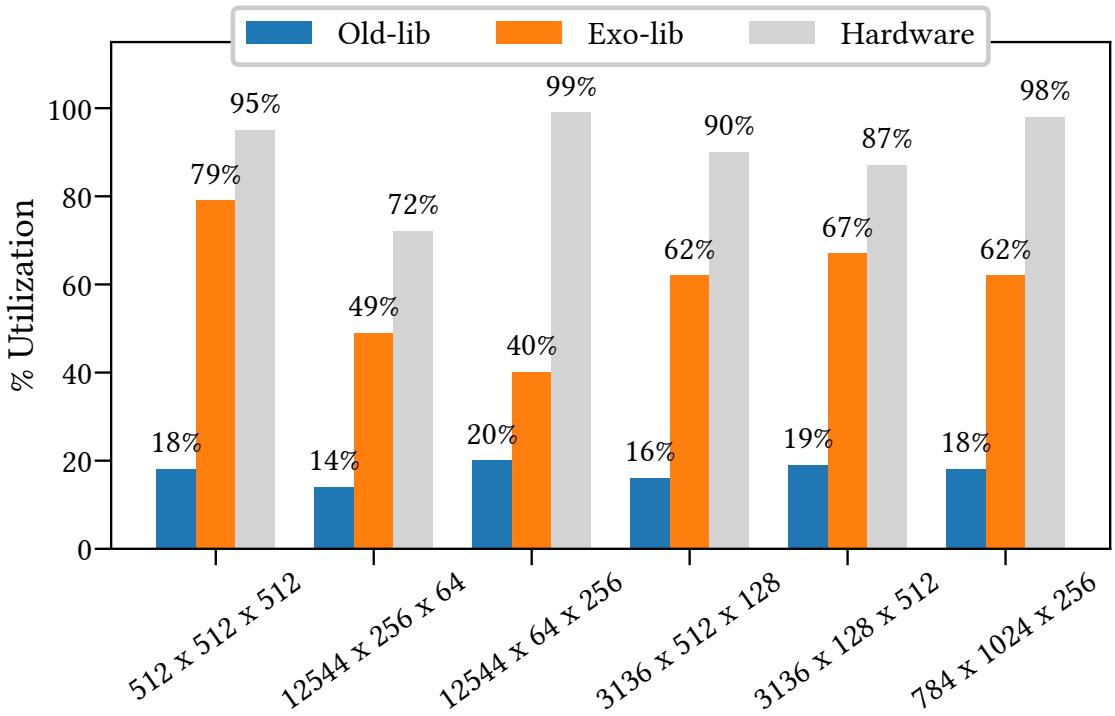
We targeted Gemmini's default architectural instantiation, which includes a 16x16 systolic array that performs block matrix multiplications, a 256KB scratchpad for quantized inputs and weights, and a 64KB accumulator for partial sums. Gemmini's instruction-set architecture (ISA) includes low-level instructions to move strided matrices to and from the scratchpad, as well as instructions to calculate dot products and perform nonlinear activations on this data.

Gemmini also ships with a hand-written C library for common DNN kernels. This library wraps calls to Gemmini's low-level ISA in statically scheduled, hand-tuned loops. However, Gemmini can also be built with hardware loop unrollers that dynamically schedule these kernels to maximize overlap between data loads, data stores, and matrix-multiply operations. The hardware implementations typically run much faster than the software implementations at the cost of hardware complexity, area, power consumption, and reduced scheduling flexibility. The hardware kernels also have fixed loop orders and dataflows, while the software can adapt these to different tensor shapes.

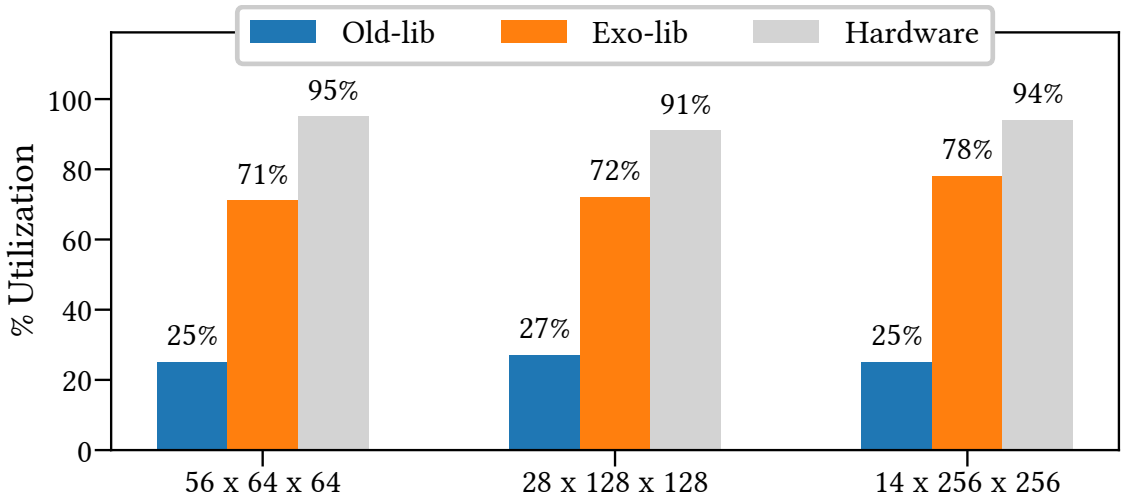
We implemented kernels for matrix multiply (MATMUL) and convolutional (CONV) layers in Exo and compared their performance against Gemmini's handwritten C library and hardware loop unrollers. The results are shown in Figures 4.1(a) and 4.1(b), respectively. The tensor shapes in both are selected from those in a ResNet-50 DNN with a batch size of 4.

On average, Exo-generated code outperforms Gemmini's handwritten C library by 3.5× on the MATMUL sizes listed above and achieves 67% of the performance of the hardware loop unrollers. For the convolutions listed, it runs 2.9× faster than the handwritten library and is competitive with the hardware loop unroller, achieving 79% of its performance.

Note that the hardware loop unrollers use optional hardware resources (increasing area and power consumption) which are not available to Exo or the handwritten C library. However, we expect that changing Gemmini's ISA to support coarser-granularity instructions and better schedules may be able to close this performance gap in the future, providing software-programmable performance comparable to the inflexible hardware loop-unrollers.



(a) MATMUL utilization (as a percentage of peak FLOPS). X-axis labels are the sizes of matrices in N x M x K.



(b) CONV utilization (as a percentage of peak FLOPS). X-axis labels are the shape of convolution in *output dimension* x *output channel* x *input channel*.

Figure 4.1: Performance of Exo-generated code on the Gemmini DNN accelerator. Exo-generated code achieves much higher performance than the DNN kernels hand-written by the designers of Gemmini (Old-lib). Gemmini's dynamically scheduled hardware loop unrollers (Hardware) outperform Exo by using additional hardware resources but therefore require additional chip area and power consumption.

Finally, Exo enabled faster codesign of Gemmini's hardware-software interface. When we started targeting Gemmini, its low-level hardware-configuration instructions had many side effects that made optimizations difficult to reason about, limiting the performance we could achieve. We worked with the Gemmini hardware designers to disaggregate these configuration instructions into more orthogonal components; e.g. instructions which configured Gemmini's memory units would no longer have any side effects on the arithmetic units. 46 lines in Gemmini's handwritten C library had to be updated after this change, compared to only 5 in Exo's implementation. Exo made it easier for programmers to target fluid and changing hardware targets, which is common when developing new accelerators.

4.3.2 x86

As an acid test of the language design, we optimized matrix-matrix multiplication (SGEMM) for x86, where we can compare against state-of-the-art libraries that run near theoretical peak compute throughput. We chose to target single-core x86 with AVX-512 extensions.[1]

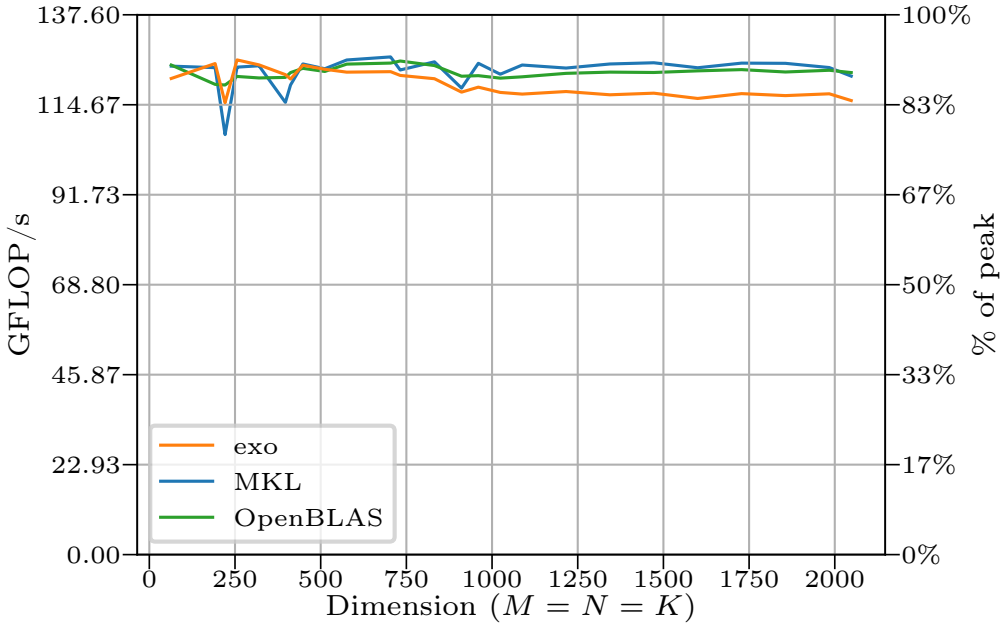
Recall that the computation is given by $C \mathrel{+}= A \cdot B$ where C is $M \times N$, A is $M \times K$, and B is $K \times N$. Our Exo implementation decomposes the problem as follows: at the deepest level of blocking, a register-blocked microkernel accumulates the inner dimension into a 6×64 panel of C, the output matrix. The level above the microkernel handles edge cases by dispatching to *specialized* versions of the microkernel for each edge case. Along the bottom, five distinct kernels are needed as they are always 64 elements wide and never 0 or 6 tall; similarly, four distinct kernels are needed along the right. The variable tail on the right edge is handled by masked loads. Finally, one level above this handles staging memory and blocking.

Every one of these routines was produced by scheduling and specializing a single, naive implementation of SGEMM consisting of three nested loops. Unification and equivalent-call replacement were crucial for avoiding any sort of error-prone, manual optimization.

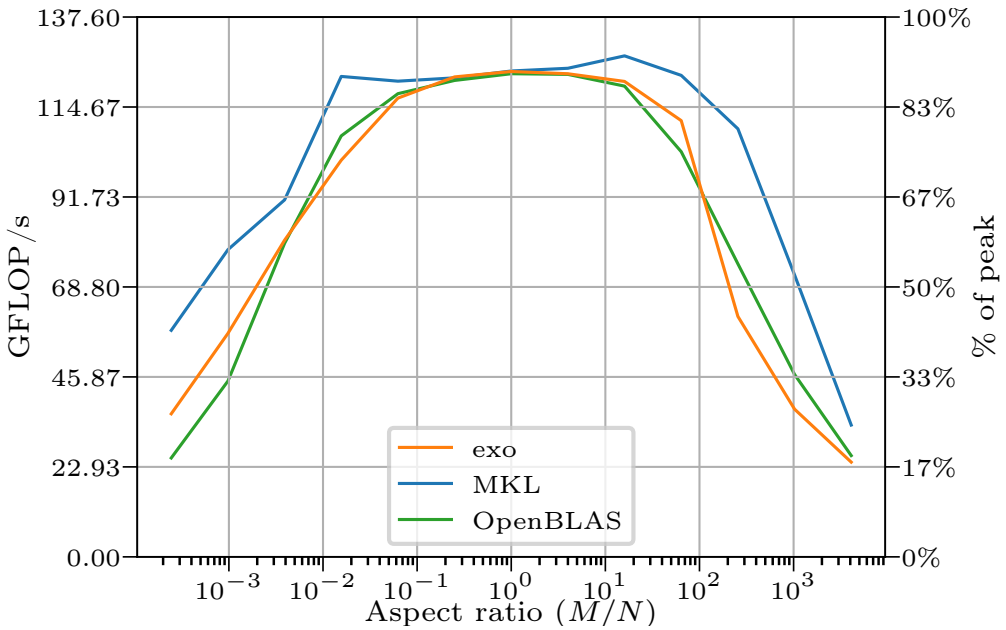
The performance results are shown in Figure 4.2. All benchmarks were run on an Intel i7-1185G7 at 4.3 GHz, a Tiger Lake CPU with AVX-512 instructions and peak single-precision floating-point performance of 137.60 GFLOPs. We tested our SGEMM against the hand-optimized implementations in Intel's MKL and the open-source OpenBLAS in two experiments. First (Figure 4.2a), we tested square matrices , so $M = N = K$. Each implementation performs quite closely (within measurement noise), between 80-95% of theoretical peak FLOPS across the parameter range.

Second (Figure 4.2b), we tested our SGEMM on a fixed workload, but with a variable aspect ratio for C. Specifically, we fix the inner dimension $K = 512$ and the product $MN = 512^2$, then we sweep across the ratio M/N keeping the total FLOP count identical across experiments. Here, Exo matches OpenBLAS almost exactly, but MKL pulls ahead of both implementations when the aspect ratio is very far from square. MKL includes more specialized kernels for these extreme aspect ratios, which would be natural to do with further scheduling in Exo, as well.

[1] Although multicore implementations are valuable, single-core workloads are representative of practice (ML inference in interactive web services is often run batch-parallel on single-core kernels), and the baselines are highly optimized.



(a) SGEMM performance on square matrices. We approximately match other systems on square matrices.



(b) SGEMM performance with fixed workload and variable output aspect ratio. $K = 512$ and $M \times N = 512^2$, with the ratio of M to N varying. We match OpenBLAS performance across aspect ratios.

Figure 4.2: SGEMM performance compared to state-of-the-art libraries on x86. Benchmarks were run on one core of an Intel i7-1185G7 running at 4.3GHz.

Impl.	N	W	H	IC	OC	% of peak
Exo	5	82	102	128	128	40.50%
Halide	5	82	102	128	128	40.59%
oneDNN	5	82	102	128	128	40.55%

Table 4.11: Summary of x86 CONV performance results. Single-threaded performance of various implementations with no padding and unit stride. A ReLU activation is applied. Benchmarks were run on an Intel i7-1185G7 running at 4.3GHz on a single core. The size was chosen to match the previously published hand-scheduled Halide implementation. All three specialize or JIT to tune their code to specific sizes.

For a final experiment, we tried to replicate the convolutional-layer performance of a highly tuned implementation provided by the Halide project. State-of-the-art convolutions specialize or JIT-compile code templates to particular input, output, and kernel sizes. In Halide's case, it specialized to a batch size of 5, a kernel size of 3×3, an output size of 80×100, and 128 channels for both input and output. There is no padding and unit stride is used. We configured Intel's oneDNN convolution to use these parameters and scheduled a basic description of convolution in Exo to these parameters, too. The results are shown in Table 4.11. Our CONV performs almost identically to the optimized baselines.

Overall, we believe these results show that Exo can be used to achieve performance competitive with state-of-the-art, highly hand-tuned libraries on x86.

4.3.3 Code Size

Table 4.12 summarizes some statistics regarding the size of Exo programs relative to hand-written C baselines.

On x86, our SGEMM schedule instantiates many specialized microkernels for handling loop-tail cases at higher levels. Unlike Gemmini, it does not have SGEMM-specific hardware to utilize that might reduce the scheduling burden. Even so, the basic algorithm is expressed in 11 statements (the function signature, three loops, an accumulation statement, and a handful of size assertions) and 162 scheduling directives. The generated C code totals 831 source lines of code (after formatting with "clang-format" and counting with "cloc"). This already constitutes a nearly 5x code size reduction, but a comparison to OpenBLAS (an established open-source implementation) is even more favorable: at least 1690 source lines of code[2] make up that implementation. MKL is more complex, still.

Although the x86 conv implementation is "only" half the size of the equivalent generated C, it is much more flexible since other specialized versions can be quickly instantiated by metaprogramming the schedule in Python. The size of the most comparable open-source implementation, Intel's oneDNN, is difficult to measure; just one file in the implementation measures well over 5000 source lines of code[3]. The size of the Halide code and schedule was nearly identical to ours: 64 relevant lines, compared to 62.

[2]Summing the source-line counts of the files mentioned in `kernel/x86_64/KERNEL.SKYLAKEX` for non-transposed SGEMM gives a very loose lower bound

[3]`src/cpu/x64/jit_avx512_common_conv_kernel.cpp`

App.	Platform	C (gen)	C (ref)	Alg.	Sched.
MATMUL	Gemmini	462	313	23	43
CONV	Gemmini	8317	450	26	44
SGEMM	x86	846	>1,690	11	162
CONV	x86	102	>5,400	23	39

Table 4.12: Source-code sizes for matrix multiplication and convolutional layer on Gemmini and x86. Gemmini implements a fixed-point matrix-multiply neural-network layer (with fused ReLU activation), while x86 implements the BLAS SGEMM kernel. Both implement a standard 2D convolutional layer with ReLU activation. The Exo sources are counted in lines of code for the algorithm and number of operations for the schedule. This is compared to the size of both the Exo-generated C and state-of-the-art reference implementations (Gemmini standard library, OpenBLAS, and oneDNN, respectively) in source lines of code.

The story is similar for our Gemmini kernels. Both the matmul and conv Exo implementations are an order of magnitude smaller than the original, handwritten C implementations. The large generated code sizes reflect the high degree of loop unrolling in the generated schedules. A real application would likely either resort to the C preprocessor to manage this complexity or not attempt the transformation at all (or as aggressively) beyond whatever the C compiler might choose to do automatically.

In this chapter, we obtain a preliminary form of amortization: a compact algorithm plus a kernel-specific schedule generates a much larger body of optimized C. The schedule itself, however, remains largely a one-off sequence of low-level primitives, even though a few helper functions provide some amortization. The next chapter targets this remaining source-level cost by encapsulating recurring primitive sequences as reusable scheduling-library functions. For both case studies, this refactoring roughly halves the schedule source code while achieving the same performance (Figure 5.7).

Chapter 5

Externalizing Optimization from the Compiler

In this chapter, we unpack the second of the two pillars of Exocompilation: externalizing optimization itself out of the compiler and into user code. Chapter 4 externalized the hardware target but left the vocabulary of optimization fixed: the scheduling primitives of Section 4.2 are compiler built-ins, deliberately as low-level as possible, buying maximum control at the cost of productivity. This chapter amortizes that cost without giving the control back. Instead of growing the primitive set, we let users compose primitives into higher-level scheduling operators that live in libraries, external to the compiler. Vectorization, bounds inference, and configuration hoisting are all libraries in Exo, whereas in Halide or TVM they would be compiler passes. Because these user-level operators are built from safety-checked primitives (Section 3.4), they inherit functional equivalence guarantees by construction. The engineer may always descend to the primitives when an abstraction does not fit.

5.1 Growing a Scheduling Language

The process of optimizing a program consists of repeatedly modifying it into new programs which compute the *same result* more efficiently. Exo is one of a family of user-schedulable languages (USLs)—including Halide, TVM, TACO, and Taichi—that increase programmer productivity when doing such optimization [25,44,47–49]. As we have seen (Section 4.2), USLs reify this optimization process into an explicit *scheduling metaprogram* that transforms an underlying *object program* into a better-performing one. By providing formal or informal guarantees that scheduling transformations preserve the equivalence of the object program, USLs enable performance engineers to focus on optimization strategies rather than painstakingly ensuring correctness [50].

Some non-user-schedulable languages, such as C++ and Python, are designed to "grow" to large projects through the development and composition of libraries built on top of them [51]. These libraries add richer interfaces and functionality to the language without requiring modification of the compiler. Existing USLs are not built this way. Their scheduling operations come as a fixed set, targeted at a particular application domain (e.g., image processing) and/or class of optimization (e.g., loop transformations). Much as scripting

languages like `bash`, `sed`, and `awk` excel at writing small scripts, USLs excel at optimizing small, individual kernels when the task aligns with the language's capabilities—and much like those scripting languages, they are not designed to grow by implementing libraries. Exo, as presented so far, is no exception.

The goal of these fixed sets of scheduling operations is to provide explicit *control* over key optimization choices while abstracting away tedious details. Well-designed USLs must therefore carefully choose a delineation between *automated* optimizations and *user-scheduled* choices exposed as scheduling operations. For instance, the Halide language automates bounds inference, register allocation, and instruction selection (to name a few) but gives users explicit control over loop tiling, fusion, and work vs. locality tradeoffs. When the design works, it shields performance engineers from unnecessary concerns, improving productivity.

In practice, however, it is difficult to design this automation-control boundary perfectly. When the design breaks down, performance engineers have no way to cross the boundary and must drop down to C or assembly code (or even modify the USL compiler) to regain control. These failures can happen either because an optimization is not expressible with the provided controls or because of imperfect automation. For example, prefetching was initially automatic in Halide but later became user-schedulable, and fixed-point vector-instruction selection still requires substantial research and engineering efforts to improve its automation, more than a decade after Halide's initial release [27,28,52]. Targeting novel hardware accelerators introduces additional challenges. Halide has yet to support tensor accelerators (e.g., Gemmini, Tensor Core, TPU) because they pose many nontrivial questions about the separation of control and automation. For example, should Halide automatically handle instruction selection for accelerators as well? If so, to what extent can the existing vector-instruction selection logic be reused? If not, how should Halide expose control over instruction selection to users? Such design failures are inevitable and generate pressure on USL implementers both to extend the scheduling language and to improve automation.

Furthermore, even in an ideal world where USLs provide users ample control and automation in their scheduling operations, users would still face challenges when implementing realistic HPC kernels. Leading HPC libraries, such as OpenBLAS, cuDNN, and MKL, provide a wide range of configurations through their APIs. With existing USLs, users must laboriously write different schedules for each of these configurations—a cross-product of operations, data types, operational parameters, storage formats, and target architectures. This approach quickly becomes unwieldy and cannot scale effectively. Thus, providing users with means to abstract and automate schedule generation is crucial to effectively manage this vast configuration space.

These observations give the requirements for the second pillar. A USL must let users (i) build automation *external* to the compiler and (ii) decide to use more or less automation in different contexts. This chapter presents Exo's scheduling language, which empowers users to safely build *new scheduling operations* from a set of *primitive*, equivalence-preserving operations (Section 4.2). By composing safe primitives, users can write libraries external to the compiler and automate common optimizations, allowing schedule reuse across different kernels while retaining safety guarantees. When users require more control, they can always fall back to using lower-level primitives, satisfying both aforementioned criteria.

Drawing an analogy from the classic perception-control model in robotics and cognition, we conceptualize scheduling languages through the **actions** they take on object code (Section 5.2),

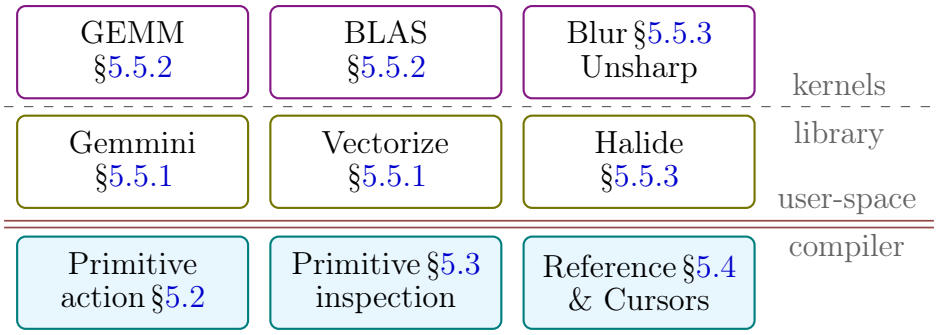


Figure 5.1: Chapter overview.

their **inspection** or perception of that code (Section 5.3), and the **references** pointing to the object code (Section 5.4). Exo's reference mechanism, *Cursors*, binds these three mechanisms together and enables the creation of scheduling libraries within user code (Figure 5.1, middle). With this, we demonstrate libraries which amortize scheduling effort across more than 80 high-performance kernels, including BLAS level 1, 2, GEMM, blur, and unsharp mask (Figure 5.1, top). These libraries reduce total scheduling code by an order of magnitude and deliver performance comparable to or better than MKL, OpenBLAS, BLIS, and Halide on AVX2, AVX-512, and Gemmini platforms.

5.2 Action

The core of any USL is a set of scheduling operations tailored to specific applications, optimization classes, and target programmers. In order to design a USL for growth, the set of scheduling operations should enable users to (i) develop libraries for automation while (ii) retaining the ability to resort to low-level control when necessary. While many USLs' scheduling operation sets are designed to fit specific use cases, Exo's design revolves around composing low-level primitives. Instead of building in scheduling operations closely matched to what we think users will want, we argue that scheduling languages should focus on providing the essential composable building blocks to ultimately be able to achieve those same transformations. We call these *scheduling primitives*. Primitives should be fine-grained, offering control over low-level details and enabling a wide range of transformations through composition. Although composition can always provide abstraction, it cannot provide *finer-grained* control than the underlying primitives themselves. Primitives should also be *safe*: the language implementation should verify that the program remains functionally equivalent after applying a primitive. `divide_loop` and `lift_scope` are examples of such primitives used in the following sections. We implemented a rich set of 46 scheduling primitives for Exo, which is fully described in Section 4.2. These low-level primitives provide users with precise control but reduce the degree of automation provided. In the following sections, we show various ways of composing these scheduling primitives to build automation.

5.2.1 Sequential Composition of Primitives

Recall that schedules in Exo are Python metaprograms where each takes a procedure (such as `gemv`) as input and returns a functionally equivalent but rewritten procedure as output. Throughout this chapter, we frame Exo object code in boxes to distinguish it from Exo *scheduling* code, which appears unframed. Our running optimization target is the matrix-vector multiply procedure `gemv`, shown inset.

```
def gemv(M: size, N: size,
         A: f32[M, N] @DRAM,
         x: f32[N] @DRAM,
         y: f32[M] @DRAM):
  assert M % 8 == 0
  assert N % 8 == 0
  for i in seq(0, M):
    for j in seq(0, N):
      y[i] += A[i, j] * x[j]
```

Suppose we want to write a schedule that tiles a loop nest, a common optimization for improving data locality and one we already applied to GEMM in Section 4.1.1. Schedules can often be expressed as series of rewrites, and these rewrites can be composed sequentially. Each scheduling operation returns a new procedure that can be further scheduled by the next operation. For example, we can tile `gemv` by sequentially composing `divide_loop` and `lift_scope` primitives as follows. For brevity, we abbreviate `gemv` to just `g`.

```
g = divide_loop(g,'i',8,['io','ii'],perfect=True)
g = divide_loop(g,'j',8,['jo','ji'],perfect=True)
g = lift_scope(g,'jo')
```

Since we know that the factor 8 perfectly divides both `M` and `N`, we can omit code for handling tail cases by passing `perfect=True` to `divide_loop`. `lift_scope` takes either a `for` loop or an `if` statement and interchanges it with the surrounding `for` or `if`. This simple composition of primitives yields the tiled `gemv` object code, as shown on the right.

```
for io in seq(0, M/8):
  for jo in seq(0, N/8):
    for ii in seq(0, 8):
      for ji in seq(0, 8):
        y[8*io+ii] += A[...]*x[...]
```

5.2.2 Reusable Schedule Fragments as Functions

Since tiling is a common optimization, many existing USLs provide built-in tile-scheduling operations. However, there is no need to provide tiling as a built-in: function abstraction offers a natural way to encapsulate such scheduling patterns as a reusable, user-level scheduling function.

```
def tile2D(p,               # procedure
           i_lp, j_lp,      # loop names
           i_itrs, j_itrs,  # list of new names
           i_sz, j_sz):     # tile sizes
  p = divide_loop(p,i_lp,i_sz,i_itrs,perfect=True)
  p = divide_loop(p,j_lp,j_sz,j_itrs,perfect=True)
  p = lift_scope(p, j_itrs[0])
  return p
```

Now, instead of calling scheduling primitives directly, we achieve the same transformation with a single call to `tile2D`:

```
g = tile2D(g,'i','j',['io','ii'], ['jo','ji'], 8,8)
```

Functionally, `tile2D` has the same behavior as if it were a built-in scheduling operation. Recall from Section 3.3.2 that an Exo primitive has the type $\mathsf{Op} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc}$, taking a procedure, a cursor, and other arguments and returning a procedure. This design contrasts with the more common method-chaining style used in Halide [25] and TVM [44], where scheduling operators are methods on a program object. Our design enables users to create user-defined scheduling operations with the same type, allowing for seamless integration of primitives and user-defined functions. With the expected type Op, `tile2D` is indistinguishable from a built-in scheduling primitive.

5.2.3 Schedules with Control Flow

Traditional control-flow constructs are useful when developing new scheduling operations and automation. Conditionals enable schedules to perform different actions based on the situation, such as applying different schedules depending on the precision or hardware target. Loops allow us to parameterize the notion of repeating an action. For example, the `tile2D` function could be generalized to work on an arbitrary number of dimensions:

```
def tilenD(p, loops, new_iters, tile_sizes):
    for i, loop in enumerate(loops):
        p = divide_loop(p, loop, tile_sizes[i],
                        new_iters[i], perfect=True)
    for i, _ in enumerate(loops):
        for j in range(0, i):
            p = lift_scope(p, new_iters[i][0])
    return p
```

Since Exo scheduling primitives are guaranteed to raise an error when the transformation breaks functional equivalence, exception-handling mechanisms like `try/except` can thus implement different behaviors based on the safety of scheduling primitives. This approach generalizes `tile2D` to imperfectly divisible loops as shown below.

```
def general_tile2D(...): # same signature as tile2D
  orig_p = p
  try:
    p = tile2D(p, ...) # call to tile2D by default
  except:
    p = divide_loop(orig_p, ..., tail="guard")
    p = divide_loop(p, ..., tail="guard")
    p = lift_scope(p, new_j_iters[0])
    p = lift_scope(p, new_j_iters[0])
  return p
```

This schedule first tries to perfectly tile the code by calling `tile2D`, and if unsafe (i.e. it raises an exception), it defaults to a general tiling schedule by restarting the process using the tail strategy (`tail="guard"`), which adds a guard to avoid out-of-bounds accesses. Since two

calls to `divide_loop` generate two `if` guards, we need to lift the outer loop (`new_j_iters[0]`) twice to achieve the desired tiled loop ordering.

In practice, there are three distinct types of user-facing errors that can occur, and it is beneficial to differentiate between them. The first type of error is the *SchedulingError*, which is raised by the compiler analysis when user code attempts to apply transformations that do not preserve functional equivalence. The second type is the *InvalidCursorError*, which is triggered when navigating to invalid locations (see Section 5.4.2). Lastly, there may be internal compiler errors caused by compiler implementation bugs or other internal issues. It is preferable to specify the type of error being caught, for example, by using `except SchedulingError`, rather than catching all exceptions indiscriminately.

5.2.4 Higher-Order Scheduling Functions

We define operations of type $\widehat{\mathsf{Op}} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc} \times \mathsf{Cursor}$. Any Op, defined in Section 5.2.2, can be lifted to an $\widehat{\mathsf{Op}}$ by $\mathsf{lift}\ op = \lambda(p, c).(op(p), c)$. This allows us to define higher-order scheduling combinators that take $\widehat{\mathsf{Op}}$s as arguments and produce $\widehat{\mathsf{Op}}$s as output.

```
def seq(*ops):
  def func(p, c, *args):
    for op in ops:
      p,c = op(p,c,*args)
    return p, c
  return func

def try_else(op, opelse):
  def func(p,c,*args):
    try:
      p,c = op(p,c,*args)
    except:
      p,c = opelse(p,c,*args)
    return p, c
  return func
```

```
def repeat(op):
  def func(p, c, *args):
    try:
      while True:
        p,c = op(p,c,*args)
    except:
      return p, c
  return func

def reduce(op, top):
  def func(p,cur,*args):
    for c in top(cur):
      p,c = op(p,c,*args)
    return p, c
  return func
```

Higher-order scheduling functions are implemented using these combinators. For example, `seq(lift_alloc, lift_alloc)` is a scheduling function that lifts an allocation twice, while `repeat(lift_alloc)` lifts an allocation as much as possible. Both functions retain the type $\widehat{\mathsf{Op}}$ and so can be further combined as expected.

5.3 Inspection

Type reflection, also known as introspection for homogeneous systems and inspection for heterogeneous systems, is a metaprogramming feature that enables programs to dynamically examine object code through built-in functions (e.g., `typeOf` in Haskell) or pattern matching. To the best of our knowledge, no existing USL exposes inspection of object code to users, making operations such as "unroll all loops with bounds less than 5" inexpressible. We

believe that inspection is essential for effective metaprogramming in user-extensible USLs. For example, when creating a vectorizer library, identifying properties such as reductions, buffer precisions, and loop-invariant vectors in the object code is crucial for determining parallelization strategies and choosing vector instructions. Figure 5.4 and Section 5.5.1 demonstrate how the presence of FMA instructions affects optimal staging, necessitating inspection to identify them.

Exo provides type reflection through cursors, allowing users to examine standard AST properties like variable names, literal expression values, and annotations (e.g., memory spaces and precisions) at scheduling time. Cursors also support local AST navigation, accessing loop bounds (`loop.hi()`) and bodies (`loop.body()`). More complex inspection operations can be built by combining inspection and navigation primitives. For instance, bounds inference, which determines all possible index accesses to an array within a given scope, is provided as a *built-in* feature in Halide. In contrast, Exo provides users with facilities to implement bounds inference as a new inspection operator *external* to the language, which we can then use as a building block of Exo's Halide library (Section 5.5.3). As an example, consider inferring access bounds for `arr` within the `io` loop:

```
for io in seq(0, N / 32):
    # arr is accessed within [32 * io : 32 * io + 34]
    for ii in seq(0, 32):
        x = arr[32 * io + ii] + arr[32 * io + ii + 1]
                              + arr[32 * io + ii + 2]
```

Determining the bounds for the array `arr` can be reduced to unioning bounds for each array access. Each index expression is an affine expression consisting of constants and variables which may be either free or bound in the scope. In the index expression `32 * io + ii + 1`, `ii` is a bound variable, and `io` is a free variable. Knowing that $0 \leq$ `ii` < 32, the index expression spans the window `[32 * io + 1 : 32 * io + 33]`. Our implementation combines primitive cursor inspections, which query the loop-bounds expressions, with ordinary Python code which maintains the environment of free/bound variables and unions the inspected bounds.

5.4 Reference

5.4.1 References in USLs

Nominal vs. relative reference. Suppose we want to specify the loop nest to which we want to apply `tile2D`. Given a pattern (Figure 5.2, left), there is no guarantee that this reference is unique (Figure 5.2, program 1) or that we are referring to anything at all (Figure 5.2, program 2), because resolving references depends on the context, or *frame of reference*. Therefore, USLs must address questions of ambiguity and invalidity for object-program references. Halide, for example, opts for a *nominal* reference system that eliminates ambiguity by design. In Halide, each statement is uniquely identified by the buffer it writes to (e.g., `blur_x`; see Figure 5.12 for sample code), and each loop within a loop nest has a unique iterator variable (`x`, `y`, and `xi`). This allows for globally unambiguous reference of loops using the pair (`blur_x`, `y`).

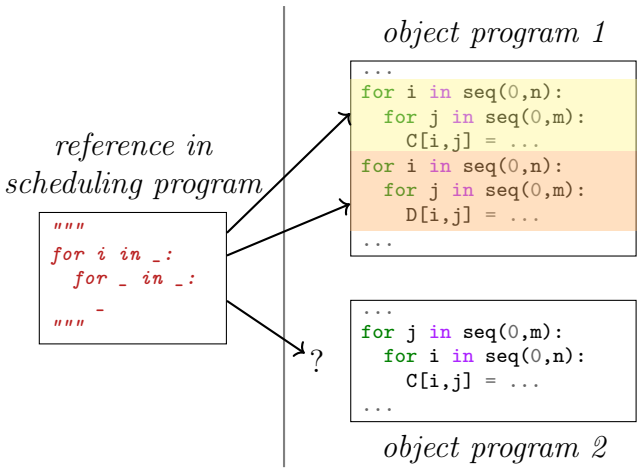


Figure 5.2: Schedules need mechanisms to *refer* to the object code they wish to modify. The meaning of a given reference expression is often dependent on context (*frame of reference*) and may refer to zero, one, or many parts of the program.

Global nominal references prevent ambiguity but introduce significant limitations, requiring users to manage *globally distinct* names for each object code fragment. Without context-dependent references, it is impossible to specify transformations such as "parallelize the second most outer loop" or "unroll three stages back". Instead, schedules would be hard-coded with nominal references for specific object code. We propose that *relative* references that depend on the frame of reference are essential for building growable scheduling languages. The "nominal" and "relative" reference distinction is analogous to "proper nouns" and "noun phrases" – proper nouns must refer by name (e.g., Boston, NY), while noun phrases enable context-based references (e.g., cities on the East Coast), potentially matching zero, one, or many objects. If we change the context from the US to Canada, relative references remain applicable (though they refer to different objects), while nominal references do not.

Single vs. multiple references. In contrast to Halide, ELEVATE [45,46] allows users to systematically define traversal strategies to disambiguate references with respect to a tree traversal order. For instance, `topDown` applies a rewrite at the *first* successful application, while `tryAll` applies the rewrite wherever possible. However, ELEVATE only allows having a single reference at a time. This single-frame limitation in ELEVATE can be inflexible and inconvenient because users always have to navigate relative to a single point of reference. Halide, on the other hand, allows multiple references to coexist. This means that users can refer to multiple specific points in the code simultaneously, using their unique names.

Stable vs. one-time references. Scheduling is a programming process that evolves over time, meaning that frames of reference have both temporal and spatial dimensions. Suppose there are two references to the object code: one pointing to the loop with iterator `i` (Figure 5.3 light blue) and the other to the loop with iterator `j` (Figure 5.3 orange). If we tile loops `i` and `j`, what happens to the second reference? Has the object it's pointing to changed, or has it ceased to exist?

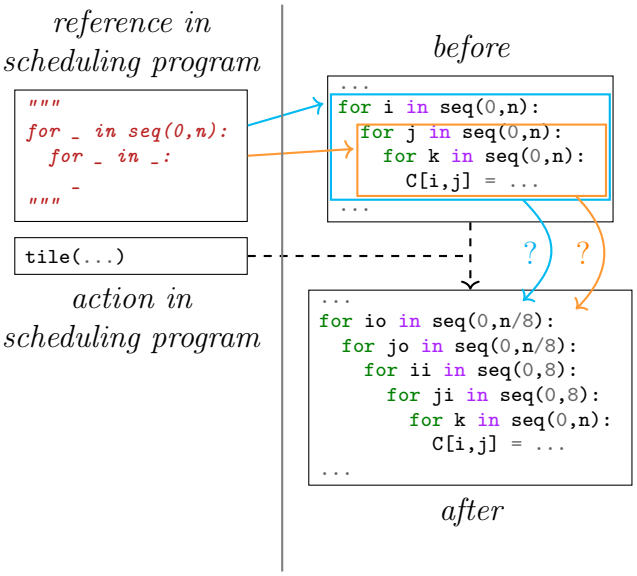
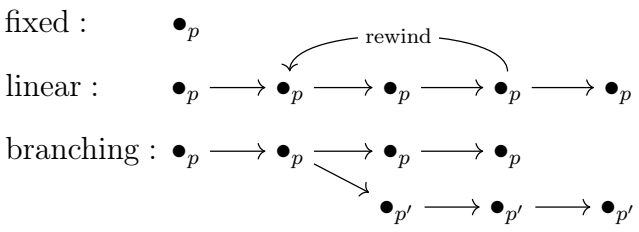


Figure 5.3: The process of transforming code raises the question: where should references to the original code point after a transformation is applied, or should they be invalidated?

```
for i in seq(0,n):
  for j in seq(0,n):
    C[i,j] = ...
for i in seq(0,n):
  for j in seq(0,n):
    D[i,j] = ...
```

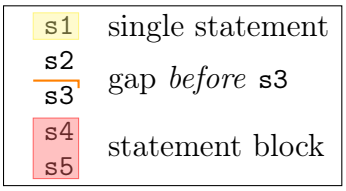
Now consider the same schedule applied to a different program (inset left). In this case, there is no complication because the loop nest to which the second cursor points is unaffected by tiling the first.

These examples demonstrate how references are always implicitly made at a specific moment in the scheduling process. When we resolve a reference, we are essentially asking, "What does this reference denote, *right now*?" To answer this, we identify another taxonomy for USLs: *stable* references that can be reused after applying an action and *one-time* references that get invalidated after an action. Nominal references, like those in Halide, are effective at providing stable references because they are globally unambiguous. In contrast, ELEVATE's references are one-time, which is similar to other transformative metaprogramming systems like jQuery. Pattern matching, as used in these systems, is a brittle mechanism for the scheduling process. A pattern that existed at one point in the scheduling process might not exist or may point to completely different object code after applying an action (e.g., how the `for i` loop ceases to exist after tiling in Figure 5.3). Furthermore, pattern matching requires exact knowledge of the code structure and its potential transformations, hindering the encapsulation of scheduling operations.

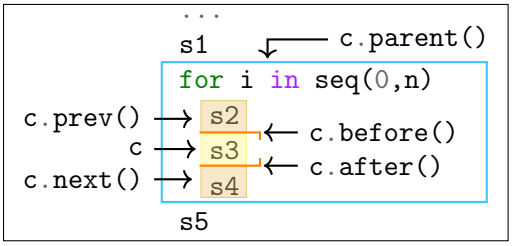


Scheduling time models in USLs The choice of aforementioned reference designs affects the fundamental design of USLs, and consequently affects the scheduling time models for USLs: (i) The *fixed* time model (inset top) uses a single, static reference frame, largely achieved by Halide's *multiple, stable, nominal* references. (ii) The *linear* time model (inset middle) advances the reference frame for each action and allows it to "rewind" after encountering an error. ELEVATE's *single, one-time, relative* reference supports this model. (iii) The *branching* time model (inset bottom), adopted by Exo's *multiple, stable, relative references*, treats time as branching into parallel versions of the procedure, with cursors living at specific versions and scheduling creating different cursors for each branch. The branching time model encompasses the other time models, as we will demonstrate by recreating Halide and ELEVATE-style references in Section 5.5.3.

5.4.2 Cursors

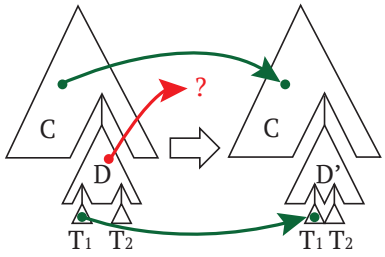


Exo's reference mechanism, embodied by *Cursors*, maintains *multiple, stable, relative* references. Cursors, inspired by the blinking vertical bar in text editors, allow users to select and refer to parts of the code such as expressions, blocks of statements, and gaps between statements (inset left).



Multiple, relative cursors require modulating the spatial reference frame (navigation) and querying cursors in-context. Cursors enable spatial navigation within a procedure (i.e. "after _," "before _," "parent of _," etc.) to proximate locations (see inset above). Moreover, instead of invoking `proc.find(...)`, we can invoke `cursor.find(...)` to restrict our pattern match to the sub-AST identified by the `cursor`. *InvalidCursorError* is raised when navigation is invalid. For instance, `c.parent()` is invalid when `c` is already pointing to the top-level statement.

Forwarding *Cursor forwarding* is the mechanism by which Exo supports stable references. Consider a generic action which rewrites the inset left AST to the inset right AST. We can decompose the AST of the object code as $P = C[S]$, where C is the unmodified subtree, and S is the modified subtree. Furthermore, we can usually identify some *invariant* subtrees $T_1, \ldots, T_k$ of the modified subtree S which are not changed by the action. Therefore, we have the further decomposition $P = C[S] = C[D[T_1, \ldots, T_n]]$. Now, we can describe the procedure after the action as $P' = C[D'[T_1, \ldots, T_n]]$ — all changes are isolated to the transformation of $D \to D'$. Given the decomposition of a program into an outer context C, a subtree D that is being transformed, and subtrees $T_1, \ldots, T_n$ that replace D, any cursor pointing to a statement in C or T_i can be unambiguously forwarded after the action is completed. Cursors to gaps and statement blocks can be unambiguously forwarded if attached to statements or within C or T_i. For composite actions, if a cursor-forwarding policy



handles the unambiguous primitive-action cases correctly, it will correctly forward cursors to any AST part untouched by all scheduling actions.

The ambiguous cases occur when the cursor points to a statement in D. There are two high-level design decisions for handling such ambiguous cases: (i) invalidate any ambiguous cursor, which ensures unambiguous behavior but makes scheduling programs more laborious to write; or (ii) attempt to produce a valid cursor whenever possible, which maximizes convenience but may lead to unintuitive behavior. Exo has chosen the second approach and defines heuristic forwarding rules for each scheduling primitive. This can lead to ambiguous forwarding behavior, especially when calling scheduling library functions that compose many forwarding effects. However, since forwarding is deterministic even in ambiguous cases, we find that simply printing cursors after the function call is usually sufficient to understand the behavior in practice.

When a primitive action takes in a procedure and input cursors, Exo will *implicitly* forward the cursors to the procedure's reference frame, since cursors should, by definition, always point to the procedure. Therefore, `expand_dim(p, c, ...)` becomes a shorthand for `expand_dim(p, p.forward(c), ...)`. However, navigation and forwarding do not commute in general. If `c.next()` is passed to a primitive action, the implicit forwarding behavior is equivalent to `p.forward(c.next())`, not `p.forward(c).next()`.

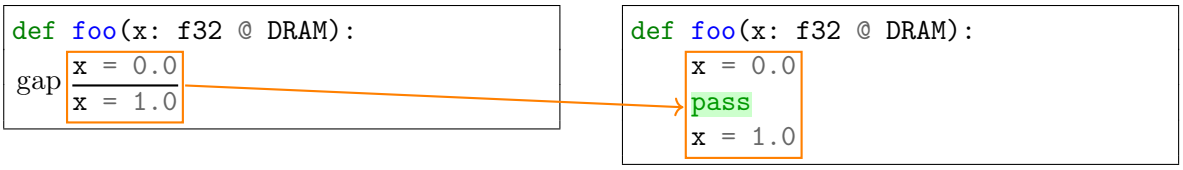
Implementation Internally, a cursor stores two values: a reference to the procedure it's pointing at (i.e. a *time* coordinate) and a path defining its relative location inside that procedure's AST (a *spatial* coordinate). The path describes navigation in an AST as a downward traversal. In an AST, all children are labeled (e.g. the *rhs* of a binary operation), and each is either a node or list of nodes (e.g. the *"body"* of a for loop). Thus, each downward step in the AST may be represented as a label-index pair, where the index is null if the child is not a list.

For example, in this code block, the `y` variable has path $(body, 1)$, $(body, 0)$, $(rhs, \emptyset)$, $(lhs, \emptyset)$. Traverse to the second statement of the procedure body $(body, 1)$, the first statement of the loop body $(body, 0)$, the right-hand side of the assignment $(rhs, \emptyset)$, and then the left-hand side of the addition $(lhs, \emptyset)$. Spatial navigation operations such as `.parent()`, `.rhs()`, `.body()[idx]`, or `.next()` are straightforward to implement by modifying this path representation. Cursors to gaps are relative to the statement nodes, and a cursor to a statement block simply stores a *range* at the final path index, instead of a single value.

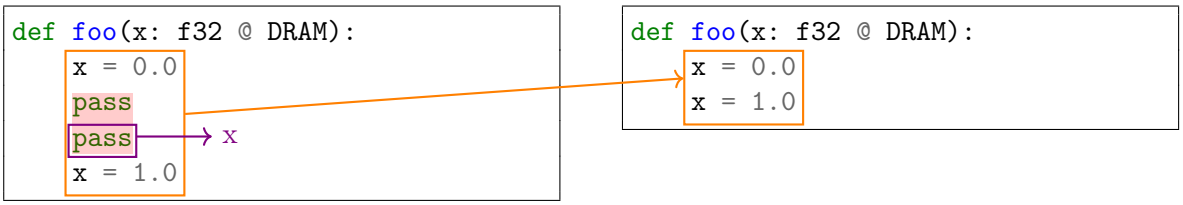
```
x: i8
for j in seq(0, 2):
  x = y + z
```

Whenever a scheduling action is applied to procedure p to produce a new procedure p', the Exo runtime records this provenance along with an internal forwarding function. This internal forwarding function defines how to forward from a cursor c into p to a cursor c' into p'. This is done by decomposing the effect of every primitive action into atomic AST edits (insert, delete, replace, move, and wrap; as shown below), each with its canonical forwarding function.

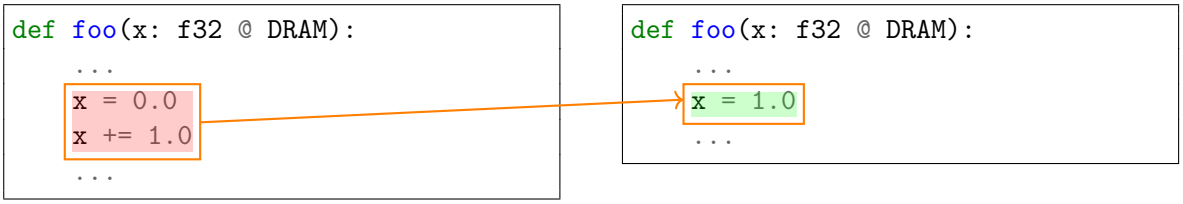
Insertion Inserting an IR fragment into the AST preserves existing paths. The forwarding function adjusts paths through the insertion point by incrementing preexisting paths at the appropriate tree level. The orange block cursor illustrates how existing cursors are forwarded when a new `pass` is inserted into a gap.



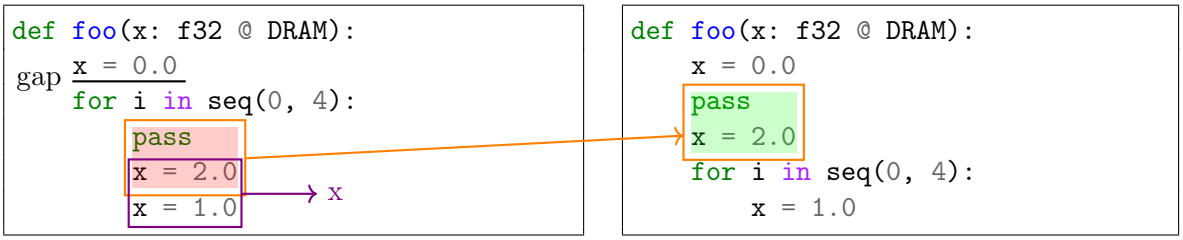
Deletion Deleting a subtree from the AST invalidates paths within the subtree (shown as a violet cursor below), while paths outside remain valid (orange cursor). The forwarding function decrements preexisting paths past the deletion point at the appropriate tree level.



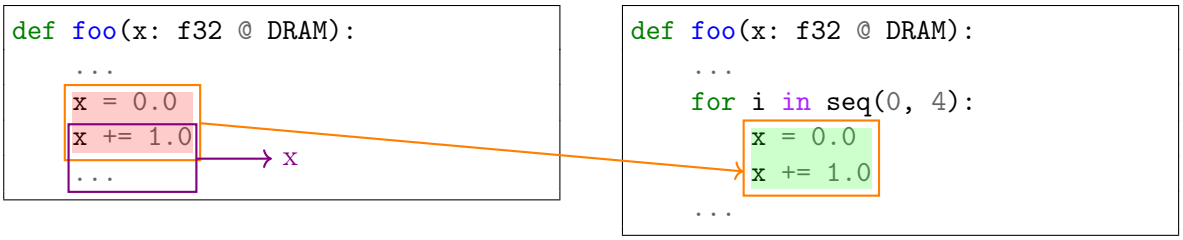
Replacement Replacement supports higher-level operations like algebraic simplification by replacing an existing subtree (two statements with a red background below) with a new one (`x = 1.0`). The forwarding behavior is similar to inserting the new subtree and deleting the old, except the unique path to the old subtree remains valid.



Movement Moving a subtree (orange block cursor) to a gap within the AST preserves its node identity but invalidates cursors across the subtree boundary (violet block cursor). The forwarding function maps paths to their natural correspondences, handling cases of moving to higher or lower levels in the AST.



Wrapping Wrapping an existing subtree (orange cursor) with a one-hole IR fragment (e.g. `for i in seq(0, 4): _`, where the body of the for loop is the hole) is equivalent to insertion with movement for blocks but not for expressions. Consequently, we separate this function as its own atomic edit. Paths into the wrapped subtree are adjusted by inserting a step at the appropriate level to navigate into the wrapping fragment. Paths through the wrapping point are decremented at the level by the length of the replaced subtree minus one.



The internal forwarding function for a primitive action is the sequential composition of forwarding through its constituent atomic edits. For example, suppose `p` is the result of applying a primitive action to cursor `c`, and this primitive action was comprised of n atomic edits. When `p.forward(c)` is called, Exo composes the forwarding functions (f_1, f_2, ..., f_n) for each primitive action between `p_0 = c.proc()` and `p_n = p` (the output procedure). This composition produces a single function $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$ that maps `c_0` from its location in the original procedure to the corresponding location in p_n. Therefore the forwarded cursor can be obtained by computing $f(c)$. If any forwarding function returns an invalid cursor, invalidity is maintained by each subsequent forwarding function.

5.5 Building Scheduling Libraries

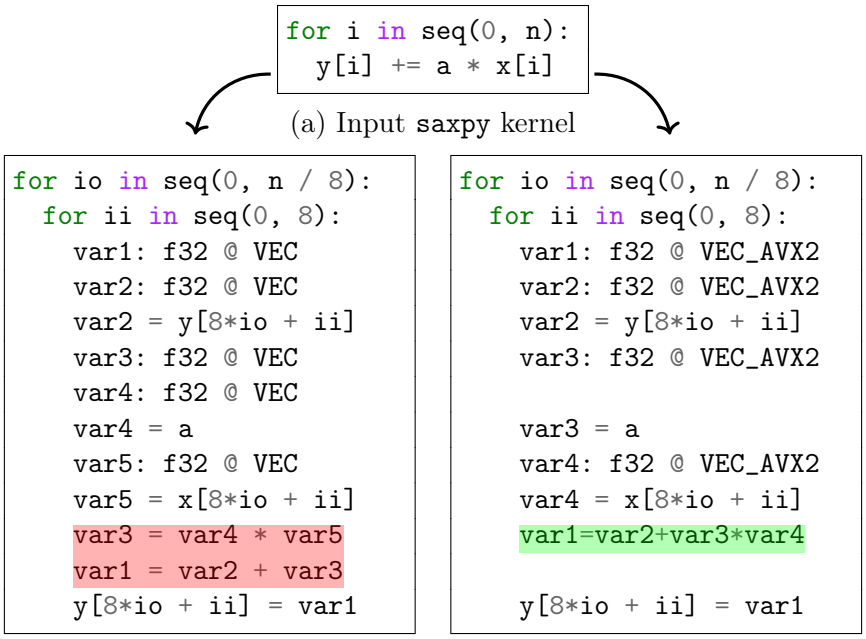
5.5.1 Target-Specific Functions

Exo's action, inspection, and reference empower users to create scheduling libraries external to the compiler. This section shows how Exo enables users to automate hardware architecture-specific optimization strategies while maintaining control over low-level performance considerations.

Vector Architectures

We define a new `vectorize` scheduling operator in user code, which is parameterized over vector width, precision, memory type, and vector instructions, so it can be instantiated for many different vector machines. By leveraging the safety guarantees of the `fission` primitive, which checks for loop-carried dependencies, we eliminate the need for separate dependency analyses, allowing for the concise implementation of transforming loop operations into their SIMD equivalents. Programmers can customize the vectorization process by scheduling deviations as a prologue (e.g., common-subexpression elimination) or epilogue (e.g., loop-invariant code motion) or by incorporating hooks to modify behavior in certain cases (e.g., detecting a fused multiply-add, or FMA).

```
def vectorize(p, loop, vw, precision,
                  mem_type, instrs, rules=[]):
    p = divide_loop(p, loop, vw, tail="cut")
    p = parallelize_reductions(p, loop, ...)
    inner = p.forward(loop).body()[0]
    p = stage_compute(p, inner, ..., rules)
    p = fission_into_singles(p, inner)
    return replace_all_stmts(p, instrs)
```



(a) Input `saxpy` kernel

(b) Staging without FMA

(c) Staging with FMA

Figure 5.4: Code transformations for staging the computation (step 3 of `vectorize`), depending on the presence of FMAs.

The signature of `vectorize` follows the type $\mathsf{Op} = \mathsf{Proc} \times \mathsf{Cursor} \times \ldots \to \mathsf{Proc}$ (Section 5.2.2), which takes a procedure and a cursor pointing to the loop to be vectorized. The implementation follows these steps: (1) Expose parallelism by dividing the loop. (2) Parallelize all reductions in the loop. (3) Get a cursor `inner` to the innermost loop and stage the computation into temporary assign statements, representing unary (e.g., load) or binary (e.g., addition) operations. Figure 5.4 shows an example object code of this step, depicting the starting loop in Figure 5.4(a) and applying the default staging in Figure 5.4(b). Users can overwrite the default automation when necessary. For instance, Figure 5.4(c) shows more efficient staging when an FMA instruction is available. Staging behavior can be customized by `rules`, which scan the expression and specify which subexpressions in the subtree should be recursively staged. (4) Finally, we fission between the statements to generate a loop-based representation of the SIMD operations, which are replaced with the equivalent hardware target instructions.

Gemmini

Gemmini is a machine-learning accelerator developed at UC Berkeley that supports matrix-matrix multiplication (tensor) operations [39]. The GEMM optimization strategy for tensor operations inherently depends on the specificity of the accelerator, and Exo allows users to implement such accelerator-specific optimization passes in a library, which is fully explained in Appendix A.1. This section focuses on one of the Gemmini library functions, configuration hoisting, which is a unique and specific requirement of stateful accelerators.

Gemmini programmers must program configuration registers to influence the behavior

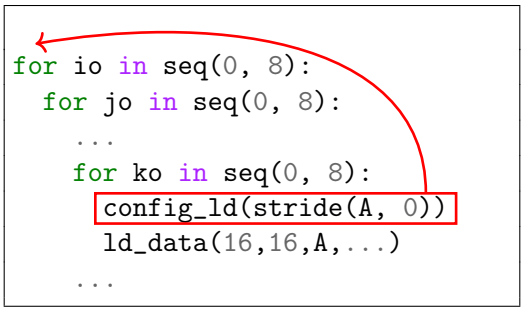


Figure 5.5: Gemmini object code. The `ld_data` instruction, which reads from the configuration state, is prepended with the `config_ld` instruction to set the stride. The goal in this object-code example is to hoist an expensive `config_ld` instruction out of the loops.

```
while True:
  try:
    try:
      while True:
        try:
          p = reorder_stmts(p, ...)
        except:
          raise Exception("...")
    except:
      p = fission(p, ...)
      p = remove_loop(p, ...)
  except:
    break
```

(a) Schedule for hoisting a single statement to the top of an object program.

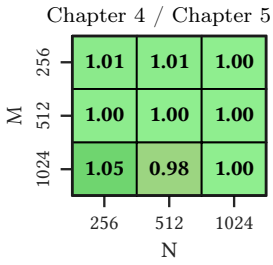
```
repeat(
  try_else(
    seq(
      fission_after,
      remove_parent_loop
    ),
    reorder_before
  )
)(p, c)
```

(b) Alternatively, the same schedule as in Figure 5.6(a) can be implemented using higher-order functions defined in Section 5.2.4. Section 5.5.3 defines the scheduling operations combined with relative references.

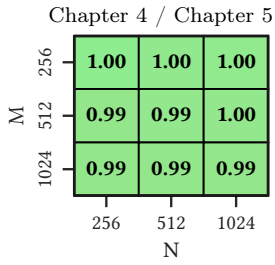
Figure 5.6: Two possible schedules for hoisting a Gemmini configuration instruction.

of instructions, such as stride, activation, and quantization. The simplest approach for a compiler is to naively reset this configuration state before every operation, resulting in patterns like `config_ld` followed by `ld_data` (Figure 5.5). However, these configuration instructions must be hoisted outside the inner loops to avoid expensive, redundant reissuing. Figures 5.6(a) and 5.6(b) show two schedules that hoist a single statement as much as possible: one using Python's loop and exception constructs, and the other using higher-order scheduling operations defined in Section 5.2.4. The statement-hoisting schedule repeatedly attempts to reorder the statement to the beginning of the loop, fission the loop, and remove the enclosing loop. This schedule is then used to implement a function that hoists all Gemmini configuration statements.

We implemented matrix-matrix multiplication using the user-level libraries presented in this section and compared its performance against the Exo implementation from Chapter 4 (without Cursors). As shown in Figure 5.7(a), the Cursors-based implementation achieved a



(a) Gemmini Runtime



(b) AVX-512 Runtime

	Gemmini	AVX-512
Library ref	313	>1,690
Exo Chapter 4 schedule	61	180
Exo Chapter 5 schedule	29	97

(c) Lines-of-code comparison. Exo schedules are compared against state-of-the-art reference implementations (Gemmini standard library and OpenBLAS).

Figure 5.7: Comparison of the Cursor-based matrix-multiplication schedules from Chapter 5 with the schedules from Chapter 4. Higher is better in (a) and (b); K is set to 512.

runtime similar to that of Chapter 4's implementation (which is itself already 3.5× faster than the original Gemmini library [29]) while requiring fewer lines of code. Performance was measured on Chipyard version 1.9.1 running the FireSim simulator on an FPGA, using Gemmini version 0.7.1 [53,54].

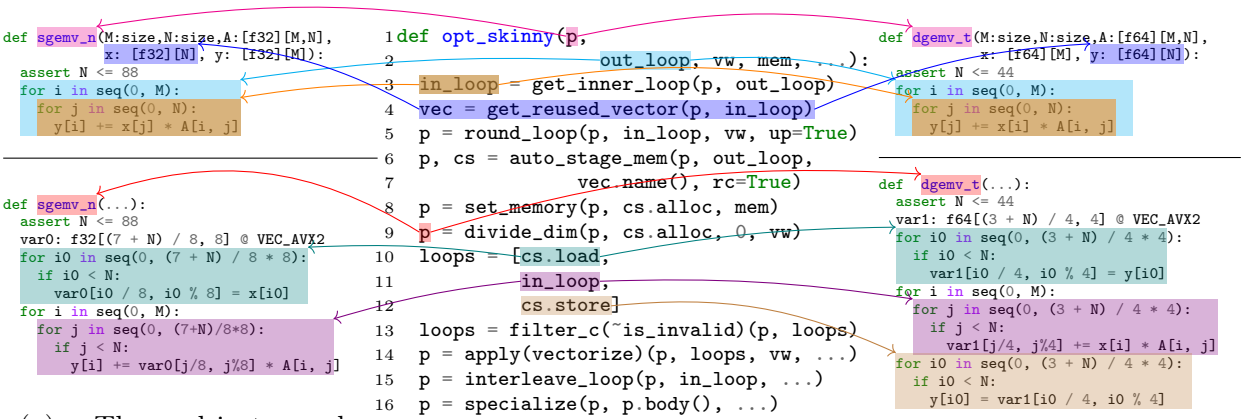
5.5.2 Linear-Algebra Library

Unlike existing USLs, Exo allows users to encapsulate application-specific shared scheduling strategies in libraries. This greatly simplifies the optimization of HPC kernels over various configurations—a cross-product of operations, data types, operational parameters, storage formats, and target architectures. We built a linear-algebra scheduling library and used it to optimize most of the kernels in BLAS levels 1 and 2, as well as GEMM.

BLAS Level 1 Operations

An $O(n)$ BLAS operation can be implemented using a single loop. The optimization process involves two main strategies. First, SIMD vector parallelism can be exploited by interleaving loop iterations (modulo the vector width) and autovectorizing the resulting code. Second, the amount of work can be reduced through common-subexpression elimination (CSE) and loop-invariant code motion (LICM). Increased register pressure is generally not a concern due to the low arithmetic intensity of level 1 operations. For reductions (dot, asum), computing partial sums in each vector lane exposes data parallelism for vectorization. The degree of loop interleaving is tuned to balance performance gains with register pressure. In practice, CSE and LICM do not typically cause register-spilling issues or limit the amount of beneficial loop interleaving that can be performed for BLAS level 1 operations.

We developed a scheduling operator (`optimize_level_1`) that optimizes all of the BLAS level 1 kernels, including asum, axpy, dot, rot, rotm, scale, swap, copy, and dsdot, for a total of 24 kernel variants. However, due to limitations in Exo's object code, which lacks support for value-dependent control, we were unable to support the nrm2 and iamax kernels. Appendix A.3.1 offers the implementation of `optimize_level_1` and performance graphs comparing our results against OpenBLAS, MKL, and BLIS using AVX2 and AVX-512 instructions. All results presented in this chapter were collected on an AWS m7i.xlarge



(a) The object code of single-precision non-transposed matrix-vector multiplication.

(b) The shared schedule.

(c) The object code of double-precision transposed matrix-vector multiplication.

Figure 5.8: 5.8(b) encapsulates BLAS level 2, skinny matrix-specific optimization in a single library function, used to optimize both transpose (5.8(c)) and non-transpose (5.8(a)) gemv and ger, amortizing the cost of writing schedules across kernel variants.

instance equipped with an Intel(R) Xeon(R) Platinum 8488C processor running at 3.2GHz. Our optimized kernels match the performance of these libraries across the board.

BLAS Level 2 Operations

We show how optimization for $O(n^2)$ BLAS level 2 operations can be shared across different kernels, precisions, and operational parameters.

General Matrix The common optimization approach for general matrices is turning the loop into a batched program of multiple dot products, allowing vector reuse and reducing memory loads (unroll-and-jam). In contrast to normal loop unrolling, this transformation will jam c iterations from an outer loop to an inner loop and then unroll the iterations (it will also do normal unrolling for any statements around the inner loop). The following is the result of applying this to `sgemv_n` (see Figure 5.8(a) top for the starting object code) with $c = 2$:

```
def sgemv_n(...):
    for io in seq(0, M / 2):
        for j in seq(0, N): # <--- main inner loop
            y[0+2*io] += x[j] * A[0+2*io, j]
            y[1+2*io] += x[j] * A[1+2*io, j]
    for ii in seq(0, M % 2):
        for j in seq(0, N): # <--- tail inner loop
            y[ii+M/2*2] += x[j] * A[ii+M/2*2, j]
```

In the first inner loop, `x[j]` is loaded from memory twice. To optimize this, we can load `x[j]` once and store it in a register for both updates, which is equivalent to applying

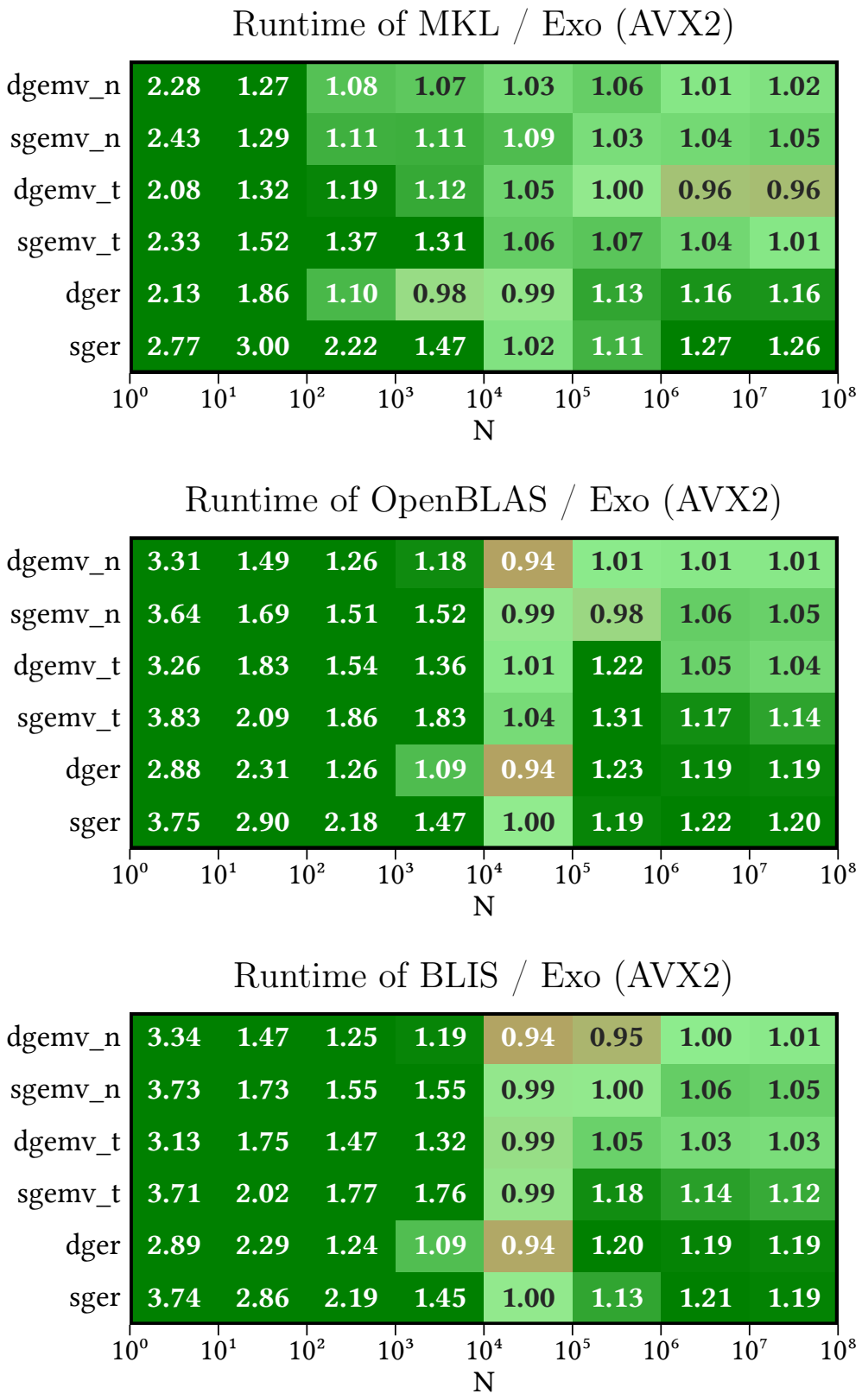


Figure 5.9: Performance evaluation of Exo against Intel MKL, OpenBLAS, and BLIS on gemv and ger variants. `_n` suffix means non-transpose, and `_t` means transpose.

CSE. Our implementation treats the resulting inner loop as a level-1 problem and calls `optimize_level_1` from Section 5.5.2.

Triangular Matrix In triangular matrices, the inner loop bounds depend on the outer loop variable, preventing direct loop reordering for data reuse. To address this, the inner loop bound is rounded up or down to a multiple of the blocking factor, removing the dependence on the outer loop variable. This allows the loops to be reordered using unroll-and-jam to enable data-reuse optimizations. The exact rounding strategy depends on whether the diagonal is included and if the target machine supports predicated vector memory operations.

The combined scheduling code for general and triangular matrices, along with their performance graphs, can be found in Appendix A.3.2. We optimized 50 kernel variants, supporting most BLAS level-2 operations (excluding banded operations and packed-triangular formats) across all configurations. This includes operations (gemv, ger, symv, syr, syr2, trmv, trsv), precisions (float (`s`), double (`d`)), operational parameters (transpose (`t`), non-transpose (`n`), lower (`l`), upper (`u`) triangular, unit (`u`), non-unit (`n`)), and target architectures (AVX2, AVX-512). Our implementation achieved competitive performance with OpenBLAS, MKL, and BLIS on both AVX2 and AVX-512 platforms.

Skinny Matrix A more efficient strategy for short vectors and skinny matrices is to load the entire vector into registers before the quadratic math loop. Since the vector is stored in registers, the vector length must be statically determined as the closest multiple of the register width, and a specialized case must be generated for each multiple.

Figure 5.8 shows the implementation of this shared schedule (5.8(b)) and its application on two programs (5.8(a) top) and (5.8(c) top). We developed this shared schedule for CPUs with vector extensions that support predicated vector loads/stores (e.g. x86 AVX2 and AVX-512). The schedule follows four steps: (1) Inspect the program to obtain cursors to the inner loop (line 3) and the reused vector (line 4). (2) Stage the reused vector into registers by rounding up the loop-iteration bound to a multiple of the vector width (line 5) and staging the vector around the doubly nested loops (line 6). The bottom of Figures 5.8(a) and 5.8(c) show the resulting object codes after this step. (3) Optimize the generated loops (load, inner math loop, and store) by applying vectorization (line 14) and interleaving the inner loop to increase ILP. (4) Specialize the program to ensure static information about the number of registers and their accesses is known for efficient code generation.

We compared the performance of the generated programs against Intel's MKL, OpenBLAS, and BLIS on problems with $N = 40$ and M having powers of 2 and 3. Figure 5.9 shows the results, where each heatmap cell represents the geometric mean over problems in the respective x-axis bucket. The evaluated programs also handle scaling, which was omitted for brevity in Figure 5.8.

Matrix-Matrix multiply

The leading gemm literature, such as GotoBLAS [36] and BLIS [37], suggests building a general high-performance GEMM out of highly performant register-level tiles (micro-kernels). In addition to the main micro-kernel that fully utilizes registers and L1 cache size, we also need to generate smaller micro-kernels of size $m \times 16n$, where either $m = 6$ and $n \leq 4$, or

	BLAS-lib	std-lib	ins-lib	Level 1	Level 2	sgemm
Obj.	0	0	0	90	139	11
Schd.	241	1089	449	18	34	97
Gen. C	N/A	N/A	N/A	3196	11040	521

(a) Breakdown of the lines-of-code distribution in our library and kernels. std-lib refers to higher-order and target-specific functions like `vectorize`, BLAS-lib represents BLAS-specific optimizations such as `opt_skinny` and `optimize_level_1`, and ins-lib refers to the inspection library functions.

kernel	rewrites	kernel	rewrites	kernel	rewrites
asum	1760	axpy	1826	dot	1208
rot	1716	rotm	3372	scal	1308
swap	1484	gemv	3996	ger	2848
symv	3310	syr2	7832	syr	4046
trmv	4828	trsv	5370	sgemm	756

(b) Number of primitive rewrites required for optimizing each kernel. This includes the generation of all configurations, such as single and double precision, transpose and non-transpose operations, stride-1 and stride-any cases, and other variations (e.g., upper or lower triangular, and unit or non-unit options).

Figure 5.10: (a) Lines of code and (b) the number of primitive rewrites for library functions and kernels from Section 5.5.2. sgemm refers to the AVX-512 sgemm kernel.

$m \leq 6$ and $n = 4$. Generating these micro-kernels by hand is laborious and typically written in assembly in the aforementioned literature.

We implemented a single scheduling function, `gen_ukernel`, for all the micro-kernel generation, parameterized by precision, vector width, vector predicates, and the micro-kernel sizes N and M. The `gen_ukernel` function stages memory for registers, applies `vectorize` (see Section 5.5.1), and inserts calls to AVX-512 intrinsics. This function is used to generate all the micro-kernels in the main, right, and bottom panels (see Appendix A.2).

Figure 5.10 shows the lines of code in the scheduling library and the number of primitive rewrites performed to optimize each kernel. Exo reduces scheduling complexity by allowing users to implement new scheduling operators and reuse them for the optimization of multiple kernels rather than hand-writing primitives for each kernel one-by-one. We show the number of primitive rewrites required for scheduling kernels as it reflects what users would directly write in Chapter 4 Exo, without Cursors. This may slightly overstate the difference compared to an equivalent best-effort Chapter 4 Exo schedule but accurately reflects the order-of-magnitude improvement we observe with Cursors. We also refactored the Chapter 4 AVX-512 GEMM schedule using Cursors. Figure 5.7(b) shows that the Chapter 5 version preserves the Chapter 4 runtime, which is comparable to MKL, while reducing the schedule from 180 to 97 lines (Figure 5.7(c)).

5.5.3 Reproducing Existing Scheduling Languages

Existing USLs hardcode actions and referencing schemes, limiting users to specific design choices. For instance, ELEVATE limits users to a linear time model and actions controlled

by traversal functions. Halide limits users to a well-chosen set of scheduling primitives and a fixed-time, nominal referencing scheme. In contrast, Exo allows users to choose referencing schemes and the level of automation in scheduling operators. By reproducing ELEVATE-style traversal functions, Halide's scheduling operations, and their respective referencing schemes within the user-level library code, we demonstrate that Exo can recreate even complex actions such as `compute_at` and `store_at` and show how Exo's branching time model encompasses the linear and fixed time reference models employed by other USLs. Our approach gives users not only the choice of operators and references but also enables interoperability between different user-defined operators and referencing schemes within the same program as needed.

Reproducing ELEVATE

Traversal functions A traversal function specifies a traversal order starting from a cursor and has type $\mathsf{Top} = \mathsf{Cursor} \to \mathsf{Stream}[\mathsf{Cursor}]$. In rewrite-based USLs, users must explicitly control the order in which scheduling actions are applied to the object code. This is because the sequence of actions does not commute in general, even when the same scheduling primitive is repeatedly applied, similar to the phase-ordering problem in general compilers. For example, suppose there is a three-nested loop of `i, j, k`. Applying `lift_scope` on `j` and then `k` will yield a loop nest of `j, k, i`, while applying it on `k` and then j will yield a loop nest of `i, j, k`. Therefore, it is necessary to define rules for different traversal strategies.

Strategy languages [55,56] and USLs like ELEVATE separate traversal strategies from rewrite rules. By leveraging navigation and introspection, we can support traversal strategies over cursors, such as the postorder traversal (`lrn`).

```
def lrn(c):
  for c in c.body():
    if isinstance(c,
       (ForCursor,IfCursor)):
      yield from lrn(c)
    yield c
```

Linear time Moreover, we can reproduce a functional-style linear-time referencing model using higher-order functions. Cursors were implicitly forwarded and returned in all the higher-order scheduling functions we discussed earlier (see Section 5.2.4). However, we can also define higher-order scheduling functions that return other cursors by using the `nav` function, which navigates the frame of reference:

```
def nav(move):
  return lambda p,c: p, move(p.forward(c))
def savec(op):
  return lambda p,c: op(p,c)[0], c
```

Let `move` : $\mathsf{Cursor} \to \mathsf{Cursor}$ be a function that transforms a cursor. `savec` allows us to navigate the reference frame arbitrarily within the argument `op` but restore it afterwards. We can compose these combinators into a useful pattern where the cursor is navigated to take some action and then restored afterward. This pattern allows us to avoid ambiguity in our references by staging all operations relative to a stable point within the AST.

```
reframe = lambda move, op: savec(seq(nav(move), op))
```

Because `nav` forwards the cursor before performing any spatial navigation, these combinators have now recreated linear-time reference frames (Section 5.4.1) used by strategy languages and ELEVATE.

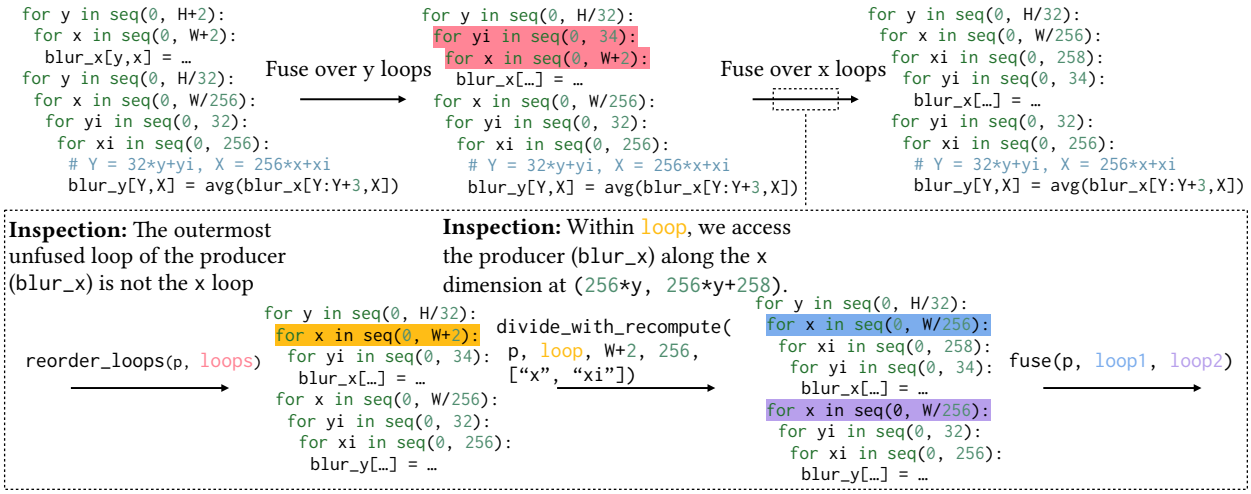


Figure 5.11: Exo implementation of Halide's `blur_x.compute_at(blur_y, x)` operation. The starting code is a tiled version of the blur algorithm. We first fuse the `blur_x` and `blur_y` computations at the `y` loop level and then fuse at the `x` loop level. Focusing on the second fusion, the bounds-inference inspection described in Section 5.3 determines the producer (`blur_x`) accesses in the `x` loop. Then, a sequence of primitive actions is applied to the object code to yield code fused at the `x` loop level.

Reproducing Halide

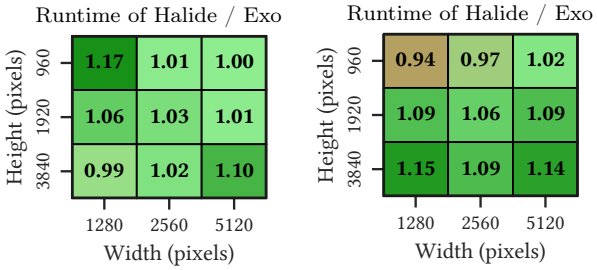
Blur in Halide and Exo As shown in Figure 5.12, the 3x3 blur algorithm and schedule from the Halide paper [25] uses built-in actions (`compute_at`, `tile`, `vectorize`, `parallel`) and nominal referencing. When `compute_at` is called without a corresponding `store_at`, Halide automatically stores the buffer at the same loop level as the computation. The equivalent Exo starting object code (Figure 5.11 upper left, after tiling to `y` and `x` has already been applied) is longer than the Halide version because Exo's lower-level IR explicitly expresses loops, allocations, and read/write/reduce statements, while Halide is restricted to pure functions of integer coordinates. For simplicity, we restrict input images to whole multiples of the tile size.

```
// ALGORITHM
blur_x(x,y)=(inp(x,y)+inp(x+1,y)+inp(x+2,y))/3;
blur_y(x,y)=(blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;
// SCHEDULE
blur_y.tile(x, y, xi, yi, 256, 32)
      .vectorize(xi, 16)
      .parallel(y);
blur_x.compute_at(blur_y, x)
      .vectorize(x, 16);
```

Figure 5.12: Halide's blur algorithm and schedule.

```
p = H_tile(p,"blur_y","y","x","yi","xi",32,256)
p = H_compute_store_at(p,"blur_x","blur_y","x")
p = H_parallel(p, "y")
p = H_vectorize(p, "blur_x", "xi", 16)
p = H_vectorize(p, "blur_y", "xi", 16)
p = H_store_in(p, "blur_x", DRAM_STACK)
```

Figure 5.13: Exo's blur schedule.



(a) 2D 3×3 box blur. (b) Unsharp masking.

	Rewrites	Exo Schd.	Halide Schd.
blur	116	6	5
unsharp	975	26	13

(c) Lines of code for the Exo schedule, Halide schedule, and the number of rewrites performed during the Exo scheduling process.

Figure 5.14: Exo schedules show competitive performance to corresponding expert-written Halide schedules on a variety of image sizes.

Exo Blur Schedule Halide's `compute_at` and `store_at` scheduling operations use automated bounds inference to determine loop bounds and buffer sizes. We leverage the previously discussed bounds-inference inspection (Section 5.3) to reproduce these Halide operations. Figure 5.11 summarizes our `compute_at` implementation, and similar ideas apply to `store_at`. To bridge the gap between Halide's and Exo's referencing schemes, we defined `H_`-prefixed functions that expect nominal references and internally convert to Exo's cursor references. Halide's nominal referencing scheme uses action-specialized built-in navigation. For example, `blur_x.compute_at(blur_y, x)` refers to the `x` loop enclosing the `blur_y` computation. Since cursors generalize nominal referencing, this simply involves translating the built-in navigation performed by Halide's scheduling operations. As shown in Figure 5.13, using the Halide-like scheduling operations with nominal referencing, Exo can reproduce Halide-style schedules. `H_compute_store_at` is called to implement Halide's aforementioned compile-time decision of calling both `compute_at` and `store_at`.

With this interface, we optimized 3×3 box blur and unsharp masking and compared the results with expert-written Halide schedules [57]. Figure 5.14 shows that Exo's Halide-like schedules offer similar performance to Halide.

Chapter 6

Value-Sensitive Analysis for Looping Array Code

In this chapter, we return to a limitation left open by the program analysis of Chapter 3. Exo's effect analysis (Section 3.4) is value-insensitive on buffers: it justifies a rewrite by proving that statements touch disjoint locations, discarding the values written. Its global dataflow analysis (Section 3.4.3) is control-sensitive: it propagates the symbolic control values that guard each statement, including scalar configuration fields, but it summarizes loops coarsely, collapsing any loop-varying configuration field to an unknown value. This suffices for the scheduling primitives of Chapter 4 and the libraries of Chapter 5, but it leaves a class of optimizations—removing redundant array writes, fusing loops with overlapping stores, and safely binding loop-varying configuration state such as RISC-V's vector length—out-of-reach, because their safety depends on *what* values are written, not merely *where*. This chapter develops Prexo, an abstract-interpretation layer built on the Exo IR that recovers this information. Prexo introduces value-sensitivity along two axes that Exo's control-sensitive dataflow does not reach: from loop-invariant to loop-varying configuration state, and from scalar configuration to arbitrary array contents. We introduce a cylindrical algebraic decomposition (CAD) tree domain and flood-fill widening to track array contents across both space and time, and we expose the result through an analysis API and three new scheduling operations. The analysis is external to the core compiler, and its guarantees compose with the equivalence-preserving primitives of Chapter 4.

6.1 Beyond Dependency Analysis

As established in previous chapters, USLs enable performance optimization by separating algorithm specification ("what to compute" as a program) from scheduling ("how to compute" the program), allowing rapid exploration of optimizations for locality, parallelism, and recomputation without modifying the core computation logic. While classic USLs like Halide [25,47] and TVM [44] use functional-programming paradigms, recent imperative USLs like Exo and Allo [58] better align with hardware execution models through explicit loads, stores, and control flow, offering more intuitive reasoning for performance engineers.

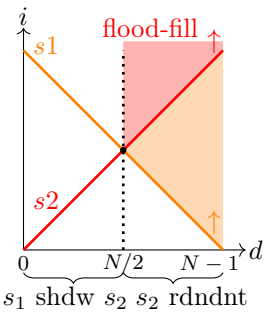
Unlike functional USLs, where computations are expressed as pure functions of their

inputs, imperative USLs require reasoning about the safety of scheduling operations in the presence of state mutation and side effects, which poses unique challenges in program analysis. Traditional dependency analysis treats statements as opaque operations and justifies rewrites by proving that statements are independent–that they access disjoint sets of memory locations. However, some program optimizations can only be justified by examining the actual values being computed. We call this "value-sensitive analysis," which reasons about the specific values flowing through the program rather than just their dependencies, enabling us to identify when certain statements become redundant or when new computations can be inserted without changing program semantics.

To motivate value-sensitive analysis for performance optimization, consider the loop shown on the right. Each element of x is written twice with the same value, making it safe to remove either statement. However, traditional dead-code analysis cannot identify this optimization opportunity. If we attempt to remove $s2$, we encounter a problem: on iteration 0, the effect $x[0] = y[0]$ is overwritten by $s1$ on the final iteration, which traditional analysis can detect. But on the final iteration $N-1$, the execution $x[N-1] = y[N-1]$ is not overwritten by $s1$ at all—instead, it is *redundant* because it writes the same value that already exists at that memory location. Traditional dead-code analysis fails here because it does not consider whether the value being written is *the same as the right before* the statement executes. Since redundancy provides an equally valid justification for statement removal, $s2$ can be safely removed, but the reasoning differs across loop iterations.

```
for i in 0..N:
s1:  x[N-1-i] = y[N-1-i]
s2:  x[i] = y[i]                ✓
```

Consider the space-time diagram on the right, depicting the contents of array x at location d (horizontal axis) after iteration i (vertical axis) completes. Statements $s1$ and $s2$ appear as diagonal lines marking the points in space and time where they modify array x. This diagram shows that the statement which writes the final (vertically highest) value depends on array position: in the first half of the array, we are justified in deleting $s2$ because it is overwritten, or *shadowed*, by $s1$ (shown as s_1 shadow (shdw) s_2). In the second half, this justification is no longer sound. Instead, we must verify that values written by $s1$ reach $s2$ uninterrupted and that $s2$ writes identical values (shown as s_2 redundant (rdndnt)). This requires value-sensitive analysis that tracks x across both space and time. Existing array-content analyses using abstract interpretation [59–62] cannot discover that partitioning should occur at index $N/2$, since this value is absent from the source code, leading to imprecision.



We introduce a *cylindrical algebraic decomposition (CAD) tree* domain and *flood-fill* widening to address this issue. The CAD tree domain decomposes the array's iterator-offset space (i, d) into sign-invariant cells and enables us to systematically find partitions for values that only appear semantically in the code (such as $\lfloor N/2 \rfloor$), as CAD's projection phase finds the resultants of input polynomials (namely $2d - N + 1$ in this case) from which it trivially obtains the partition. Furthermore, the flood-fill widening allows us to precisely capture values flowing through the loop by propagating stored values upwards in time in the iterator-offset space (shown by the upward arrow in the inset figure). This allows us to track x in space to see if $s1$ reaches $s2$ uninterrupted and with what value.

Our system, Prexo, builds on the Exo IR and analyses of Chapters 3–5 and combines the obtained abstract value information with the scheduling transformations to enable value-sensitive analysis for imperative USLs. We propose a new analysis API that constructs safety-check queries using abstract values, which is further used in the implementation of three scheduling operations that enable safe program transformations for inserting, deleting, and binding mutable variable stores.

We evaluate Prexo from two perspectives. First, we measure the expressiveness and runtime performance of the CAD-tree domain design and flood-fill widening on targeted microbenchmarks. We demonstrate that our analysis is both more expressive and scales better than a previous abstract-interpretation approach for array analysis [59]. Second, we evaluate Prexo by optimizing kernels on the RVV architecture to assess its practical utility and expressiveness in performance-engineering applications. RVV's vector length-agnostic instructions pose challenges to compilers due to the global "vector length" (*vl*) state, which modifies the vector-length behaviour of executed instructions. Naive dependency analysis cannot safely reason about rewriting *vl*, which motivates our value-sensitive analysis approach. We achieve performance comparable to expert-written DAXPY code with modest analysis runtime overhead, with end-to-end compilation completing in under 3 seconds.

6.2 Motivational Examples

In many performance-critical scenarios, transformations cannot be handled by dependency analysis. We illustrate this with two concrete examples.

6.2.1 Example 1: Sliding-Window Optimization

We present code examples using Exo IR rather than Prexo IR for readability. Although our analysis operates on Prexo IR (defined in Section 6.4), its SSA form makes it verbose for illustrative purposes. As shown in Section 6.3, Prexo takes Exo IR as input and converts it to Prexo IR for analysis. Recall that, in Exo IR, both procedure arguments and variable declarations follow the syntax $\langle name \rangle : \langle type \rangle$ `[`$\langle size \rangle$`]`. In `foo` (Figure 6.1(a)), `f32` is a numeric type for 32-bit floating point. The $\langle size \rangle$s may be constants or refer dependently to other arguments. For instance, in `foo`, arrays can use dependent sizes like `[n + m]` that reference the procedure's dimension parameters. Sequential loops are written as `for i in seq(0, m)`, iterating from 0 to $m - 1$ (inclusive).

Function `foo` in Figure 6.1(a) takes two size parameters `n` and `m`, scalar values `a` and `res`, and an input array `inp` of size `n + m`. It creates a temporary buffer `x` of the same size and copies the input array through nested loops that iterate over indices `i` from 0 to `n-1` and `j` from 0 to `m-1`, while accumulating `x[i+j]*a` to `res`. While the temporary buffer `x` serves no functional purpose in this example since it merely duplicates the input array, it demonstrates a common optimization pattern for utilizing vector registers or accelerator scratchpads.

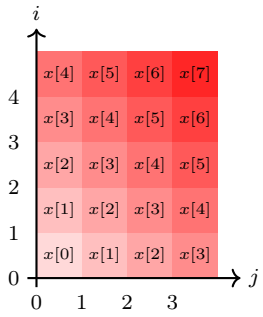
Figure 6.1(c) shows the iteration space of the load statement $x[i + j] = inp[i + j]$, where array elements with identical offsets are highlighted in matching red tones. This redundancy occurs when multiple (i, j) pairs produce the same index $i + j$, such as $(0, 1)$ and $(1, 0)$ both yielding $i+j = 1$. Writing identical values to the same location wastes computational resources,

```
def foo(n: size, m: size, a: f32,
        res: f32, inp: f32[n + m]):
    x: f32[n + m]
    for i in seq(0, n):
        for j in seq(0, m):
            x[i + j] = inp[i + j] ←
            res += x[i + j] * a
```

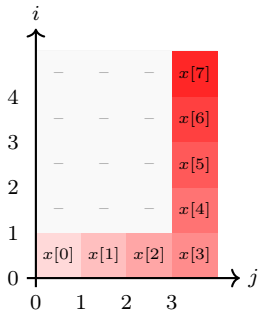
(a) Original code for `foo`

```
def foo(...):
    x: f32[n + m]
    for i in seq(0, n):
        for j in seq(0, m):
            if i == 0 or j == m - 1:
                x[i + j] = inp[i + j] ←
            res += x[i + j] * a
```

(b) Optimized code for `foo`



(c) Iteration space for $x[i+j] = inp[i+j]$ of (a)



(d) Iteration space for $x[i+j] = inp[i+j]$ of (b)

Figure 6.1: Code snippets for `foo` illustrating sliding-window optimization from (a) to (b). (c) and (d) show the iteration space when $n = 5$ and $m = 4$.

making it essential to remove redundant memory operations for optimal performance. To achieve this, the original code can be transformed into the new code shown in Figure 6.1(b), which includes a guard `if i == 0 or j == m - 1` that shields the code from redundant loads. The iteration space of the transformed code, shown in Figure 6.1(d), demonstrates this removal, with values from 0 to $n + m - 2$ being loaded only once.

This pattern is common in many high-performance kernels, such as convolution, due to overlapping regions in the updated buffer. However, this poses a challenge for program analysis, as it requires safely reasoning about the removal of redundant writes when the newly written value matches previously computed value. Symbolically, this can be represented as:

$$x[\text{idx}] = v_1; \quad x[\text{idx}] = v_2 \;\rightsquigarrow\; \begin{cases} x[\text{idx}] = v_1, & \text{if } v_1 = v_2, \\ x[\text{idx}] = v_1;\ x[\text{idx}] = v_2, & \text{otherwise.} \end{cases}$$

This type of transformation depends not only on iteration-space properties but also on data values. Therefore, dependency analysis, which relies on iteration-domain and access-pattern analysis, fails to directly capture these opportunities. Value-sensitive analysis is, therefore, essential in cases where eliminating or merging redundant writes can significantly optimize the program runtime.

6.2.2 Example 2: Loop Fusion and Redundant Writes

```
x : f32[n+1]
for i in seq(0, n):
  x[i] = y[i]
for i in seq(0, n):
  x[i+1] = y[i+1]
```

(a) Unfused

```
x : f32[n+1]
for i in seq(0, n):
  x[i] = y[i]
  x[i+1] = y[i+1]
```

(b) Fused

```
x : f32[n+1]
for i in seq(0, n+1):
  x[i] = y[i]
```

(c) Redundant write removed

Figure 6.2: Code snippets illustrating loop optimization from (a) to (b) to (c).

Consider a second example in Figure 6.2. The initial code in Figure 6.2(a) has two separate loops: the first copies elements from `y` to `x` for indices 0 through `n-1`, while the second copies elements for indices 1 through `n`. This results in redundant writes to `x[i]` for $i \in [1, n-1]$. Figures 6.2(b) and 6.2(c) show a loop-optimization sequence. First, loop fusion transforms (a) to (b) by combining both loops into a single loop body, eliminating the overhead of two separate loop structures. However, this intermediate form (b) still contains redundant memory operations. The memory location of `x[i+1]` has already been written by `x[i]` in the previous iteration with the same value for $i \in [0, n-1]$. The second optimization step transforms (b) to (c) by recognizing this pattern and extending the loop bounds to `seq(0, n+1)` with a single write operation, eliminating all redundant stores.

The safety of both transformations hinges on value-sensitive analysis. Traditional dependency-based approaches, such as those in Exo for loop fusion, would reject both transformations. Standard fusion analysis requires that loop effects commute, but the two loops in Figure 6.2(a) write to overlapping memory regions (`x[1]` through `x[n-1]`), making their effects noncommutative. The fundamental limitation is that dependency analysis only tracks *which* locations are accessed, not *what* values are written. Value-sensitive analysis addresses this limitation by recognizing that both loops copy identical values from `y[i]` to `x[i]` within the overlapping region.

6.3 System Overview

As shown in Figure 6.3, value-sensitive analysis is embedded within Exo's compilation and optimization stack (Chapters 3–5), which provides type and bound checking, scheduling operations, and a backend for memory and precision checks as well as code generation. Value-sensitive analysis operates exclusively on the Prexo IR, and Exo IR is converted to Prexo IR as shown in Figures 6.3 and 6.4. For the syntax of Exo IR, we refer readers back to Section 3.2. The syntax and semantics of Prexo IR are defined in the next section; intuitively, the transformation rewrites imperative array code with loops into a system of space-time arrays. This conversion occurs at Exo's compilation time through a standard SSA-conversion process:

1. Inlining: All function calls and window statements in the Exo IR are inlined.

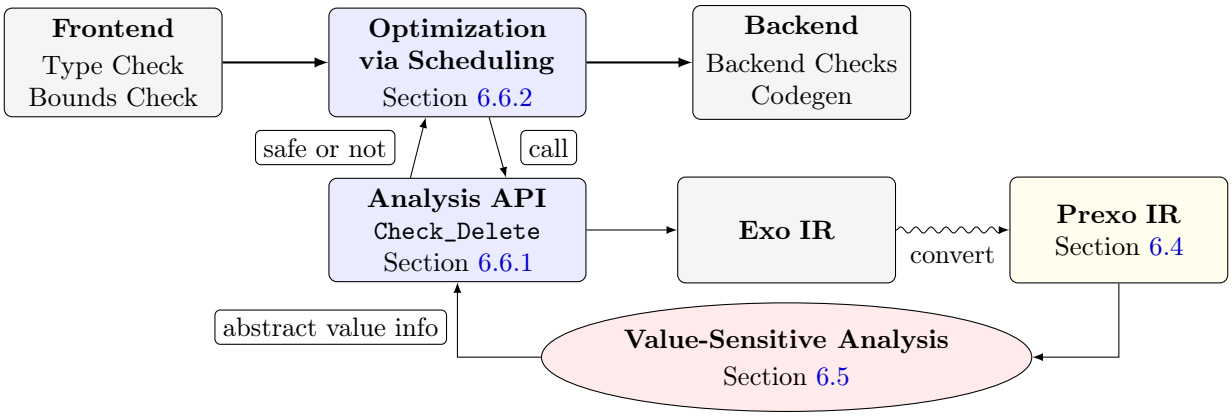


Figure 6.3: System overview of Prexo, showing how value-sensitive analysis integrates into the compiler stack of Exo (Chapters 3–5). The Prexo IR (Section 6.4), value-sensitive analysis (Section 6.5), and analysis API (Section 6.6) represent the contributions of this chapter. Gray boxes (frontend, backend, Exo IR) remain unchanged.

```
for i in seq(0, 10):
    x[i] = 0.0
```

(a) Exo IR

convert

```
x_1 =\i \d0 (i == 0 ? x[d0] : x_2[i - 1, d0]) # LoopStart
x_2 =\i \d0 (i == d0 ? 0.0 : x_1[i, d0])
x_3 =\d0 (10 > 0 ? x_2[10 - 1, d0] : x[d0]) # LoopExit
```

(b) Prexo IR

Figure 6.4: Conversion from Exo IR to Prexo IR.

2. Arguments: Exo IR's function arguments are not converted to Prexo IR, which has no notion of program parameters. Instead, they surface as free variables in the resulting Prexo IR.

3. SSA-ify Assignments: Each assignment in the Exo IR introduces a fresh symbol for Prexo IR. Exo IR's allocation statements are ignored, as SSA allocates new variables upon assignment.

4. Space-time Arrays: When loops enclose arrays, loop counters are added as new array dimensions. For example, a one-dimensional array in Exo IR becomes three-dimensional in Prexo IR when nested within two loops. The resulting dimensions track both space (via array offsets) and time (via loop iteration counters), as formalized in the following sections.

5. Merging Control Flows: At control-flow merge points (loop entries, loop exits, and if exits), additional Prexo IR assignment statements are inserted. These serve as phi nodes, similar to those in SSA-based IRs like LLVM IR, allowing Prexo IR to preserve functional behavior with Exo IR even without explicit branching and loop constructs.

p : Proc	::= Fix s	program
s : Stmt	::= $s_1; s_2$	sequencing
	$\mid y = \lambda\imath^*.(b\,?\,c_t : c_f)$	control assign
	$\mid x = \lambda\imath^*.\lambda\eta^*.(b\,?\,e_t : e_f)$	data assign
c : CExpr	::= n	integer
	$\mid y \mid y[c^*]$	scalar/array reads
	$\mid c_1{+}c_2 \mid c_1{-}c_2 \mid c{*}n \mid c/n$	(quasi-) affine arith
e : DExpr	::= d	data value
	$\mid x \mid x[c^*]$	scalar/array reads
	$\mid e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1/e_2$	compound expr
b : BExpr	::= t	Boolean
	$\mid b_1$ and $b_2 \mid b_1$ or b_2	logical ops
	$\mid c_1 == c_2 \mid c_1{<}c_2 \mid c_1{\leq}c_2$	relational ops
v : Var	::= $x \in \mathbb{X}$	data variable
	$\mid y \in \mathbb{Y}$	control variable
n	$\in \mathbb{Z}$	integer
t	$\in \mathbb{B}$	Boolean
d	$\in \mathbb{D}$	data values

Figure 6.5: The syntax of Prexo IR. $\imath^*$ and η^* are tuples of control variables $y \in \mathbb{Y}$, each denoting loop-iterator variables and array-offset variables.

6.4 Program Syntax and Concrete Semantics

6.4.1 Syntax of Prexo IR

Figure 6.5 defines the syntax of Prexo IR. Prexo explicitly distinguishes between control and data expressions and is a static control program, where control flow depends only on loop indices and control/constant parameters, not on data computed at runtime. We denote the sets of data variables and control variables by $x \in \mathbb{X}$ and $y \in \mathbb{Y}$, respectively. Each control variable y has an iterator offset arity $\mathrm{itr}(y)$ and an array offset space $\mathbb{Z}^{\mathrm{itr}(y)}$. Each data variable x has an iterator offset arity $\mathrm{itr}(x)$, an array offset arity $\dim(x)$, and an array offset space $\mathbb{Z}^{\mathrm{itr}(x)} \times \mathbb{Z}^{\dim(x)}$. The truth values $\mathbb{B} \triangleq \{\mathrm{true}, \mathrm{false}\}$ are ranged over by a metavariable t, and n is the metavariable for $\mathbb{Z} \triangleq \{\cdots, -1, 0, 1, 2, \cdots\}$. The metavariable d ranges over data values in $\mathbb{D}$, including 8-, 32-, and 64-bit integers as well as single- and double-precision floating-point numbers.

Control expressions CExpr encompass integer, scalar and array control variable reads, and (quasi-) affine arithmetic operations. Data expressions DExpr are similarly defined, but operations are not constrained to be affine. Boolean expressions BExpr include the control expressions with basic logical and relational operations. Indices c^* (we use $.^*$ to denote a tuple) used in array reads are specified as tuples of control expressions, expressing access to multidimensional arrays. In both data and control expressions, scalar read is syntactic sugar for an array access of $c^* = \langle\,\rangle$ (empty tuple). A read $y[c^*]$ requires $|c^*| = \mathrm{itr}(y)$, and $x[c^*]$ requires $|c^*| = \dim(x) + \mathrm{itr}(x)$. Scalar reads x and y are well-typed only when $\mathrm{itr}(y) = 0$ and $\dim(x) = \mathrm{itr}(x) = 0$.

A statement $s \in$ Stmt can be a sequencing, control-assignment, or data-assignment

$$
\begin{array}{l}
\mathcal{E}_c^e\,[\![c]\!] : \Sigma_c \to \mathbb{Z}_\perp^\top \\
\qquad\qquad \mathcal{E}_c^e\,[\![n]\!]\,(\sigma_c) = n \\
\qquad\qquad \mathcal{E}_c^e\,[\![y]\!]\,(\sigma_c) = \sigma_c(y, \langle\,\rangle) \\
\qquad\qquad \mathcal{E}_c^e\,[\![y[c^*]]\!]\,(\sigma_c) = \sigma_c(y, \langle \mathcal{E}_c^e\,[\![c_i]\!]\,(\sigma_c)\,\rangle_i) \\
\qquad\qquad \mathcal{E}_c^e\,[\![c_1 \text{ op } c_2]\!]\,(\sigma_c) = \mathcal{E}_c^e\,[\![c_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_c^e\,[\![c_2]\!]\,(\sigma_c) \quad (\text{op} \in \{+, -, *, /\}) \\
\hline
\mathcal{E}_d^e\,[\![e]\!] : \Sigma_c \to \Sigma_d \to \mathbb{D}_\perp^\top \\
\qquad\qquad \mathcal{E}_d^e\,[\![d]\!]\,(\sigma_c, \sigma_d) = d \\
\qquad\qquad \mathcal{E}_d^e\,[\![x]\!]\,(\sigma_c, \sigma_d) = \sigma_d(x, \langle\,\rangle) \\
\qquad\qquad \mathcal{E}_d^e\,[\![x[c^*]]\!]\,(\sigma_c, \sigma_d) = \sigma_d(x, \langle \mathcal{E}_c^e\,[\![c_i]\!]\,(\sigma_c)\,\rangle_i) \\
\qquad\qquad \mathcal{E}_d^e\,[\![e_1 \text{ op } e_2]\!]\,(\sigma_c, \sigma_d) = \mathcal{E}_d^e\,[\![e_1]\!]\,(\sigma_c, \sigma_d)\ \widetilde{\text{op}}\ \mathcal{E}_d^e\,[\![e_2]\!]\,(\sigma_c, \sigma_d) \quad (\text{op} \in \{+, -, *, /\}) \\
\hline
\mathcal{E}_b\,[\![b]\!] : \Sigma_c \to \mathbb{B}_\perp^\top \\
\qquad\qquad \mathcal{E}_b\,[\![t]\!]\,(\sigma_c) = t \\
\qquad\qquad \mathcal{E}_b\,[\![b_1 \text{ op } b_2]\!]\,(\sigma_c) = \mathcal{E}_b\,[\![b_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_b\,[\![b_2]\!]\,(\sigma_c) \quad (\text{op} \in \{\texttt{and}, \texttt{or}\}) \\
\qquad\qquad \mathcal{E}_b\,[\![c_1 \text{ op } c_2]\!]\,(\sigma_c) = \mathcal{E}_c^e\,[\![c_1]\!]\,(\sigma_c)\ \widetilde{\text{op}}\ \mathcal{E}_c^e\,[\![c_2]\!]\,(\sigma_c) \quad (\text{op} \in \{==, <, \leq\}) \\
\hline
\mathcal{E}_c\,[\![s]\!] : \Sigma_c \to \Sigma_c \\
\qquad\qquad \mathcal{E}_c\,[\![s_1; s_2]\!]\,(\sigma_c) = \mathcal{E}_c\,[\![s_2]\!]\,\big(\mathcal{E}_c\,[\![s_1]\!]\,(\sigma_c)\big) \\
\qquad \mathcal{E}_c\,[\![\, y = \lambda \imath^*.(b\ ?\ c_t : c_f)\,]\!]\,(\sigma_c) = \sigma_c\big[y \mapsto a_y\big] \\
\text{where } a_y : \mathbb{Z}^{\text{itr}(y)} \to \mathbb{Z}_\perp^\top \text{ and } a_y(z^*) = \text{ite}\big(\mathcal{E}_b\,[\![b]\!]\,(\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}_c^e\,[\![c_t]\!]\,(\sigma_c[\imath^* \mapsto z^*]),\ \mathcal{E}_c^e\,[\![c_f]\!]\,(\sigma_c[\imath^* \mapsto z^*])\big). \\
\hline
\mathcal{E}_d\,[\![s]\!] : \Sigma_c \to \Sigma_d \to \Sigma_d \\
\qquad\qquad \mathcal{E}_d\,[\![s_1; s_2]\!]\,(\sigma_c, \sigma_d) = \mathcal{E}_d\,[\![s_2]\!]\,\big(\mathcal{E}_c\,[\![s_1; s_2]\!]\,(\sigma_c),\ \mathcal{E}_d\,[\![s_1]\!]\,\big(\mathcal{E}_c\,[\![s_1]\!]\,(\sigma_c), \sigma_d\big)\ \big) \\
\mathcal{E}_d\,[\![\, x = \lambda \imath^*.\lambda \eta^*.(b\ ?\ e_t : e_f)\,]\!]\,(\sigma_c, \sigma_d) = \sigma_d\big[x \mapsto a_x\big] \\
\quad \text{where } a_x : \mathbb{Z}^{\text{itr}(x)} \times \mathbb{Z}^{\dim(x)} \to \mathbb{D}_\perp^\top \text{ and } a_x(z_1^*, z_2^*) = \text{ite}\big(\mathcal{E}_b\,[\![b]\!]\,(\sigma_c'),\ \mathcal{E}_d^e\,[\![e_t]\!]\,(\sigma_c', \sigma_d),\ \mathcal{E}_d^e\,[\![e_f]\!]\,(\sigma_c', \sigma_d)\big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sigma_c' = \sigma_c[\imath^* \mapsto z_1^*, \eta^* \mapsto z_2^*]. \\
\hline
\mathcal{E}\,[\![p]\!] : \Sigma_c \times \Sigma_d \to \Sigma_c \times \Sigma_d \\
\qquad\qquad \mathcal{E}\,[\![\text{Fix } s]\!]\,(\sigma_c^0, \sigma_d^0) = (\sigma_c^*,\ \sigma_d^*) \\
\qquad\qquad\qquad\qquad \sigma_c^* = \text{lfp}\big(F_c\big) \quad \text{where } F_c(\sigma_c) = \sigma_c^0\ \sqcup_c\ \mathcal{E}_c[\![s]\!](\sigma_c) \\
\qquad\qquad\qquad\qquad \sigma_d^* = \text{lfp}\big(F_d^{\sigma_c^*}\big) \quad \text{where } F_d^{\sigma_c}(\sigma_d) = \sigma_d^0\ \sqcup_d\ \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)
\end{array}
$$

Figure 6.6: Denotational semantics of Prexo IR.

statement. Following prior work [63,64], Prexo IR adopts the SSA form, where each assignment represents both the declaration and initialization of control or data variables. During the Exo IR to Prexo IR conversion, whenever an assignment overwrites an array, it is renamed to a fresh identifier, effectively treating it as a distinct array instance. The tuples $\imath^*$ and η^* consist of distinct λ-bound control variables from $\mathbb{Y}$, where $\imath^*$ represents the loop iterators (temporal dimension) and η^* represents the array-offset variables (spatial dimension), together forming the space-time array of assign statements. Notably, the syntax excludes "if" and "loop" statements. Instead, conditional logic is expressed via ternary operators $(b\ ?\ e_t : e_f)$ and $(b\ ?\ c_t : c_f)$ within assignments, and loops are transformed into space-time arrays during the Exo IR to Prexo IR conversion. The conversion also inserts additional assignments to handle control-flow merging. Free (unassigned) variables in the Prexo IR program Fix s represent the function input parameters in Exo IR.

6.4.2 Concrete Semantics of Prexo IR

Figure 6.6 shows denotational semantics of Prexo IR. Both control and data values range over flat lattices $\mathbb{Z}_\perp^\top \triangleq \mathbb{Z} \cup \{\perp, \top\}$ and $\mathbb{D}_\perp^\top \triangleq \mathbb{D} \cup \{\perp, \top\}$ with $\perp \sqsubseteq v \sqsubseteq \top$ for all concrete v. Let $\mathbb{B}_\perp^\top \triangleq \mathbb{B} \cup \{\perp, \top\}$ with the flat order $\perp \sqsubset \mathsf{false}, \mathsf{true} \sqsubset \top$ ($\mathsf{false}, \mathsf{true}$ incomparable). Control and data variables live in disjoint environments. A control environment $\sigma_c \in \Sigma_c$ maps each $y \in \mathbb{Y}$ to an array $\mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}_\perp^\top$. A data environment $\sigma_d \in \Sigma_d$ maps each $x \in \mathbb{X}$ to an array $\mathbb{Z}^{\mathrm{itr}(x)} \times \mathbb{Z}^{\dim(x)} \to \mathbb{D}_\perp^\top$. Since data variables can depend on control variables but not vice versa, the evaluation functions are defined as $\mathcal{E}_c[\![s]\!] : \Sigma_c \to \Sigma_c$ and $\mathcal{E}_d[\![s]\!] : \Sigma_c \times \Sigma_d \to \Sigma_d$. We write $\sqcup_c$ (resp. $\sqcup_d$) for the join on Σ_c (resp. Σ_d), which are defined pointwise. Hence $(\Sigma_c, \sqsubseteq_c)$ and $(\Sigma_d, \sqsubseteq_d)$ are complete lattices.

Expressions are evaluated over the flat lattices $\mathbb{Z}_\perp^\top$, $\mathbb{D}_\perp^\top$, and $\mathbb{B}_\perp^\top$ with $\perp \sqsubseteq v \sqsubseteq \top$. All case splits below are applied in the written order (left-to-right). For each arithmetic operator $\mathrm{op} \in \{+, -, *, /\}$ we use the strict lift $u \,\widetilde{\mathrm{op}}\, v \;=\; \perp$ if $u{=}\perp$ or $v{=}\perp$; $= \top$ if $u{=}\top$ or $v{=}\top$; $= \mathrm{op}(u, v)$ when defined (e.g. $v \neq 0$ for '/'); and $= \perp$ otherwise. We interpret '/' as a floor division for control expressions. Relational and logical operators are lifted analogously: for $\mathrm{op} \in \{==, <, \leq, \texttt{and}, \texttt{or}\}$, $u \,\widetilde{\mathrm{op}}\, v = \perp$ if either operand is $\perp$, $= \top$ if either is $\top$, and $= \mathrm{op}(u, v)$ otherwise. Array reads are strict in each index: $y[c^*]$ and $x[c^*]$ first evaluate c^* via $\mathcal{E}_c^e$; if any index is $\perp$ (resp. $\top$) the read yields $\perp$ (resp. $\top$), otherwise the concrete lookup is used. The conditional is $\mathrm{ite}(b, u, v) = u$ if $b{=}\mathsf{true}$, v if $b{=}\mathsf{false}$, $\perp$ if $b{=}\perp$, and $u \sqcup v$ if $b{=}\top$, where $\sqcup$ is the flat join in the target lattice.

Sequential composition for control variables composes by standard function composition; y-assignment adds $a_y : \mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}_\perp^\top$ defined pointwise with a parameter z^*. The data-evaluation function composes by first evaluating the data store of s_1 under $(\mathcal{E}_c[\![s_1]\!](\sigma_c), \sigma_d)$ and then s_2 under the accumulated control store $\mathcal{E}_c[\![s_1; s_2]\!](\sigma_c)$ and the result of evaluating s_1. a_x also is defined pointwise. Control assignments leave σ_d unchanged, and data assignments leave σ_c unchanged.

A program is written $\mathrm{Fix}\, s$ and evaluated from an initial store (σ_c^0, σ_d^0). We compute a control fixed point $\sigma_c^* = \mathrm{lfp}(F_c)$ where $F_c(\sigma_c) = \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)$, then a data fixed point $\sigma_d^* = \mathrm{lfp}(F_d^{\sigma_c^*})$ where $F_d^{\sigma_c}(\sigma_d) = \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Both fixed points exist by Tarski–Knaster because the stores form complete lattices and both F_c and $F_d^{\sigma_c}$ are monotone (proof in Appendix B.1). Although we write lfp to emphasize the order-theoretic construction, the semantics is deterministic: since statement execution is deterministic, the denotation of $\mathrm{Fix}\, s$ is uniquely determined by s and the initial store.

6.5 Value-Sensitive Analysis

To enable sound safety checking, we propose a value-sensitive analysis that tracks program variables and overapproximates their values. We introduce a novel abstract domain called the cylindrical algebraic decomposition (CAD) tree domain, and develop an efficient value-sensitive analysis on top of it by designing a new flood-fill widening operator. In this section, we define the CAD tree domain (Section 6.5.1) and its abstraction and concretization functions (Section 6.5.2). Subsequently, we detail the transfer functions (Section 6.5.3) and the flood-fill widening operator (Section 6.5.4) and establish the soundness guarantee of the analysis.

$\zeta : \mathcal{B}_C$	$::= \top_\zeta \mid \bot_\zeta$	top/bot
	$\mid n \mid y$	ctrl scalars
	$\mid \zeta_1{+}\zeta_2 \mid \zeta_1{-}\zeta_2 \mid \zeta{*}n$	affine arith
	$\mid \zeta/n$	quasi affine

$\delta : \mathcal{B}_D$	$::= d \mid x[\zeta^*]$	data/arrays
	$\mid \top_\delta \mid \bot_\delta$	top/bot

g : Guard	$::= t$	Boolean
	$\mid \zeta_1 \,\mathrm{op}\, \zeta_2$	op $\in \{==, <, \leq\}$
	$\mid g_1$ `and` g_2	logical and

(a) Base domains $\mathcal{B}_C$ and $\mathcal{B}_D$.

$\tau_\pi : \mathcal{T}\langle\pi\rangle$	$::= (\text{vars} : \imath^*\eta^*, \text{cuts} : \zeta^*, \text{tree} : \mathit{node}\langle\pi\rangle)$
	$\mid \bot_{\mathcal{T}\langle\pi\rangle} \mid \top_{\mathcal{T}\langle\pi\rangle}$
$\mathit{node}\langle\pi\rangle$	$::= \text{Leaf}(\pi, \mathbb{Z}^{\lvert\imath^*\rvert+\lvert\eta^*\rvert}) \mid \text{LinSplit}(\mathit{cell}\langle\pi\rangle^*)$
$\mathit{cell}\langle\pi\rangle$	$::= \text{Cell}(g, \mathit{node}\langle\pi\rangle)$

(b) CAD-tree family $\mathcal{T}\langle\pi\rangle$.

$\mathcal{T}_C \triangleq \mathcal{T}\langle\zeta\rangle$	control tree domain
$\mathcal{T}_D \triangleq \mathcal{T}\langle\delta\rangle$	data tree domain

(c) Control and data CAD-tree domains.

Figure 6.7: Syntax of the base domains and the CAD-tree domains. In (b), $\mathcal{T}$ is parameterized by π to collapse the control and data tree definitions instantiated in (c). $\imath^*$ and η^* are tuples of control variables $y \in \mathbb{Y}$ denoting loop-iterator variables and array-offset variables, respectively.

6.5.1 CAD Tree Domain

We separate the *base domain* into a control domain $\mathcal{B}_C$ and a data domain $\mathcal{B}_D$ (Figure 6.7(a)). Elements of $\mathcal{B}_C$, written ζ, denote abstract control values and are closed under affine operators ($+$, $-$, $*n$) and a quasi-affine operator ($/n$), together with lattice elements $\top_\zeta$ and $\bot_\zeta$. Elements of $\mathcal{B}_D$, written δ, denote abstract data values: either a data value d, an array element $x[\zeta^*]$, or extreme elements $\top_\delta$ and $\bot_\delta$, where x is a free variable in the Prexo IR procedure and ζ^* is a tuple of control expressions from $\mathcal{B}_C$. Intuitively, $x[\zeta^*]$ carries relational information about arrays by referring to the values of the Exo IR procedure's data-array parameter, which appears as free variables in Prexo IR. For example, if ζ^* evaluates to index expressions $i_1, \ldots, i_k$, then $x[i_1, \ldots, i_k]$ is the abstract value symbolically depicting the value stored *at the beginning of* the Exo IR procedure. Guards g compare ζ-expressions via op $\in \{==, <, \leq\}$ and a conjunction is expressed with `and`. Disjunction is represented structurally by sibling CAD cells; we therefore restrict guards to conjunctions.

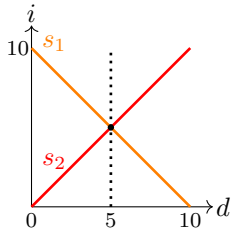
Appendix B.2 reviews classical CAD, a finite partition of $\mathbb{R}^n$ into semi-algebraic cells on which each input polynomial has constant sign [65,66]. We restrict the input polynomials to affine, yielding a cylindrical, sign-invariant decomposition with (potentially) rational roots; a cell is declared infeasible when its induced polyhedron contains no integer point, verified via an ILP feasibility check. We encode this as a CAD-tree whose leaves (cells) carry a base-domain annotation. The CAD algorithm takes ordered variables (which we term "vars") $(y_1, \ldots, y_n)$ and affine expressions of those variables ("cuts") as inputs and returns a CAD tree ("tree").

The CAD-tree domain is defined as a family $\mathcal{T}\langle\pi\rangle$ parameterized by the base domain type π (Figure 6.7(b)). An abstract value $\tau_\pi \in \mathcal{T}\langle\pi\rangle$ is $\bot_{\mathcal{T}\langle\pi\rangle}$, $\top_{\mathcal{T}\langle\pi\rangle}$, or a triple (vars : $\imath^*\eta^*$, cuts : ζ^*, tree : $\mathit{node}\langle\pi\rangle$). Here, $\imath^*\eta^* = (\imath_1 \prec \cdots \prec \eta_1 \prec \cdots) \subset \mathbb{Y}$ are totally ordered CAD generator variables each denoting loop-iterator and array-offset variables, and ζ^* is a tuple of quasi-affine expressions over the generator variables. *vars* and *cuts* yield a CAD tree represented by a *tree*: $\text{LinSplit}(\mathit{cell}\langle\pi\rangle^*)$ captures the decomposition of *cells* with pairwise disjoint guards; each $\text{Cell}(g, \mathit{node}\langle\pi\rangle)$ pairs a guard g (over generator variables *vars*) with a further subtree; leaves are $\text{Leaf}(\pi, \mathbb{Z}^{\lvert\imath^*\rvert+\lvert\eta^*\rvert})$, which store a base domain value π with an integer

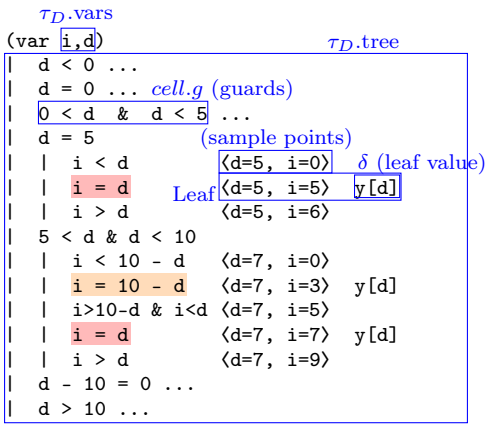
tuple $(n_1, \cdots, n_{|\iota^*|+|\eta^*|})$ representing a sample point for vars $(\iota_1 = n_1, \cdots, \eta_1 = n_{|\iota^*|}, \cdots)$ that route to the current cell. We use two CAD tree instances (Figure 6.7(c)): the control tree domain $\mathcal{T}_C \triangleq \mathcal{T}\langle\zeta\rangle$ and the data tree domain $\mathcal{T}_D \triangleq \mathcal{T}\langle\delta\rangle$, whose leaves carry, respectively, control base domain values and data base domain values. Note that $\eta^* = \langle\rangle$ for the control tree domain $\mathcal{T}_C$, as control variables do not have array offsets (Figure 6.5).

```
  for i in seq(0, 11):
s1:   x[10-i] = y[10-i]
s2:   x[i] = y[i]
```

(a) Array Reverse Array-Write in Exo IR



(b) Visualization of the CAD tree for (a)



(c) Pretty printing of the CAD tree for (a)

Figure 6.8: CAD tree domain for array reversal. (b) Space-time diagram of x at location d after iteration i, with diagonal lines for s_1 and s_2. (c) CAD-tree representation with implicit default leaf values $\bot_\delta$ (not shown).

Example 6.1. To build intuition for the CAD-tree domain, consider the concrete example in Figure 6.8, which demonstrates τ_D for the Array Reverse Array-Write microbenchmark also shown in Figure 6.15 (m). In this example, each element of data array x is written twice with the same y value through two statements for each iteration i from 0 to 10. Figure 6.8(b) illustrates this behavior through a space-time diagram, where the horizontal axis represents the array location d and the vertical axis represents the iteration count i. The diagonal lines mark the points in space and time where statements $s1$ and $s2$ modify array x, creating a partition at $d = 5$. Figure 6.8(c) shows the corresponding τ_D resulting from invoking CAD with the parameters (vars=$\langle i, d\rangle$, cuts=$\langle i - d, i + d - 10\rangle$). The highlighted guards indicate which statement's expression generated each constraint.

Employing the CAD-tree domain as an abstract domain enables us to decompose the array's iterator-offset space into sign-invariant cells. This decomposition supports efficient flood-fill widening for propagating values forward in iteration time, as discussed in Section 6.5.4. Furthermore, CAD's projection phase computes the resultants of input cuts, with which we can systematically identify *partitions* for values that appear only semantically in the code, such as $d = 5$ in the example.

Definition 6.2 (Partial orders on base domains)**.** Both $\mathcal{B}_C$ and $\mathcal{B}_D$ are flat lattices modulo a canonical syntactic congruence $\equiv$. We lift $\equiv$ pointwise to tuples ($\equiv^*$) and define it on $\mathcal{B}_D$ structurally so that $x[\zeta^*] \equiv x'[\zeta'^*]$ iff $x = x'$, $|\zeta^*| = |\zeta'^*|$, and $\zeta^* \equiv^* \zeta'^*$. The partial orders

are:

$$\zeta \sqsubseteq_{\mathcal{B}_C} \zeta' \iff (\zeta = \bot_\zeta) \ \vee \ (\zeta' = \top_\zeta) \ \vee \ (\zeta, \zeta' \notin \{\bot_\zeta, \top_\zeta\} \wedge \zeta \equiv \zeta'),$$
$$\delta \sqsubseteq_{\mathcal{B}_D} \delta' \iff (\delta = \bot_\delta) \ \vee \ (\delta' = \top_\delta) \ \vee \ (\delta, \delta' \notin \{\bot_\delta, \top_\delta\} \wedge \delta \equiv \delta').$$

For $\mathcal{B}_\pi \in \{\mathcal{B}_C, \mathcal{B}_D\}$, $\sqcup_\pi$ is the *flat* join modulo $\equiv$: $\bot_\pi$ is neutral, $\top_\pi$ is absorbing, and for non-extreme $\beta, \beta' \in \mathcal{B}_\pi$, $\beta \sqcup_\pi \beta' = \beta$ iff $\beta \equiv \beta'$, otherwise $\top_\pi$. $\mathcal{B}_C$ and $\mathcal{B}_D$ are complete lattices.

Definition 6.3 (Partial order $\sqsubseteq_{\mathcal{T}}$ on $\mathcal{T}\langle\pi\rangle$)**.** Let $\tau_i = (\text{vars} : \imath_i^* \eta_i^*, \text{cuts} : \zeta_i^*, \text{tree} : n_i) \in \mathcal{T}\langle\pi\rangle$ or $\tau_i \in \{\bot_{\mathcal{T}\langle\pi\rangle}, \top_{\mathcal{T}\langle\pi\rangle}\}$ for $i \in \{1, 2\}$. Let $\tau_i(\hat{n}) \in \mathcal{B}_C \cup \mathcal{B}_D$ denote the payload base-domain value at the leaf reached by traversing τ_i from the root, choosing at each LinSplit the child $\text{Cell}(g, \cdot)$ whose guard g is satisfied under $\hat{n}$, where $\hat{n} \in \mathbb{Z}^{|\imath^*|+|\eta^*|}$. Since CAD provides a complete decomposition, any $\hat{n} \in \mathbb{Z}^{|\imath^*|+|\eta^*|}$ maps to exactly one cell in the tree. Then

$$\tau_1 \sqsubseteq_{\mathcal{T}\langle\pi\rangle} \tau_2 \iff (\tau_1 = \bot_{\mathcal{T}\langle\pi\rangle}) \ \vee \ (\tau_2 = \top_{\mathcal{T}\langle\pi\rangle}) \vee$$
$$(\imath_1^* \equiv \imath_2^* \ \wedge \ \eta_1^* \equiv \eta_2^* \ \wedge \ \forall \hat{n} \in \mathbb{Z}^{|\imath_1^*|+|\eta_1^*|}.\ \tau_1(\hat{n}) \ \sqsubseteq_\pi \tau_2(\hat{n})).$$

For $\tau_1, \tau_2 \in \mathcal{T}\langle\pi\rangle$, $\tau_1 \sqcup_{\mathcal{T}\langle\pi\rangle} \tau_2$ is $\top_{\mathcal{T}\langle\pi\rangle}$ if $(\imath_1^* \not\equiv \imath_2^*$ or $\eta_1^* \not\equiv \eta_2^*)$; otherwise it is the pointwise lift of the base join: $\forall\ \hat{n}.\ (\tau_1 \sqcup_{\mathcal{T}\langle\pi\rangle} \tau_2)(\hat{n}) = \tau_1(\hat{n}) \sqcup_\pi \tau_2(\hat{n})$. Thus, $\mathcal{T}\langle\pi\rangle$ is a complete lattice.

6.5.2 Abstraction and Concretization

We define abstraction $\alpha : \mathcal{P}(\Sigma) \to \Sigma^\#$ and concretization $\gamma : \Sigma^\# \to \mathcal{P}(\Sigma)$ functions. The soundness (for every set of concrete states $S \subseteq \Sigma$, $S \subseteq \gamma(\alpha(S))$) is proved as Theorem B.5 in Appendix B.3.

Definition 6.4. (Abstract Environment) We denote the set of abstract environments by $\Sigma^\#$. Let $\Sigma^\# \triangleq \Sigma_c^\# \times \Sigma_d^\#$ with $\Sigma_c^\# \triangleq (\mathbb{Y} \to \mathcal{T}_C)$ and $\Sigma_d^\# \triangleq (\mathbb{X} \to \mathcal{T}_D)$.

Based on the definition of the partial order $\sqsubseteq_{\mathcal{T}\langle\pi\rangle}$ over the CAD tree domain $\mathcal{T}\langle\pi\rangle$, we can introduce the partial order over $\Sigma^\#$.

Definition 6.5 (Partial Order $\sqsubseteq_{\Sigma^\#}$)**.** For $\sigma_{c1}^\#, \sigma_{c2}^\# \in \Sigma_C^\#$ and $\sigma_{d1}^\#, \sigma_{d2}^\# \in \Sigma_D^\#$, define the orders

$$\sigma_{c1}^\# \sqsubseteq_{\Sigma_C^\#} \sigma_{c2}^\# \iff \forall y \in \mathbb{Y}.\ \sigma_{c1}^\#(y) \sqsubseteq_{\mathcal{T}_C} \sigma_{c2}^\#(y), \quad \sigma_{d1}^\# \sqsubseteq_{\Sigma_D^\#} \sigma_{d2}^\# \iff \forall x \in \mathbb{X}.\ \sigma_{d1}^\#(x) \sqsubseteq_{\mathcal{T}_D} \sigma_{d2}^\#(x),$$

where $\sqsubseteq_{\mathcal{T}_C}$ and $\sqsubseteq_{\mathcal{T}_D}$ are the CAD-tree orders of Definition 6.3. Bottom/top are pointwise: $(\bot_{\Sigma_C^\#})(y) \triangleq \bot_{\mathcal{T}_C}$, $(\top_{\Sigma_C^\#})(y) \triangleq \top_{\mathcal{T}_C}$ (and analogously for $\Sigma_D^\#$). Writing $\sigma_1^\# = (\sigma_{c1}^\#, \sigma_{d1}^\#)$ and $\sigma_2^\# = (\sigma_{c2}^\#, \sigma_{d2}^\#)$ for the control/data components of environments in $\Sigma^\#$, the partial order is

$$\sigma_1^\# \sqsubseteq_{\Sigma^\#} \sigma_2^\# \iff \sigma_{c1}^\# \sqsubseteq_{\Sigma_C^\#} \sigma_{c2}^\# \ \wedge \ \sigma_{d1}^\# \sqsubseteq_{\Sigma_D^\#} \sigma_{d2}^\#,$$

with $\bot_{\Sigma^\#} \triangleq (\bot_{\Sigma_C^\#}, \bot_{\Sigma_D^\#})$ and $\top_{\Sigma^\#} \triangleq (\top_{\Sigma_C^\#}, \top_{\Sigma_D^\#})$.

Definition 6.6 (Join $\sqcup_{\Sigma^\#}$)**.** The join is defined as the pointwise lift of the tree joins: for $\sigma_i^\# = (\sigma_{ci}^\#, \sigma_{di}^\#)$,

$$\sigma_1^\# \sqcup_{\Sigma^\#} \sigma_2^\# \triangleq (y \mapsto \sigma_{c1}^\#(y) \sqcup_{\mathcal{T}_C} \sigma_{c2}^\#(y), \quad x \mapsto \sigma_{d1}^\#(x) \sqcup_{\mathcal{T}_D} \sigma_{d2}^\#(x)).$$

It is the least upper bound in $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$, with $\bot_{\Sigma^\#}$ neutral and $\top_{\Sigma^\#}$ absorbing. Thus, $\Sigma^\#$ is a complete lattice.

We write $\lceil\cdot\rceil_C : \mathbb{Z} \to \mathcal{B}_C$ and $\lceil\cdot\rceil_D : \mathbb{D} \to \mathcal{B}_D$ for lifting from concrete values to base domain values: $\lceil n \rceil_C \triangleq n \quad (n \in \mathbb{Z})$ and $\lceil d \rceil_D \triangleq d \quad (d \in \mathbb{D})$. When indices are integers we implicitly coerce them into $\mathcal{B}_C$, e.g. $\lceil x[i_1, \ldots, i_k] \rceil_D = x[\lceil i_1 \rceil_C, \ldots, \lceil i_k \rceil_C]$. For guards $g_1, \ldots, g_k$, we define the syntactic conjunction $\bigwedge_{r=1}^k g_r$ as g_1 `and` $\cdots$ `and` g_k. For a variable tuple y^*, guard g, and an abstract value π, $[\ y^* \ \mid g \Rightarrow \pi\ ] \triangleq (\ y^*,\ \langle\rangle,\ \text{LinSplit}(\langle \text{Cell}(g, \text{Leaf}(\pi, \langle\rangle))\rangle)\ )$. For any index set I and a tuple of cells $\langle [y^* \mid g_i \Rightarrow \pi_i] \rangle_{i \in I}$ we define the structural disjunction $\bigsqcup_{i \in I} [y^* \mid g_i \Rightarrow \pi_i] \triangleq \Big(\ y^*,\ \langle\rangle,\ \text{LinSplit}(\langle \text{Cell}(g_i, \text{Leaf}(\pi_i, \langle\rangle))\rangle_{i \in I})\ \Big)$. Increasing I simply adds sibling cells in the topmost LinSplit node (disjunction-as-structure).

Definition 6.7 (Abstraction α)**.** We first define a per-state abstraction $\hat{\alpha} : \Sigma \to \Sigma^\#$ pointwise, and then lift it to sets by join: $\alpha(S) \triangleq \bigsqcup_{\Sigma^\#}^{\sigma \in S} \hat{\alpha}(\sigma)$ for $S \subseteq \Sigma$. For a concrete state $\sigma = (\sigma_c, \sigma_d)$, $\hat{\alpha}(\sigma_c, \sigma_d) \triangleq (\alpha_c(\sigma_c), \alpha_d(\sigma_c, \sigma_d))$, defined pointwise.

Control. For $y \in \mathbb{Y}$ with fresh control variables $\vec{\imath} = (\imath_1, \ldots, \imath_{\text{itr}(y)})$,

$$\alpha_c(\sigma_c)(y) \triangleq \bigsqcup_{\vec{i} \in \mathbb{Z}^{\text{itr}(y)}} \left[\ \vec{\imath} \ \middle|\ \bigwedge_{r=1}^{\text{itr}(y)} (\imath_r = i_r) \ \Rightarrow\ \left\lceil \sigma_c(y)(\vec{i}) \right\rceil_C \right].$$

Data. Let $\text{Lhs}_p \subseteq (\mathbb{X} \cup \mathbb{Y})$ be the variables that are being assigned in the concrete program p. For $x \in \mathbb{X}$ with fresh $\vec{\imath} = (\imath_1, \ldots, \imath_{\text{itr}(x)})$ and $\vec{\eta} = (\eta_1, \ldots, \eta_{\text{dim}(x)})$,

$$\alpha_d(\sigma_c, \sigma_d)(x) \triangleq \begin{cases} \left[\ \vec{\eta} \ \middle|\ \text{true} \Rightarrow \lceil x[\vec{\eta}] \rceil_D \right] & \text{if } x \notin \text{Lhs}_p, \\ \bigsqcup\limits_{\substack{\vec{i} \in \mathbb{Z}^{\text{itr}(x)} \\ \vec{j} \in \mathbb{Z}^{\text{dim}(x)}}} \left[\ \vec{\imath}\vec{\eta} \ \middle|\ \left(\bigwedge_{r=1}^{\text{itr}(x)} \imath_r = i_r\right) \texttt{ and } \left(\bigwedge_{s=1}^{\text{dim}(x)} \eta_s = j_s\right) \right. \\ \qquad\qquad \left. \Rightarrow\ \left\lceil \sigma_d(x)(\vec{i}, \vec{j}) \right\rceil_D \right] & \text{if } x \in \text{Lhs}_p. \end{cases}$$

Definition 6.8 (Lifting $\mathcal{L}^\#$)**.** The lifting function $\mathcal{L}^\# : \Sigma^\# \to \Phi$, defined in Figure 6.9, traverses an abstract environment and produces a first-order formula. Writing $\Sigma^\# = (\Sigma_c^\#, \Sigma_d^\#)$,

$$\mathcal{L}^\#(\Sigma_c^\#, \Sigma_d^\#) \triangleq \Big(\bigwedge_{y \in \text{dom}(\Sigma_c^\#)} \mathcal{L}^\#_{\mathcal{T}\langle\zeta\rangle}(y, \Sigma_c^\#(y))\Big) \wedge \Big(\bigwedge_{x \in \text{dom}(\Sigma_d^\#)} \mathcal{L}^\#_{\mathcal{T}\langle\delta\rangle}(x, \Sigma_d^\#(x))\Big).$$

Definition 6.9 (Concretization γ)**.** We write $a \models b$ to mean b is true in a. We interpret an abstract environment as the set of concrete states that satisfy the formula produced by $\mathcal{L}^\#$:

$$\gamma(\Sigma^\#) \triangleq \{\sigma \in \Sigma \mid \sigma \models \mathcal{L}^\#(\Sigma^\#)\}.$$

$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle} : (\mathbb{X} \cup \mathbb{Y}) \to \mathcal{T}\langle\pi\rangle \to \Phi$$
$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\, \bot_{\mathcal{T}\langle\pi\rangle}) = \mathit{false} \quad \mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\, \top_{\mathcal{T}\langle\pi\rangle}) = \mathit{true}$$
$$\mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r,\, (\mathit{vars},\, \mathit{cuts},\, \mathit{tree})) \;=\; \mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\, \mathit{tree})$$

$$\mathcal{L}^{\#}_{n\langle\pi\rangle} : (\mathbb{X} \cup \mathbb{Y}) \to \mathit{node}\langle\pi\rangle \to \Phi$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\, \mathrm{Leaf}(\pi',\, \_)) \;=\; \mathcal{L}^{\#}_{\pi}(r,\, \pi')$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\, \mathrm{LinSplit}(\langle \mathrm{Cell}(g_i, t_i)\rangle_{i\in I})) = \bigvee_{i\in I}(\mathcal{L}^{\#}_{g}(g_i) \wedge \mathcal{L}^{\#}_{n\langle\pi\rangle}(r, t_i))$$
$$\mathcal{L}^{\#}_{n\langle\pi\rangle}(r,\, \mathrm{LinSplit}([])) \;=\; \mathit{false}$$

$$\mathcal{L}^{\#}_{g} : \mathrm{Guard} \to \Phi$$
$$\mathcal{L}^{\#}_{g}\,(\mathsf{true}) = \mathit{true} \qquad \mathcal{L}^{\#}_{g}\,(\mathsf{false}) = \mathit{false}$$
$$\mathcal{L}^{\#}_{g}\,(\zeta_1 \,\mathrm{op}\, \zeta_2) = \nu^{C}_{\mathrm{new}}\,(y_1, y_2 : \mathbb{Y}).$$
$$(\textstyle\bigwedge_{i=1}^{2} \mathcal{L}^{\#}_{\zeta}(y_i,\, \zeta_i) \wedge [\![y_1 \,\widetilde{\mathrm{op}}\, y_2]\!]) \quad \mathrm{op} \in \{==, <, \leq\}$$
$$\mathcal{L}^{\#}_{g}\,(g_1 \;\mathtt{and}\; g_2) = \mathcal{L}^{\#}_{g}(g_1) \;\wedge\; \mathcal{L}^{\#}_{g}(g_2)$$

$$\mathcal{L}^{\#}_{\zeta} : \mathbb{Y} \to \mathcal{B}_C \to \Phi$$
$$\mathcal{L}^{\#}_{\zeta}\,(r,\, n) \;=\; [\![r = n]\!] \qquad \mathcal{L}^{\#}_{\zeta}\,(r,\, y) \;=\; [\![r = y]\!]$$
$$\mathcal{L}^{\#}_{\zeta}\,(r, \zeta_1 \,\mathrm{op}\, \zeta_2) = \nu^{C}_{\mathrm{new}}\,(y_1, y_2 : \mathbb{Y}).$$
$$(\textstyle\bigwedge_{i=1}^{2} \mathcal{L}^{\#}_{\zeta}(y_i,\, \zeta_i) \wedge [\![r = y_1 \,\widetilde{\mathrm{op}}\, y_2]\!])$$
$$\mathrm{op} \in \{+, -, *, /\}$$
$$\mathcal{L}^{\#}_{\zeta}(r,\, \top_{\zeta}) \;=\; \mathit{true} \qquad \mathcal{L}^{\#}_{\zeta}(r,\, \bot_{\zeta}) \;=\; \mathit{false}$$

$$\mathcal{L}^{\#}_{\delta} : \mathbb{X} \to \mathcal{B}_D \to \Phi$$
$$\mathcal{L}^{\#}_{\delta}(r,\, d) \;=\; [\![\, r = d \,]\!]$$
$$\mathcal{L}^{\#}_{\delta}(r,\, x[\zeta_1, \ldots, \zeta_k]) = \nu^{C}_{\mathrm{new}}\,(y_1, \ldots, y_k \;: \mathbb{Y}).$$
$$\Big(\textstyle\bigwedge_{i=1}^{k} \mathcal{L}^{\#}_{\zeta}(y_i,\, \zeta_i)\Big) \wedge [\![\, r = \nu^{D}_{\mathrm{new}}(x,\, y_1, \ldots, y_k) \,]\!]$$
$$\mathcal{L}^{\#}_{\delta}(r,\, \top_{\delta}) \;=\; \mathit{true} \qquad \mathcal{L}^{\#}_{\delta}(r,\, \bot_{\delta}) \;=\; \mathit{false}$$

Figure 6.9: Lifting functions $\mathcal{L}^{\#}$ from CAD trees to first-order logic. $[\![\ldots]\!]$ denotes the code quote of the first-order-logic formula expressing that $r \in \mathbb{X} \cup \mathbb{Y}$ equals the given value or expression. $\nu^{D}_{\mathrm{new}} : \mathbb{X} \times \mathbb{Y}^* \to \mathbb{X}$ is an injective renaming from array-index tuples to variables in $\mathbb{X}$. $\nu^{C}_{\mathrm{new}}\,(\vec{y} : \mathbb{Y})$. introduces fresh control variables.

Example 6.2. Suppose we have a variable $x \in \mathbb{X}$ and an abstract value $\tau'_\delta = (\mathrm{vars} : \langle n\rangle, \mathrm{cuts} : \langle n\rangle, \mathrm{tree} : \mathrm{LinSplit}\langle \mathrm{Cell}(n < 0, \mathrm{Leaf}(\bot_\delta, \langle -1\rangle)),$ $\mathrm{Cell}(n = 0,\, \mathrm{Leaf}(1.0, \langle 0\rangle)), \mathrm{Cell}(n > 0,\, \mathrm{Leaf}(\top_\delta, \langle 1\rangle))\rangle)$. This tree is visualized as the inset figure on the right. The corresponding formulas obtained by lifting function $\mathcal{L}^{\#}_{\mathcal{T}\langle\delta\rangle}(x, \tau'_\delta)$ is:

```
(var n)
| n < 0 ⟨n=-1⟩ bot
| n = 0 ⟨n=0⟩  1.0
| n > 0 ⟨n=1⟩  top
```

$$(n < 0)\ ?\ \mathrm{false} : (n = 0\ ?\ x == 1.0 : \mathrm{true}).$$

The lifting process traverses the CAD tree, converting each branch condition and leaf value into a Boolean expression. Bottom leaves ($\bot_\pi$) lift to false (infeasibility), top leaves ($\top_\pi$) lift to true (all possible values), and concrete scalars like 1.0 lift to equality formulas $[\![x == 1.0]\!]$. The tree structure naturally translates to nested conditional expressions, preserving the logical flow of the original abstract value while converting it to formulas.

6.5.3 Transfer Function

We have constructed three lattices, namely $(\mathcal{T}_C, \sqsubseteq_{\mathcal{T}_C})$, $(\mathcal{T}_D, \sqsubseteq_{\mathcal{T}_D})$, and $(\Sigma^{\#}, \sqsubseteq_{\Sigma^{\#}})$. Based on them, we formulate the abstract semantics of Prexo IR through transfer functions that update abstract environment, faithfully aligning with the concrete semantics in Figure 6.6. We introduce *evaluation functions* $A^{\#}, B^{\#}, C^{\#}, D^{\#}$ that evaluate Prexo IR expressions over the abstract environment $\sigma^{\#}$ (Figure 6.10). $A^{\#}$ extracts affine forms from control expressions, $B^{\#}$ turns Boolean guards into tuples for cuts, and $C^{\#}/D^{\#}$ lift control/data expressions

$$
\begin{array}{l}
A^{\#} : \mathrm{CExpr} \to \Sigma^{\#} \to \mathcal{B}_C \\
\qquad A^{\#}\,[\![n]\!]\,\sigma^{\#} = n \quad A^{\#}\,[\![y]\!]\,\sigma^{\#} = y \quad A^{\#}\,[\![y[c_1,\ldots,c_n]]\!]\,\sigma^{\#} = \top_\zeta \\
\qquad A^{\#}\,[\![c_1 \,\mathtt{op}\, c_2]\!]\,\sigma^{\#} = A_1, A_2 \notin \{\bot_\zeta, \top_\zeta\}\ \mathtt{?}\ A_1 \,\widetilde{\mathtt{op}}\, A_2 : \top_\zeta, \quad \text{where } A_i = A^{\#}\,[\![c_i]\!]\,\sigma^{\#} \\
\hline
B^{\#} : \mathrm{BExpr} \to \Sigma^{\#} \to \zeta^* \\
\qquad B^{\#}\,[\![t]\!]\,\sigma^{\#} = \langle\,\rangle \quad B^{\#}\,[\![b_1 \,\mathtt{and}\, b_2]\!]\,\sigma^{\#} = \Pi_1 \uplus \Pi_2 \\
\qquad B^{\#}\,[\![b_1 \,\mathtt{or}\, b_2]\!]\,\sigma^{\#} = \Pi_1 \uplus \Pi_2, \quad \text{where } \Pi_i = B^{\#}\,[\![b_i]\!]\,\sigma^{\#} \\
\qquad B^{\#}\,[\![c_1 \bowtie c_2]\!]\,\sigma^{\#} = A_1, A_2 \notin \{\bot_\zeta, \top_\zeta\}\ \mathtt{?}\ \langle\, A_1 - A_2 \,\rangle : \langle \top_\zeta \rangle, \\
\qquad\qquad \text{where } A_i = A^{\#}\,[\![c_i]\!]\,\sigma^{\#} \text{ and } \bowtie \in \{==, <, \leq\} \\
\hline
C^{\#} : \mathrm{CExpr} \to \Sigma^{\#} \to \mathcal{T}_C \\
\qquad C^{\#}\,[\![n]\!]\,\sigma^{\#} = [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow n\ ] \quad C^{\#}\,[\![y]\!]\,\sigma^{\#} = y \in \mathrm{Lhs}_p\ \mathtt{?}\ \sigma^{\#}(y) : [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow y\ ] \\
\qquad C^{\#}\,[\![c_1 \,\mathtt{op}\, c_2]\!]\,\sigma^{\#} = A \notin \{\bot_\zeta, \top_\zeta\}\ \mathtt{?}\ [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow A] : \top_{\mathcal{T}_C}, \quad \text{where } A = A^{\#}\,[\![c_1 \,\mathtt{op}\, c_2]\!]\,\sigma^{\#} \\
\qquad C^{\#}\,[\![y[c_1,\ldots,c_n]]\!]\,\sigma^{\#} = \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\}\ \mathtt{?}\ (y^{\mathrm{itr}*},\, \theta.\mathrm{cuts},\, \theta.\mathrm{tree}) : \top_{\mathcal{T}_C}, \\
\qquad\qquad \text{where } \big(y^{\mathrm{itr}*},\, \mathrm{cuts},\, \mathrm{tree}\big) = \sigma^{\#}(y), \quad \theta = [\, y_k^{\mathrm{itr}} \mapsto A_k \,]_{k=1}^{n} \text{ and } A_i = A^{\#}\,[\![c_i]\!]\,\sigma^{\#} \\
\hline
D^{\#} : \mathrm{DExpr} \to \Sigma^{\#} \to \mathcal{T}_D \\
\qquad D^{\#}\,[\![d]\!]\,\sigma^{\#} = [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow d\ ] \quad D^{\#}\,[\![x]\!]\,\sigma^{\#} = x \in \mathrm{Lhs}_p\ \mathtt{?}\ \sigma^{\#}(x) : [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow x\ ] \\
\qquad D^{\#}\,[\![e_1 \,\mathtt{op}\, e_2]\!]\,\sigma^{\#} = \top_{\mathcal{T}_D} \\
\qquad D^{\#}\,[\![x[c_1,\ldots,c_n]]\!]\,\sigma^{\#} = \begin{cases} (y^{\mathrm{itr}*},\, \theta.\mathrm{cuts},\, \theta.\mathrm{tree}), & x \in \mathrm{Lhs}_p \text{ and } \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\} \\ [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow x[A_1,\ldots,A_n]\ ], & \forall k.A_k \notin \{\bot_\zeta, \top_\zeta\} \\ [\ \langle\,\rangle \ \mid \mathrm{true} \Rightarrow \top_\delta\ ], & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } \big(y^{\mathrm{itr}*},\, \mathrm{cuts},\, \mathrm{tree}\big) = \sigma^{\#}(x), \quad \theta = [\, y_k^{\mathrm{itr}} \mapsto A_k \,]_{k=1}^{n} \text{ and } A_i = A^{\#}\,[\![c_i]\!]\,\sigma^{\#}
\end{array}
$$

Figure 6.10: Abstract semantics of Prexo IR expressions. $\uplus$ denotes tuple concatenation, and Lhs_p denotes the set of program variables assigned in p. The notation $a?b:c$ is a shorthand for a case split: "if a then b else c".

to the control/data tree domains $\mathcal{T}_C$ and $\mathcal{T}_D$. Write Lhs_p for the set of program variables that are assigned in the procedure p. The environment $\sigma^{\#}$ stores abstract values only for variables in Lhs_p, since procedure parameters in Exo IR are free variables in Prexo IR. This explains the cases $y \in \mathrm{Lhs}_p$ and $x \in \mathrm{Lhs}_p$ in Figure 6.10: if a variable is in Lhs_p we read its tracked abstract value from $\sigma^{\#}$ (so updates are reflected); if it is not, we produce a leaf that symbolically represents the corresponding free variable. For control variables y, $C^{\#}$ therefore either looks up $\sigma^{\#}(y)$ (when $y \in \mathrm{Lhs}_p$) or emits a leaf $[\langle\rangle \mid \mathrm{true} \Rightarrow y]$. For data variables x, $D^{\#}$ behaves analogously on scalars and on arrays distinguishes two cases: (i) when $x \in \mathrm{Lhs}_p$, the read substitutes the concrete indices into the abstract tree stored at $\sigma^{\#}(x)$ (via the substitution θ); (ii) when $x \notin \mathrm{Lhs}_p$ (an input array parameter), a read $x[c_1,\ldots,c_n]$ becomes a symbolic value $x[A_1,\ldots,A_n]$ whenever the indices are affine ($A_i = A^{\#}[\![c_i]\!]\,\sigma^{\#}$), thereby preserving relational information among different accesses to the same input array; if the indices are not affine we return a $\top_\delta$.

The *transfer function* $S^{\#} : \mathtt{Stmt} \to \Sigma^{\#} \to \Sigma^{\#}$ is defined in Figure 6.11, which updates the environment by building a CAD over variables and cuts, then resolving leaves by the guard. As shown in Figure 6.11, for control $y = \lambda\imath^*.(b?c_t : c_f)$ and data $x = \lambda\imath^*.\lambda\eta^*.(b?e_t : e_f)$ assignments, we evaluate the branches to CAD-trees ($\tau_t = C^{\#}\,[\![c_t]\!]\,\sigma^{\#}$ or $D^{\#}\,[\![e_t]\!]\,\sigma^{\#}$; similarly τ_f) and transfer the guard to *cuts* $\Pi = B^{\#}\,[\![b]\!]\,\sigma^{\#}$. We then form $\tau_{\mathrm{new}} = \mathrm{CAD}_{\bot}^{\top}(\mathrm{vars}, \mathrm{cuts})$ with $\mathrm{vars} = \tau_t.\mathrm{vars} \uplus \tau_f.\mathrm{vars} \uplus \imath^*$ (and $\uplus\,\eta^*$ for data) and $\mathrm{cuts} = \tau_t.\mathrm{cuts} \uplus \tau_f.\mathrm{cuts} \uplus \Pi$; each leaf

$$
\begin{aligned}
&S^{\#} : \mathsf{Stmt} \to \Sigma^{\#} \to \Sigma^{\#} \\
&\qquad S^{\#}\,[\![s_1; s_2]\!]\,\sigma^{\#} \;=\; S^{\#}\,[\![s_2]\!]\,(S^{\#}\,[\![s_1]\!]\,\sigma^{\#}) \\
&S^{\#}\,[\![y = \lambda\imath^*.(\,b\,?\,c_t : c_f\,)]\!]\,\sigma^{\#} \;=\; \sigma^{\#}[\,y \mapsto {\tau^{y}_{\text{new}}}'\,] \\
&\qquad \tau_t = C^{\#}\,[\![c_t]\!]\,\sigma^{\#}, \quad \tau_f = C^{\#}\,[\![c_f]\!]\,\sigma^{\#}, \quad \Pi = B^{\#}\,[\![b]\!]\,\sigma^{\#} \\
&\qquad \tau^{y}_{\text{new}} = \mathrm{CAD}^{\top}_{\bot}(\text{vars}, \text{cuts}), \quad \text{vars} = \tau_t.\text{vars} \uplus \tau_f.\text{vars} \uplus \imath^*, \quad \text{cuts} = \tau_t.\text{cuts} \uplus \tau_f.\text{cuts} \uplus \Pi \\
&\qquad {\tau^{y}_{\text{new}}}' = \tau^{y}_{\text{new}}[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v, \hat{n})], \\
&\qquad\qquad \text{where } v = \text{if } \mathcal{E}_b\,[\![b]\!]\,(\{\imath_k \mapsto \hat{n}_k\}) \text{ then } \tau_t(\hat{n}) \text{ else } \tau_f(\hat{n}) \\
&S^{\#}\,[\![x = \lambda\imath^*.\lambda\eta^*.(\,b\,?\,e_t : e_f\,)]\!]\,\sigma^{\#} \;=\; \sigma^{\#}[\,x \mapsto {\tau^{x}_{\text{new}}}'\,] \\
&\qquad \tau_t = D^{\#}\,[\![e_t]\!]\,\sigma^{\#}, \quad \tau_f = D^{\#}\,[\![e_f]\!]\,\sigma^{\#}, \quad \Pi = B^{\#}\,[\![b]\!]\,\sigma^{\#} \\
&\qquad \tau^{x}_{\text{new}} = \mathrm{CAD}^{\top}_{\bot}(\text{vars}, \text{cuts}), \\
&\qquad\qquad \text{vars} = \tau_t.\text{vars} \uplus \tau_f.\text{vars} \uplus \imath^* \uplus \eta^*, \quad \text{cuts} = \tau_t.\text{cuts} \uplus \tau_f.\text{cuts} \uplus \Pi \\
&\qquad {\tau^{x}_{\text{new}}}' = \tau^{x}_{\text{new}}[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v, \hat{n})], \\
&\qquad\qquad \text{where } v = \text{if } \mathcal{E}_b\,[\![b]\!]\,(\{\imath_k \mapsto \hat{n}_k\}) \text{ then } \tau_t(\hat{n}) \text{ else } \tau_f(\hat{n})
\end{aligned}
$$

Figure 6.11: Abstract transfer $S^{\#}$ for the Prexo IR. We define $\mathrm{CAD}^{\top}_{\bot}$ as an extension of the standard CAD: it returns $\top_{\mathcal{T}\langle\pi\rangle}$ if any cuts are $\bot_\zeta$ or $\top_\zeta$, and otherwise follows the normal CAD algorithm. Leaf values are initialized to $\top_\zeta$ or $\top_\delta$ by default. The notation $\tau(\hat{n})$ is introduced in Definition 6.3.

at a sample point $\hat{n}$ is set to $\tau_t(\hat{n})$ if $\mathcal{E}_b\,[\![b]\!]\,(\hat{n})$ else $\tau_f(\hat{n})$. According to the above definitions of the evaluation functions and the transfer function $S^{\#}$, we prove the soundness of the transfer function $S^{\#}$ in Theorem B.6 in Appendix B.4.

Definition 6.10 (Abstract transfer on programs)**.** For a program $p = \mathrm{Fix}\,s$ and an abstract initial store $\sigma^{\#0} \in \Sigma^{\#}$, we define the *abstract program transfer* $P^{\#}\,[\![p]\!]\,(\sigma^{\#0}) = \sigma^{\#*}$ by the fixed point

$$
\sigma^{\#*} \;=\; \mathrm{lfp}(F^{\#}), \quad \text{where} \quad F^{\#}(\sigma^{\#}) \;\triangleq\; \sigma^{\#0} \;\sqcup_{\Sigma^{\#}}\; S^{\#}\,[\![s]\!]\,(\sigma^{\#})
$$

Since $\Sigma^{\#}$ is a complete lattice and $S^{\#}$ is monotone, $F^{\#}$ is monotone; hence the fixed points exist by Tarski–Knaster. We prove the soundness of the transformer $P^{\#}$ in Theorem B.9 in Appendix B.5.

6.5.4 Widening

In this subsection, we introduce an efficient "flood-fill" widening operator $\nabla : \Sigma^{\#} \times \Sigma^{\#} \to \Sigma^{\#}$ that *accelerates* the fixpoint computation of the abstract program transfer (Definition 6.10). The operator safely extrapolates joins, $(\sigma^{\#}_a \sqcup_{\Sigma^{\#}} \sigma^{\#}_b) \;\sqsubseteq_{\Sigma^{\#}}\; (\sigma^{\#}_a \nabla \sigma^{\#}_b)$ for all $\sigma^{\#}_a, \sigma^{\#}_b \in \Sigma^{\#}$, and it enforces finite convergence: for any initial abstract state $\widetilde{\sigma}^{\#}_0 \in \Sigma^{\#}$, a sequence $\widetilde{\sigma}^{\#}_{i+1} = \widetilde{\sigma}^{\#}_i \nabla F^{\#}(\widetilde{\sigma}^{\#}_i)$ is an ascending chain and stabilizes after finitely many steps.

Definition 6.11 (Widening $\nabla : \Sigma^{\#} \times \Sigma^{\#} \to \Sigma^{\#}$)**.** $X \nabla Y \;\triangleq\; \mathsf{Floodfill}(\,X \;\sqcup_{\Sigma^{\#}}\; Y\,)$.

To build intuition for the floodfill algorithm used in our widening, we present a concrete example in Figure 6.12. The figure demonstrates the analysis of a simple one-dimensional

```
for i in seq(0, n):
    x[i] = 0.0
```

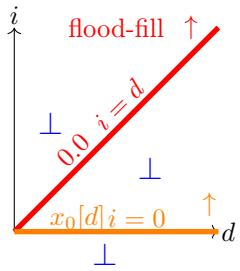
(a) Input code in Exo IR form

```
x1 = \i\d(i==0 ? x0[d] : x2[i-1,d])
x2 = \i\d(i==d ? 0.0 : x1[i,d])
x3 = \d(n > 0 ? x2[n-1,d] : x[d])
```

(b) Prexo IR form

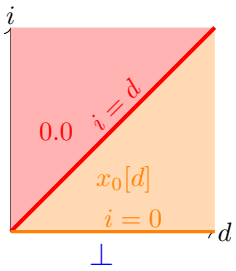
$$\mathrm{x}_3(d_0) = \begin{cases} 0, & n \geq 1 \wedge 0 \leq d_0 < n, \\ \mathrm{x}_0(d_0), & \text{otherwise.} \end{cases}$$

(c) Output from the analysis




```
(var i, d)
|  d < 0
|  |  i < d ⟨d=-1, i=-2⟩ bot
|  |  -d + i = 0 ⟨d=-1, i=-1⟩ 0.0
|  |  i < 0 and -d+i > 0 ⟨d=-1, i=-1/2⟩ bot
|  |  i = 0 ⟨d=-1, i=0⟩ x0[d]
|  |  i > 0 ⟨d=-1, i=1⟩ bot
|  -d = 0
|  |  i < d ⟨d=0, i=-1⟩ bot
|  |  -d + i = 0 ⟨d=0, i=0⟩ 0.0
|  |  i > d ⟨d=0, i=1⟩ bot
|  d > 0
|  |  i < 0 ⟨d=1, i=-1⟩ bot
|  |  i = 0 ⟨d=1, i=0⟩ x0[d]
|  |  i > 0 and -d + i < 0 ⟨d=1, i=1/2⟩ bot
|  |  -d + i = 0 ⟨d=1, i=1⟩ 0.0
|  |  i > d ⟨d=1, i=2⟩ bot
```

(d) Before floodfill




```
(var i, d)
|  d < 0
|  |  i < d ⟨d=-1, i=-2⟩ bot
|  |  -d + i = 0 or i < 0
|  |       and -d + i > 0 ⟨d=-1, i=-1⟩ 0.0
|  |  i = 0 or i > 0 ⟨d=-1, i=0⟩ x0[d]
|  -d = 0
|  |  i < d ⟨d=0, i=-1⟩ bot
|  |  i > d or -d + i = 0 ⟨d=0, i=0⟩ 0.0
|  d > 0
|  |  i < 0 ⟨d=1, i=-1⟩ bot
|  |  i = 0 or i > 0
|  |       and -d + i < 0 ⟨d=1, i=0⟩ x0[d]
|  |  i > d or -d + i = 0 ⟨d=1, i=1⟩ 0.0
```

(e) After floodfill

Figure 6.12: Visualization of the floodfill widening on Array Init 1D example.

array-initialization program. Figure 6.12(a) shows the original Exo IR code, which initializes elements of an array x to zero for indices 0 to $n-1$. This code is transformed into Prexo IR shown in Figure 6.12(b), where i denotes the loop iterator, d represents the array dimensions, and `x0` is the initial value of x at the beginning of the procedure. `x1` through `x3` represent different versions of the same array variable x at distinct program points: `x1` corresponds to the loop start, `x2` to the assignment, and `x3` to the loop exit node. The analysis produces the final abstract-interpretation output shown in Figure 6.12(c), which precisely captures that array elements in the range $[0, n)$ are set to zero when $n \geq 1$. Figure 6.12(d) illustrates the abstract domain value for x_2 after the first fixpoint iteration, with both a visualization (top diagram) and the corresponding CAD-tree representation (bottom diagram). In the visualization, the red line represents cells where $i = d$ with value 0.0, while the orange line represents cells where $i = 0$ with value $x_0[d]$; all other cells contain $\bot_\delta$. The flood-fill operation (shown as arrows) then propagates these values "upward in time", from *sections* to *sectors* immediately following them, resulting in the filled regions shown in Figure 6.12(e).

```
for i in seq(0, 10):
    x[i] = 0.0
#  ..................{x_3 : τ_3}
for i in seq(0, 10):
    x[i] = 0.0
#  ..................{x_6 : τ_6}
...
```

(a) Exo IR code where s can be removed.

```
for i in seq(0, 10):
    x[i] = -x[i]
#  ..................
for i in seq(0, 10):
    x[i] = x[i] + 2.0
#  ..................
...
```

(b) Exo IR where s cannot be removed.

```
x_1 =\i \d0 (i == 0 ? x[d0] : x_2[i - 1, d0]) # LoopStart
x_2 =\i \d0 (i == d0 ? 0.0 : x_1[i, d0])
x_3 =\d0 (10 > 0 ? x_2[10 - 1, d0] : x[d0]) # LoopExit
x_4 =\i_1 \d0 (i_1 == 0 ? x_3[d0] : x_5[i_1 - 1, d0]) # LoopStart
x_5 =\i_1 \d0 (i_1 == d0 ? 0.0 : x_4[i_1, d0])
x_6 =\d0 (10 > 0 ? x_5[10 - 1, d0] : x_3[d0]) # LoopExit
...
```

(c) Prexo IR of (a)

Figure 6.13: Deleting the second loop in (a) is correct, while (b) is incorrect. $\{x_3 : \tau_3\}$ and $\{x_6 : \tau_6\}$ represent Prexo IR program environment (variable-abstract value pair) at that program location.

Definition 6.12 (Flood-fill)**.** Fix $\pi \in \{\zeta, \delta\}$ and the flat base domain $\mathcal{B}_\pi$. For $(\sigma_c^\#, \sigma_d^\#) \in \Sigma^\#$ define pointwise $\mathsf{Floodfill}(\sigma_c^\#, \sigma_d^\#) \triangleq (\, y \mapsto \mathsf{Floodfill}_t(\sigma_c^\#(y)) \,,\ x \mapsto \mathsf{Floodfill}_t(\sigma_d^\#(x)) \,)$, where $\mathsf{Floodfill}_t : \mathcal{T}\langle\pi\rangle \to \mathcal{T}\langle\pi\rangle$ is the node-level flood-fill defined as follows. Let $\mathsf{Floodfill}_t(\bot_{\mathcal{T}\langle\pi\rangle}) = \bot_{\mathcal{T}\langle\pi\rangle}$ and $\mathsf{Floodfill}_t(\top_{\mathcal{T}\langle\pi\rangle}) = \top_{\mathcal{T}\langle\pi\rangle}$. For $\tau = (\mathrm{vars} : \imath^*\eta^*,\ \mathrm{cuts} : \zeta^*,\ \mathrm{tree} : n)$, define

$$\mathsf{Floodfill}_t(\tau) \triangleq (\mathrm{vars} : \imath^*\eta^*,\ \mathrm{cuts} : \zeta^*,\ \mathrm{tree} : m) \quad \text{where} \quad (m, \_) = \mathsf{Floodfill}_n(n,\ \bot_\pi).$$

Define $\mathsf{Floodfill}_n :\ node\langle\pi\rangle \times \mathcal{B}_\pi \ \to\ node\langle\pi\rangle \times \mathcal{B}_\pi$ as

$$\begin{aligned}
\mathsf{Floodfill}_n(\mathrm{Leaf}(\ell, \mathbf{s}),\ c) &\triangleq (\mathrm{Leaf}(\ell', \mathbf{s}),\ \ell') \quad \text{with} \quad \ell' = (\ell = \bot_\pi)\ ?\ c : \ell \\
\mathsf{Floodfill}_n(\mathrm{LinSplit}((g_i, n_i)_{i=1}^k), c) &\triangleq (\ \mathrm{LinSplit}(\mathrm{Cell}(g_1, m_1), \ldots, \mathrm{Cell}(g_k, m_k))\ ,\ c_k\ ).
\end{aligned}$$

where $c_0 \triangleq c$ and $(m_i, c_i) \triangleq \mathsf{Floodfill}_n(n_i, c_{i-1})\ \ (i = 1, \ldots, k)$. $\mathsf{Floodfill}_t$ preserves `vars`/`cuts` and overwrites only $\bot_\pi$-payload leaves in the tree; sample points $\mathbf{s}$ are preserved.

Theorems B.12 and B.15 in Appendix B.6 establish that ∇ soundly over-approximates the join and that the iteration $\widetilde{\sigma}_{i+1}^\# = \widetilde{\sigma}_i^\# \,\nabla\, F^\#(\widetilde{\sigma}_i^\#)$ stabilizes in finitely many steps.

6.6 Integrating Abstract Values into an Imperative USL

6.6.1 Analysis API

This subsection introduces the analysis API `Check_Delete(p, s)`, which determines whether a statement s can be safely removed from program p. Consider the examples in

Figures 6.13(a) and 6.13(b). In (a), loop statement s writes 0.0 to array x, duplicating the value already written by the previous loop. Since this is redundant, statement s can be safely removed. In contrast, (b) shows a case where s assigns a different value to x, making deletion unsafe as it would alter program behavior.

Figure 6.13(c) shows the Prexo IR of example (a). In this representation, x_3 denotes the program variable for the buffer x before executing statement s, while x_6 denotes the variable after s. The corresponding abstract values are τ_3 for x_3 and τ_6 for x_6. This can be expressed using Hoare-logic notation as $\{x_3 : \tau_3\}\ s\ \{x_6 : \tau_6\}$, where the pre- and postconditions represent the program variable-abstract value pairs before and after executing s (as illustrated in Figure 6.13(a)). To determine whether a statement s can be safely deleted, we must verify that all buffers modified by s retain identical values before and after its execution. Since statement s modifies only buffer x, the safety check reduces to verifying whether the program variables x_3 and x_6 are equivalent.

To determine whether the program variables x_3 and x_6 are equivalent, we compare their abstract values τ_3 and τ_6 using the lifting function $\mathcal{L}^{\#}$ (Definition 6.8). We require that for every Exo IR buffer modified in s, where $r_{\text{pre}} \in \mathbb{X} \cup \mathbb{Y}$ and $r_{\text{post}} \in \mathbb{X} \cup \mathbb{Y}$ denote the Prexo IR program variable before and after s, and where corresponding abstract values are given by $\tau_{\text{pre}} = \sigma^{\#}[\![r_{\text{pre}}]\!]$ and $\tau_{\text{post}} = \sigma^{\#}[\![r_{\text{post}}]\!] \in \mathcal{T}_{\langle\pi\rangle}$, the following condition must hold: if both $\mathcal{L}^{\#}(r_{\text{pre}}, \tau_{\text{pre}})$ and $\mathcal{L}^{\#}(r_{\text{post}}, \tau_{\text{post}})$ hold, then the abstract value must be unchanged by s, that is, $r_{\text{pre}} = r_{\text{post}}$. Formally, the safety condition of `Check_Delete` can be written as:

$$\mathcal{L}^{\#}(r_{\text{pre}}, \tau_{\text{pre}}) \wedge \mathcal{L}^{\#}(r_{\text{post}}, \tau_{\text{post}}) \implies (r_{\text{pre}} = r_{\text{post}}).$$

Applying this to our deletion problem: if x_3 and x_6 are equivalent under this definition, then statement s produces no observable change to variable x, making it a candidate for safe deletion. This condition ensures that if the pre- and post-abstract values of x concretize to the same concrete values, then deleting s will not alter the value of x in the concrete environment. We use `Check_Delete(p, s)` to denote that the formula above is verifiable for a procedure p and a statement s.

In our implementation, $\mathcal{L}^{\#}$ translates CAD trees into first-order formulas that we discharge to an SMT solver. By construction, the image of $\mathcal{L}^{\#}$ stays within linear arithmetic: control expressions are closed under affine operators (no variable-variable products; divisions are by constants), guards compare affine terms with $\{=, <, \leq\}$ and combine by conjunction while LinSplit yields only finite disjunctions, and array reads $x[\zeta^*]$ are renamed to fresh scalars via the injective ν^D_{new}, eliminating arrays/UFs. This yields a formula in QF-LIA, for which SMT solvers provide complete decision procedures; hence every query we emit is decidable.

6.6.2 Scheduling Operations

Scheduling in the USL context refers to a set of meta-programming operators that transform the object code, while preserving semantic equivalence. On top of the existing scheduling operations of Section 4.2 and Chapter 5, Prexo introduces three scheduling operations that leverage value-sensitive analysis to enable transformations that were unsupported or unsafe in Exo. Figure 6.14 shows these operations: `delete`, `insert`, and `bind`. The `delete` operation removes a statement `s` from program point `p` when `Check_Delete(p, s)`

Scheduling operation	Code transformation	Safety conditions

`delete(p, s)`	`s1`
	`s` ⇝ `s1`
	`s2` ⇝ `s2`
		`Check_Delete(p, s)` holds.

`insert(p, s)`	`s1` ⇝ `s1`
	`s2` ⇝ `s`
	`s2`
		`Check_Delete(p',s)` holds, where `p'` = `p[s1;s2` ↦ `s1;s;s2]`.

`bind(p, e, v)`	`s` ⇝ `v=e` `s[e↦v]`	`Check_Delete(p',v=e)` holds, where `p'` = `p[s` ↦ `v=e;s]`.

Figure 6.14: The list of new scheduling operators introduced by Prexo.

Micro-benchmarks	Traits	Prexo	#vars	#cuts	#cells	TVLA-Arr	Prexo-NW
a. Simple Conditional	br	0.16s	2	4	28	–	0.15s
b. Array Init 1D [59–61]	sl	0.61s	2	6	136	1.7s	0.51s •
c. Array Init 1D Multi-loop	ml	13.78s	3	26	1,511	–	4.52s •
d. Array Init 2D Const	ml,md	81.05s	4	24	17,190	–	56.82s •
e. Array Init 2D Non-Const [60]	ml,md	1,243.3s	5	24	264,331	–	816.44s •
f. Array Copy [59,60,67]	ar	2.3s	2	11	238	338.1s	0.61s •
g. Sliding Window Const	ml,sw	38.85s	3	21	8,161	–	28.45s •
h. Sliding Window Guard	ml,sw,br	172.38s	3	41	24,779	–	146.34s •
i. Sliding Window Non-Const	ml,sw	714.79s	4	21	145,314	–	467.46s •
j. Array Loop Fuse	af	1.16s	2	15	192	–	0.98s •
k. Array Slice Init [60]	sl	8.03s	3	6	2895	–	5.39s •
l. Array Reverse Const-Write	ra	50.53s	3	9	13123	–	42.48s •
m. Array Reverse Array-Write	ra,ar	17.97s	2	18	1354	–	2.73s •

Table 6.1: The statistics of Prexo across microbenchmarks.

holds. As demonstrated in Section 6.6.1, this safety condition verifies that no subsequent computations depend on values produced by `s`, ensuring the removal preserves program semantics. The `insert` operation adds a statement `s` at program point `p`, with safety condition `Check_Delete(p', s)` where `p'` represents the transformed program containing `s`. This condition reuses the deletion check to ensure the inserted statement is redundant (i.e., without altering the program semantics) and therefore safe to add. The `bind` operation introduces a variable binding `v = e` and replaces all occurrences of expression `e` with `v`. It requires that the binding itself be deletable in the resulting program. These operations enable scheduling transformations while maintaining semantic equivalence through value-sensitive analysis, pushing the boundary of safe program transformations with imperative USLs.

6.7 Evaluation

6.7.1 Micro-Benchmark Evaluation

Evaluation Setting We evaluated the expressiveness and runtime performance of Prexo on microbenchmarks (Figure 6.15, Table 6.1) drawn from prior work [59–61,67] as well as manually crafted programs. Figure 6.15 shows the input Exo IR and ground-truth array contents at program termination. These programs exhibit diverse features that challenge

static analyzers: branches (br) require state splitting and joining; single/multi-loops (sl, ml) require widening operators for termination; multi-loops (ml) and multi-dimensional arrays (md) induce case explosion in the number of symbolic cases that the analysis must track; array relations (ar) require reasoning about dependencies among multiple unbounded arrays with unknown initial values; array loop fusion (af) necessitates distinguishing overlapping writes across sequential loops; sliding-window access (sw), common in image processing and stencil computations, demands precise reasoning to verify the functional equivalence between guarded and unguarded loop bodies. Finally, reverse access (ra) requires introducing new partitions like $\lfloor N/2 \rfloor$, motivating our CAD abstract domain design.

```
x[0] = 2.0
if n < 3:
    x[n] = 3.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 3, & n < 3 \wedge d_0 = n, \\ 2, & d_0 = 0 \wedge n \neq 0, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(a) Simple Conditional

```
for i in seq(0, n):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 0, & n \geq 1 \wedge 0 \leq d_0 < n, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(b) Array Init 1D [59–61]

```
for i in seq(0, 10):
    for j in seq(0, 20):
        x[i] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq 9, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(c) Array Init 1D Multi-loop

```
for i in seq(0, 10):
    for j in seq(0, 20):
        x[i, j] = 0.0
```

$$\mathrm{x_o}(d_0, d_1) = \begin{cases} 0, & 0 \leq d_0 < 10 \\ & \wedge\, 0 \leq d_1 < 20, \\ \mathrm{x_i}(d_0, d_1), & \text{otherwise.} \end{cases}$$

(d) Array Init 2D Const

```
for i in seq(0, n):
    for j in seq(0, m):
        x[i, j] = 0.0
```

$$\mathrm{x_o}(d_0, d_1) = \begin{cases} 0, & 0 \leq d_0 < n \\ & \wedge\, 0 \leq d_1 < m, \\ \mathrm{x_i}(d_0, d_1), & \text{otherwise.} \end{cases}$$

(e) Array Init 2D Non-Const [60]

```
for i in seq(0, n):
    x[i] = y[i]
```

$$\mathrm{x_o}(d_0) = \begin{cases} \mathrm{y}(d_0), & n > 0 \wedge 0 \leq d_0 < n, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(f) Array Copy [59,60,67]

```
for i in seq(0,10):
  for j in seq(0,20):
    x[i + j] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq 28, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(g) SW Const

```
for i in seq(0,10):
  for j in seq(0,20):
    if i==0 or j==19:
      x[i + j] = 2.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 2, & d_0 \leq n + m - 2, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(h) SW Guard

```
for i in seq(0, n):
    for j in seq(0, m):
        x[i + j] = 2.0
```

(i) SW Non-Const

```
for i in seq(0, 10):
    x[i] = 2.0
for i in seq(0, 5):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0) = \begin{cases} 0, & d_0 \leq 4, \\ 2, & 4 < d_0 \leq 9, \\ \mathrm{x_i}(d_0), & \text{otherwise.} \end{cases}$$

(j) Array Loop Fuse

```
for i in seq(lo, hi):
    x[i] = 0.0
```

$$\mathrm{x_o}(d_0, lo, hi) = \begin{cases} 0.0, & lo \leq d_0 < hi, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(k) Array Slice Init [60]

```
for i in seq(0, N):
    x[N - i - 1] = 3.0
    x[i] = 1.0
```

$$\mathrm{x_o}(d_0, N) = \begin{cases} 3.0, & N > 0 \wedge 0 \leq d_0 < \lfloor N/2 \rfloor, \\ 1.0, & N > 0 \wedge \lfloor N/2 \rfloor \leq d_0 < N, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(l) Array Reverse Const-Write

```
for i in seq(0, 11):
    x[10-i] = y[10-i]
    x[i] = y[i]
```

$$\mathrm{x_o}(d_0) = \begin{cases} \mathrm{y}[d_0], & 0 \leq d_0 \leq 10, \\ \mathrm{x_i}[d_0], & \text{otherwise.} \end{cases}$$

(m) Array Reverse Array-Write

Figure 6.15: The micro-benchmark programs and the array contents at the last program locations ($\mathrm{x_o}$). $\mathrm{x_i}$ represents the initial array content before entering the loops. SW in (g), (h), and (i) indicates Sliding Window.

For each benchmark, we report the maximum number of variables (#vars), cuts (#cuts), and generated cells (#cells) to quantify computational complexity of CAD. We compare Prexo against two baselines: TVLA-Arr [59], an extension of TVLA supporting numeric-array abstract interpretation, and Prexo-NW, a simplified variant of Prexo that replaces our widening with trivial flood-fill returning $\top_{\mathcal{T}_{\langle\pi\rangle}}$. We exclude Exo (Chapters 3–5), which lacks array semantic reasoning entirely. While other studies on abstract interpretation [60,61,67] exist, they provide neither artifacts nor analysis result for comparison. All evaluations were conducted on a MacBook Pro with an Apple M1 Pro chip and 16 GB of unified memory, running macOS Sonoma (version 14.4).

Result Prexo achieves the desired precision with acceptable runtime overhead. For each microbenchmark, the computed abstract state exactly matches the ground truth shown in Figure 6.15, with ten out of thirteen programs analyzed within 100 seconds. The analysis runtime scales roughly linearly with the number of CAD cells generated (Table 6.1). Processing rates remain consistent across problem scales: 223 cells/s for 136 cells (0.61s total), 212 cells/s for 17,190 cells (81.05s total), and 212 cells/s for 264,331 cells (1243.3s total). Since TVLA-Arr [59] provides concrete results only for Array Init 1D and Array Copy, we compare these against ground truth in Figure 6.15. For Array Init 1D, TVLA-Arr requires 1.7 seconds while Prexo completes in 0.61 seconds with the same expressivity. For Array Copy, TVLA-Arr requires 338.1 seconds while Prexo completes in only 2.3 seconds. While different benchmark architectures preclude direct comparison, relative performance is instructive: Array Copy shows a 200× slowdown compared to Array Init 1D under TVLA-Arr, but only 3.7× under Prexo, indicating better scalability.

To the best of our knowledge, no existing analysis can precisely handle microbenchmarks (l) and (m), as syntactic segmentation cannot introduce new partitions (such as $\lfloor N/2 \rfloor$). Prexo constructs such partitions trivially during CAD's lifting phase. The Prexo-NW column shows the ablation variant's runtime. Except for Simple Conditional, Prexo-NW fails to derive precise array contents because its trivial flood-fill aggressively sets the state to $\top_{\mathcal{T}_{\langle\pi\rangle}}$, losing precision. Compared to Prexo-NW, Prexo incurs only a modest average runtime increase of $\sim$2×, while preserving full precision, indicating that CAD-tree construction dominates the computational cost, not widening.

6.7.2 Case Study: RVV Kernel Optimization

Preliminaries We evaluate Prexo on real-world performance-optimization scenarios with RISC-V's Vector Extension (RVV). RVV, like Arm's SVE, adopts a vector-length-agnostic SIMD model where instructions scale to any hardware-provided vector width, eliminating architecture-specific tail code without sacrificing performance. Performance engineers targeting RVV must account for VLEN, an implementation-defined constant that specifies the bit width of each vector register. At runtime, `vsetvli`/`vsetvl` instructions return *vl*—the number of elements fitting into one register group for the chosen element size and LMUL factor. The loop body repeats until *vl* naturally shrinks to finish the workload [68]. Vector-length-agnostic architectures pose unique challenges for compilers. RVV programmers typically

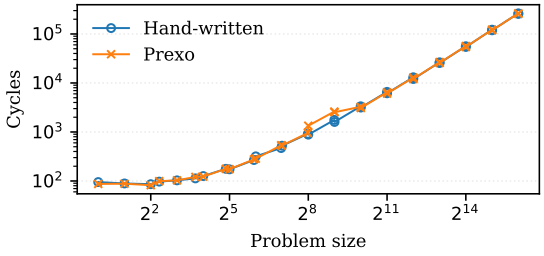
```
for io in seq(0, (n+31)/32):
    for ii in seq(0, min(32, n-32*io)):
        y[32*io+ii] += x[32*io+ii] * a
```

(a) Exo IR code before binding `rvv.vl`.

```
for io in seq(0, (n+31)/32):
    rvv.vl = min(32, n-32*io)
    for ii in seq(0, rvv.vl):
        y[32*io+ii] += x[32*io+ii] * a
```

(b) Exo IR code after binding `rvv.vl`.

Figure 6.16: DAXPY kernel (a) before and (b) after applying `bind`. `rvv.vl` is a globally persistent configuration state. To ensure sound transformation, it is essential to verify that writing a new value to this state preserves functional equivalence. VLMAX = 32 elements for VLEN = 256 bits, SEW = 64 bits, and LMUL = 8.



(a) Prexo vs. expert-written runtime.

Kernel	NCD	#vars	#cuts	#cells	Prexo
DAXPY	1.61s	3	6	93	2.28s
SAXPY	1.62s	3	6	93	2.36s

(b) Compilation time for Prexo-NCD and Prexo.

Figure 6.17: Performance and compilation time for the DAXPY kernel.

hand-write assembly using a "strip-mining" style that tracks remaining elements, decrements the count by *vl* each iteration, and exits when it reaches zero. However, compilers lack static visibility into elements per iteration or loop termination, and `vsetvl` appears as an opaque function call unless explicitly modeled in software. This creates significant challenges for USLs, which must ensure optimized RVV code processes the same `n` elements as the original code—requiring compiler analysis to track *vl* and determine elements per iteration. Exo and other USLs relying on dependency analysis are inadequate for this. To our knowledge, no USLs currently support RVV safely. To illustrate this, consider transforming code from (a) to (b) in Figure 6.16. While the transformation itself is straightforward, the analysis is nontrivial. Since `rvv.vl` is a global parameter, we must handle potential writes elsewhere (e.g., at the end of the `ii` loop) and ensure all uses remain consistent while processing every array element.

Evaluation Setting We optimize DAXPY (standard kernel in BLAS [69], double-precision a-x-plus-y) using the `bind` scheduling operation from Section 6.6.2, which internally calls `Check_Delete`, where we integrated our analysis. We use Exo's scheduling operations (Section 4.2) for other optimizations, since dependency analysis suffices for those transformations. We run the abstract interpretation only for variables requested by the scheduling operation—in this case, just `rvv.vl`, not `y` or `x`. We benchmark performance on Saturn, a parameterized RVV implementation [68], using Verilator cycle counts with `GENV256D128ShuttleConfig` (a dual-issue core with 256-bit VLEN, 128-bit SIMD datapath, and separate floating-point

and integer vector units). We compare the optimized DAXPY kernel against expert-written assembly from Saturn benchmarks [70] and an ablation variant, Prexo-NCD, which bypasses `Check_Delete` analysis to measure the API's overhead. Since Exo lacks safety checking for `bind`, Prexo-NCD effectively represents vanilla Exo applied to RVV.

Result Figure 6.17(a) shows that Prexo-optimized DAXPY code matches the performance of hand-written assembly because Prexo supports the same key optimizations: *vl* binding, staging intermediate operations in RVV registers, and generating RVV intrinsic calls. Figure 6.17(b) compares end-to-end compilation times between Prexo-NCD and full Prexo. Although Prexo compiles approximately 40% slower, its compilation time remains practical since dataflow analysis runs only on `rvv.vl`, keeping the number of variables and cells far below the values reported in Table 6.1.

Chapter 7

Exo-GPU for Tensor Cores

This chapter returns to the first pillar of Exocompilation: externalizing hardware targets. Chapter 4 showed how accelerator instructions can be defined in user code under Exo's sequential semantics. Modern GPUs additionally expose thread hierarchies, distributed memory, asynchronous instructions, and explicit synchronization. Exo-GPU extends the Exo IR and backend with these constructs while retaining the sequential baseline and rewrite-based scheduling of the preceding chapters. Scheduling rewrites establish equivalence under sequential semantics; Exo-GPU's collective and distributed-memory analyses then validate the GPU mapping; and an abstract-machine checker verifies that the final program's sequential and parallel interpretations agree.

7.1 Safety and Control on Modern GPUs

Starting with Tensor Cores introduced in NVIDIA's Volta generation of GPUs [4], the menu of tensor-specialized features in NVIDIA GPUs has expanded to encompass automated bulk movement of multidimensional data, distributed shared memory, and new asynchronous tensor cores supporting large-scale (up to 64×256) matrix accumulation in a single instruction [71]. This poses ever-increasing challenges for authoring correct, top-performing CUDA code. Not only must algorithms be rewritten to migrate work from traditional SIMT-style code to target these specialized instructions, but also, these instructions expose explicit out-of-order instruction execution to the programmer, moving synchronization that was traditionally the responsibility of the hardware (e.g., through scoreboarding) to the programmer.

High-level kernel programming languages, such as Triton [8], provide a popular approach for taming this complexity. Triton's tile-based, automatically parallelized programming model reduces the risk of programmer error by moving responsibility for correct usage of bulk data movement and synchronization instructions into the compiler. However, in practice, groundbreaking algorithms, such as the original FlashAttention 3 [72], are often written directly in CUDA C++ for greater control. On the one hand, CUDA C++ provides direct control over features such as asynchronous data movement and warp specialization required for optimal GPU usage. On the other hand, CUDA C++ exposes programmers to a variety of potential bugs, which we broadly categorize as:

1. *Dataflow Logic Bugs*: Program optimization usually requires changing the storage or

computation order of intermediate values, e.g. caching data in a faster memory or reordering instructions to hide latency. A flawed optimization, such as an indexing bug in copying a tile or exchanging two statements that don't commute safely, can change the functional behavior of the program.

2. *Instruction Logic Bugs*: Accelerator instructions misbehave if used outside vendor specification. For example, `wgmma` tensor-core instructions require data in a certain input format and uniform execution by a full warpgroup (128 aligned threads).

3. *Inter-Thread Synchronization Bugs*: Garden-variety synchronization bugs involve two threads accessing the same memory location (with one access being a write) without ordering enforced by synchronization.

4. *Intra-Thread Synchronization Bugs*: If one thread issues an asynchronous (async) instruction, and that *same thread* later issues another instruction (async or not) that operates on the same memory location, overlapped or reordered instruction execution can undermine correctness.

Our goal is to provide a programming model that gives CUDA-like control *without* accepting the risk of these bugs as a fact of life. CUDA C++ libraries such as CuTe [73] and ThunderKittens [74] offer close-to-the-metal abstractions that minimize the risk of logic bugs, particularly data movement and instruction input-format bugs; however, none of them support large-scale program analysis and synchronization checking of arbitrary input programs.

As a starting point for the Exo-GPU language design, we postulate that programs with *sequential semantics* are far easier to analyze, both by humans and by compilers. We define program behavior using standard sequential semantics, treating language features for parallelism and synchronization–including parallel for-loops across CUDA thread hierarchies and barrier statements–simply as *annotations* on sequential code. This design provides two key benefits: programmers can intuitively reason about their code without hidden control flow or side effects, and more importantly, compilers can verify whether synchronizations are legal; that is, they can verify these annotations do not alter the program behavior relative to its baseline sequential interpretation.

The Exo-GPU program, interpreted sequentially, serves as the semantic baseline—defining *what* the user intends to compute. Parallelism annotations specify *which threads* execute which statement instances, while synchronization constructs specify *how* the user intends to uphold baseline behavior. This raises a question: what degree of safety and control does Exo-GPU provide? We define safety as *sequential-parallel equivalence*–ensuring that the parallel interpretation of a program matches the sequential one. Parallelism is explicitly controlled by the programmer's annotations, rather than being hidden and automated by the compiler. The Exo-GPU compiler's role is limited to *verifying* that the user-provided synchronization ensures equivalence.

Through collective analysis (Section 7.4), we annotate each statement with a *collective tiling*, specifying which thread(s) execute each statement instance. This enables static analysis of the thread-assignment counts, ensuring instructions with thread-convergence requirements (e.g. warp MMA) are used correctly. To ensure sequential-parallel equivalence, we simulate execution on an *abstract machine* (Section 7.5). The Exo-GPU program converts to an

Abstract Machine program where data variables store access histories (reads and mutations) with their thread and async instruction *visibility*, rather than numeric values. Synchronization statements become meaningful only in the abstract machine, conditionally expanding the visibility of all prior recorded accesses.

We build Exo-GPU on the Exo IR and its scheduling framework, which so far target *sequential*, single-threaded code: parallelism is exposed only implicitly, through vector and accelerator instructions and the backend-checked `parallelize_loop()` annotation (Section 4.2.8). On CPUs and the matrix accelerators of Chapter 4, the frontend preserves instruction ordering: even when execution is out-of-order and the reorder buffer overlaps independent operations (Section 2.3), the hardware guarantees results equivalent to those of sequential execution. Establishing equivalence within a sequential model therefore suffices, and Exo can still exploit the available parallelism through latency-hiding techniques such as software pipelining. Modern GPUs revoke this guarantee, exposing explicit out-of-order and asynchronous execution to software.

This chapter extends Exo's frontend, analysis, semantics, and codegen to parallel GPUs. Treating parallel constructs as annotations over a sequential semantic baseline enabled us to reuse Exo's 46 rewrites as-is, which allowed us to transform a simple sequential GEMM program into a complex GEMM kernel for the NVIDIA Hopper architecture, with every step of transformation verified via a *chain of equivalence*: Exo verifies equivalence between the original p_1 and the final p_n while ignoring all the synchronization constructs, and Exo-GPU verifies functional equivalence between the parallel and sequential interpretations of p_n, checking the legality of the parallel and synchronous annotations (Section 7.2.3). Combining Exo's preexisting checked rewrite rules with Exo-GPU's new language features and synchronization checking, we created a GEMM kernel achieving over 80% of theoretical peak for large problem sizes, in some cases outperforming the vendor-supplied cuBLAS library (Section 7.6).

7.2 Examples and System Overview

7.2.1 Exo-GPU Language Overview

Exo-GPU is built on fundamental design goals of safe, performance-transparent, and imperative GPU programming. Unlike other GPU abstractions that hide complexity, we make threads and most CUDA instructions explicit in the IR. We introduce three first-class frontend language features derived from our design principles: (i) parallel loops that explicitly map work to blocks/warps/threads, (ii) a distributed memory model with sharding and explicit memory-space annotations, and (iii) explicit synchronization, including fences, split barriers, and asynchronous operations.

Parallel Loops: Exo-GPU exposes three kinds of loops. (1) `seq`(c_{lo}, c_{hi}) is a sequential loop that iterates i from c_{lo} to $c_{hi} - 1$. (2) `cuda_tasks`(lo, hi) distributes iterations across CUDA clusters; on pre-`sm_90` (H100) GPUs a cluster coincides with a single CTA (thread block). (3) `cuda_threads`$(lo, hi,$ `unit` $= \tau_u)$ assigns iterations to subsets of threads inside the current cluster. The `unit` takes a parameter τ_u that specifies the shape of the participating thread set but not concrete indices—for example, $\tau_u =$ `cuda_warp` denotes contiguous 32-thread groups with thread IDs 0–31, 32–63, 64–95, etc. (inclusive), while $\tau_u =$ `cuda_thread` denotes a single thread (see Figure C.2).

```
for blk in cuda_tasks(0, 37):
  # scope-1 (CTA-scope)
  for w in cuda_threads(0, 4,
                unit=cuda_warp):
    # scope-2 (warp-scope)
    for t in cuda_threads(0, 32,
                unit=cuda_thread):
      # scope-3 (thread-scope)
```

Each loop body, conditional branch, `with`-block, or procedure defines a *program scope*. As part of *collective analysis* (Section 7.4), we annotate each scope with a *collective tiling* $\omega \in \Omega$. This describes a mapping between the control-variable values and the *thread collective* (set of threads) assigned to execute statement instances in the scope (i.e. which threads execute which loop iterations). We say that a scope is a τ_u-scope when τ_u accurately describes the thread sets produced by the mapping. In the inset figure, the outermost parallel loop `cuda_tasks(0,37)` creates `scope-1` (which is a cluster scope, and, assuming `clusterDim=1`, also a CTA scope); nesting `cuda_threads(..., unit=cuda_warp)` creates `scope-2` (warp scope); and nesting `cuda_threads(..., unit=cuda_thread)` creates `scope-3` (single-thread scope). Statement instances in `scope-3` are executed by one thread.

Distributed Memory: Exo-GPU exposes explicit allocation statements for data and barrier variables, annotated by memory type, e.g. `x : f32[8] @ CudaRmem` allocates a data array `x` of type `f32` and size 8 in GPU register memory. Allocations can be *distributed*: a single logical object may be sharded across threads at multiple CUDA levels (e.g., per-thread register shards or per-CTA-in-cluster shared-memory shards). Threads must access only their owned shards except via explicit communication instructions (e.g., warp shuffles) or TMA multicast. This sharding is not annotated explicitly at the allocation but deduced from the usage pattern of the variable; all deductions must be consistent, as enforced by distributed-memory analysis (Section 7.4.1).

```
tmp: f32 @ CudaRmem  # Register
for i in cuda_threads(0, 32,
            unit=cuda_thread):
  tmp = A[i]
for i in cuda_threads(0, 32,
            unit=cuda_thread):
  B[i] = tmp
```

Inset right illustrates the motivation for this design. If the first loop was executed sequentially, `tmp` would hold `A[31]` after the first loop, so the second loop broadcasts that value to every `B[i]`. Languages like CUDA that implicitly duplicate local variables per-thread would interpret the same code as a vector copy (`B[i] = A[i]`). While not necessarily error-prone, this design would be inappropriate for our goal of defining program behavior with sequential semantics. Hence, in Exo-GPU, storing a temporary value per-thread requires a sharded allocation of `tmp`; as written, the program is rejected by the analysis because a register-resident `tmp` cannot be read by different threads without (for example) a warp shuffle. If a user intends to express a vector copy (`B[i] = A[i]`) through register `tmp`, `tmp` requires explicit *distributed* allocation per thread (`tmp:f32[32]`) and the accesses must be per-thread (`tmp[i]`).

Explicit Synchronization: Many GPU instructions require additional explicit synchronization, but overly conservative barriers can cancel the performance benefits of concurrency. Exo-GPU provides two forms of synchronization: a blocking barrier via `Fence`, and split barriers via paired `Arrive`/`Await`. As with other statements, all synchronization statements execute in program

```
# Code at CTA-scope, with blockDim=128
buf: f32[128] @ CudaSmemLinear
for tid in cuda_threads(0, 128, unit=cuda_thread):
  buf[tid] = gmem[tid]
# __syncthreads
Fence(cuda_in_order,cuda_in_order)
for tid in cuda_threads(0, 128, unit=cuda_thread):
  accum: f32 # Each thread sums up buf[...]
  accum = 0
  for i in seq(0, 128):
    accum += buf[i]
```

order.

A `Fence` causes all threads in the current thread collective (warp, CTA, or cluster) to rendezvous. At warp scope it maps to `__syncwarp`; at CTA scope it maps to `__syncthreads` (as shown in the right-hand code inset). A `Fence` takes two synchronization-timeline parameters, τ_s^{pre} and τ_s^{post} (Figure C.7). These timelines specify which memory operations (sync and/or async) before and after the fence must be ordered, and the compiler emits the minimal combination of synchronization and memory-fence instructions required to satisfy them. The `cuda_in_order` timeline specifies that the effects of async instructions are *not* synchronized but only regular synchronous instructions. If we were to, for example, replace the loads into shared memory with `cp.async` operations, the first parameter must be set to `Sm80_cp_async` so that the copy operations preceding the fence are completed before execution proceeds.

Split barriers use a shared `barrier`-typed variable z, declared as z: `barrier @` π_b, where π_b selects the completion mechanism (commit group, mbarrier, or cluster sync). Programmers express a split-barrier statement pair with `Arrive(`τ_s^{pre}`) >>` z and `Await(`z`,` τ_s^{post}`)`. The statement pairing is determined by the shared barrier variable z. A matching pair orders memory operations prior to `Arrive` (filtered by τ_s^{pre}) with those after `Await` (filtered by τ_s^{post}). The right-hand inset illustrates replacing the earlier `Fence` example with an `mbarrier`-based split barrier.

```
for tid in cuda_threads(0, 128,
                        unit=cuda_thread):
  ...
bar: barrier @ CudaMbarrier
Arrive(cuda_in_order) >> bar
# ... insert work here
Await(bar, cuda_in_order)
for tid in cuda_threads(0, 128,
                        unit=cuda_thread):
  ...
```

7.2.2 Exo-GPU Examples with Erroneous Synchronization

To motivate our language design, we build up a series of increasingly subtle, hard-to-identify inter- and intra-thread synchronization bugs; all would be rejected by synchronization checking (Section 7.5). Variables prefixed with `some_*` are allocated outside the code snippet, `some_in_order_op` represents non-async work, and `tma_multicast` and `example_wgmma` are simplified placeholders for actual TMA (tile copy) and wgmma (tensor core) instructions.

The first example shows an inter-thread synchronization bug in a cluster with 2 CTAs:

```
B: f32[2,256,32] @ CudaSmemLinear # Distributed mem: B[0,:,:] on CTA 0; B[1,:,:] on CTA 1
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    some_in_order_op(B[cta, :, :])
    Fence(cuda_in_order, cuda_in_order) # __syncthreads
tma_multicast(B[:,:,:], some_gmem[:,:]) # Copy some_gmem[:,:] to B[0,:,:] and B[1,:,:]
```

The `tma_multicast` instruction is implemented by threads in the 2 CTAs cooperating to fill each others' shared memory; however, this cooperation does not entail implicit synchronization between the two CTAs. Since the `Fence` statement, as placed in CTA-scope, only synchronizes threads within a CTA, it is possible that threads in CTA 0 proceed to the multicast instruction, filling the SMEM of CTA 1 (`B[1,:,:]`), while threads in CTA 1 are still reading from it while executing `some_in_order_op`. Although the TMA instruction is async, this is not relevant to the bug illustrated here, as the TMA appears after the in-order operations upon which it races.

The next example illustrates an intra-thread synchronization bug:

```
# Distributed : D[x,y,:,:] on CTA x, warpgroup y
D : f32[2, 2, 64, 256] @ Sm90_RmemMatrixD(64, 256)
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        example_wgmma(D[cta, wg, :, :], some_A[:, :], some_B[:, :])  # D += matmul(A, B)
        some_in_order_op(D[cta, wg, :, :])
```

This code parallelizes across CTAs and warpgroups, with each warpgroup accessing only its own shard `D[cta, wg, :, :]`. Nevertheless, the code is flawed, as the async `example_wgmma` instruction isn't waited-for before `some_in_order_op` attempts to access its output.

A more subtle bug arises when async instructions interact with inter-thread synchronization:

```
B: f32[2, 256, 32] @ Sm90_SmemSwizzled(128)        # Distributed: B[x,:,:] on CTA x
# Distributed: D[x,y,:,:] on CTA x, warpgroup y
D: f32[2, 2, 64, 256] @ Sm90_RmemMatrixD(64, 256)
# Distributed: cg[x,y] for CTA x, warpgroup y
cg: barrier[2, 2] @ CudaCommitGroup
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        some_in_order_op_1(B[cta, :, :])
        example_wgmma(D[cta, wg, :, :], some_A[:,:], B[cta, :, :])  # D += matmul(A, B)
        Arrive(wgmma_async) >> cg[cta, wg]
cs: barrier @ CudaClusterSync
Arrive(cuda_in_order) >> cs
for cta in cuda_threads(0, 2, unit=cuda_cta_in_cluster):
    for wg in cuda_threads(0, 2, unit=cuda_warpgroup):
        Await(cg[cta, wg], cuda_in_order, 0)
        some_in_order_op_2(D[cta, wg, :, :])
Await(cs, cuda_in_order, 0)
tma_multicast(B[:,:,:], some_gmem[:,:])  # Copy some_gmem[:,:] to B[0,:,:] and B[1,:,:]
```

The statements connected in blue on the left illustrate safe synchronization in a situation similar to our first example: the `Arrive/Await` pair using `cs` (cluster sync) ensures all accesses to `B` by `some_in_order_op_1` finish before any thread in the cluster overwrites `B` using TMA. The statements connected in blue on the right illustrate safe synchronization in a situation similar to our second example: the `Arrive/Await` pair using `cg` (commit group) ensures each warpgroup waits for its `example_wgmma` to finish before accessing `D[cta, wg...]` in `some_in_order_op_2`.

However, the bolded access to `B` by `example_wgmma` is unsafe. Dotted red arrows illustrate synchronization that does *not* happen. Because the `Arrive` on `cs` uses the timeline `cuda_in_order`, the completion of the `Arrive` does not depend on the prior async wgmma instructions to retire. Meanwhile, the `Await` on `cg` (commit group) occurs too late for it to interact transitively with the prior `cs` (cluster sync) `Arrive`. Therefore, while all *non-async* accesses to `B` will retire before any thread proceeds to the TMA multicast, this isn't the case for the async wgmma instructions, whose `B` operand could be overwritten unexpectedly by another CTA multicasting to its input memory. Subtle bugs like this can be difficult to spot in optimized CUDA code.

7.2.3 Optimization Process and Equivalence Checking

Exo-GPU programmers can either directly write optimized object code or apply a sequence of scheduling transformations to simple initial code. Both approaches grant full control over performance-critical details. However, the scheduling approach offers significant practical advantages for performance engineering: optimization strategies become reusable functions across similar kernels, complex transformations decompose into sequences of verified primitives that enable rapid iteration with correctness guarantees, and the compiler automatically handles complex index calculations for tiling strategies—especially valuable for GPU optimization. Below, we describe Exo-GPU's scheduling-based optimization process.

The scheduling flow in Exo-GPU begins with a simple initial procedure p_1. The programmer issues a series of *rewrite operations* R_1 ... R_{N-1} to create transformed procedures $p_2 ... p_N$.

Rewrites:	p_1	$\xrightarrow{R_1}$	p_2	...	$\xrightarrow{R_{N-1}}$	p_N	(Exo-GPU checks)	p_N
Behavior:	$\mathsf{seq}\,[\![p_1]\!]$	$\equiv$	$\mathsf{seq}\,[\![p_2]\!]$	...	$\equiv$	$\mathsf{seq}\,[\![p_N]\!]$	$\equiv$	$\mathsf{par}\,[\![p_N]\!]$

Exo's rewrite engine verifies functional equivalence of $\mathsf{seq}\,[\![p_1]\!]$ ...$\mathsf{seq}\,[\![p_N]\!]$ under sequential semantics (denoted $\mathsf{seq}\,[\![...]\!]$). Exo-GPU adds new scheduling operations for parallel and synchronization constructs, but these non-semantics-altering annotations are ignored under $\mathsf{seq}\,[\![...]\!]$ checks. Then Exo-GPU performs additional verification on the final program p_N. Specifically, synchronization checking ensures $\mathsf{seq}\,[\![p_N]\!] \equiv \mathsf{par}\,[\![p_N]\!]$, where $\mathsf{par}\,[\![...]\!]$ denotes parallel semantics. Transitively, this completes the *chain of equivalence* from $\mathsf{seq}\,[\![p_1]\!] \equiv ... \equiv \mathsf{seq}\,[\![p_N]\!] \equiv \mathsf{par}\,[\![p_N]\!]$, guaranteeing that p_1's behavior is preserved throughout all transformations.

Equivalence checking relies on three categories of reasoning:

Exo Rewrites: Exo's existing rewrites implement dependency analysis-based equivalence checking, which guards against *dataflow logic bugs*. For example, `reorder_stmts`, which reorders two statements $s_1; s_2$, is only permitted when the effects of the statements commute.

Instruction Substitution: Explicit instruction-selection rewriting (`replace`) lets programmers substitute a code block with an accelerator instruction call. Exo-GPU models instructions via their *behavior* (sequential semantics), per-parameter *memory type* (physical memory and layout), expected collective unit τ_u, and async annotations. Exo's unification verifies that the replaced code block's behavior matches the instruction, while Exo-GPU's static analysis ensures appropriate threads execute it (Section 7.4.1); together preventing *instruction logic errors*.

Exo-GPU Synchronization Check: Since parallel loops and synchronization statements (`Fence`, `Arrive`, and `Await`) are ignored during sequential checks, synchronization checking must ensure that their usage is valid and does not alter program behavior or introduce race conditions. We verify this via interpretation on the Abstract Machine (Section 7.5), which protects the program against both *inter- and intra-thread synchronization bugs* for selected concrete problem sizes.

v : Var	$::= x \in \mathbb{X}$	data variable
	$\mid y \in \mathbb{Y}$	control variable
	$\mid z \in \mathbb{B}$	barrier variable
	$\mid r \in \mathbb{W}$	warp variable
e : DExpr	$::= d$	data value
	$\mid x \mid x[w^*]$	reads
	$\mid e_1{+}e_2 \mid e_1{-}e_2$	compound expr
	$\mid e_1{*}e_2 \mid e_1/e_2$	
e_z : ZExpr	$::= z \mid z[w^*]$	reads
e_w : WExpr	$::= r \texttt{ = } (n, n, n)$	warp config
n	$\in \mathbb{Z}$	integer
t	$\in$ Bool	Boolean
d	$\in \mathbb{D}$	data values

c : CExpr	$::= n$	integer
	$\mid y$	reads
	$\mid c_1{+}c_2 \mid c_1{-}c_2 \mid c{*}n$	affine arith
	$\mid c \texttt{ / } n \mid c \texttt{ mod } n$	quasi-affine
b : BExpr	$::= t$	Boolean
	$\mid b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2$	logical ops
	$\mid c_1 == c_2 \mid c_1{<}c_2 \mid c_1{\leq}c_2$	relational ops
w : WinCoord	$::= c$	point access
	$\mid c_1..c_2$	interval access

Figure 7.1: The syntax of variables and expressions of GPU IR. We use $\cdot^*$ to mean 0 or more.

7.3 GPU IR

GPU IR is the frontend language of Exo-GPU, and all the examples in Section 7.2 are written in GPU IR.

7.3.1 Variables and Expressions

Figure 7.1 defines the expression syntax of GPU IR. GPU IR explicitly distinguishes between control, data, barrier, and warp expressions and is a static-control program. We denote the sets of data, control, barrier, and warp variables by $\mathbb{X}$, $\mathbb{Y}$, $\mathbb{B}$, $\mathbb{W}$, respectively. $x \in \mathbb{X}$ is a data variable, $y \in \mathbb{Y}$ is a control variable, $z \in \mathbb{B}$ is a barrier variable, and $r \in \mathbb{W}$ is a warp variable. The truth values $\mathsf{Bool} \triangleq \{\texttt{true}, \texttt{false}\}$ are ranged over by a metavariable t, and n is the metavariable for $\mathbb{Z} \triangleq \{\cdots, -1, 0, 1, 2, \cdots\}$. The metavariable $d \in \mathbb{D}$ ranges over data values, including 8-, 32-, and 64-bit integers as well as single- and double-precision floating-points.

Control expressions CExpr encompass integer, scalar read, and (quasi-) affine arithmetic operations. Boolean expressions BExpr define basic logical and relational operations on control expressions. Data expressions DExpr are similarly defined, but operations need not be affine. Barrier expressions ZExpr are only allowed to read but cannot be composed. Window coordinates w^* (we use $.^*$ to denote a tuple) used in data and barrier read accesses are specified as tuples of point-wise or interval control expressions, expressing windowed access to multidimensional arrays. Scalar read is syntactic sugar for an array access of $w^* = \langle\rangle$ (empty tuple).

7.3.2 Types

All types are fully defined in Appendix C.1. This section provides a general introduction to the different types in GPU IR. Figure C.1 defines data and barrier types. Data types τ_x specify precisions for data variables $\mathbb{X}$, such as `f32` and `i32`. Barrier types τ_z distinguish a non-explicitly guarded barrier from an explicitly guarded barrier (`barrier(z)`); the explicitly guarded barrier case is required only for mbarriers. A *collective type* ($\delta \in \Delta$) defines a

tuple consisting of a domain and a box, each representing d- dimensional natural numbers $((\mathbb{N}^d, \mathbb{N}^d))$. These define a d-level thread hierarchy within a thread cluster and a selection of a number of elements on each level. An argument to a `cuda_threads` loop τ_u is parameterized by a collective type δ, which is defined in Figure C.2. Figures C.3–C.5 classify three categories of memory spaces: π_{d} for data memories (such as `CudaRmem` for CUDA registers), π_{z} for barrier-completion mechanisms (such as `mbarrier` or `commit_group`), and π_{w} for special window aliasing (such as `CUtensorMap`). Similar to τ_u, π_{d} is parameterized by δ.

A qualitative timeline τ_q annotates each *runtime* buffer access. We define distinct τ_q to distinguish between buffer accesses that require different synchronization mechanisms or memory fences to resolve hazards. For example, we distinguish non-async copies from `cp.async` copies (which require `cp.async.wait_all` or similar to wait for), and we distinguish register and SMEM operands of `wgmma` instructions (which require `wgmma.fence` and `fence.proxy.async`, respectively). Section 7.5 will formally define how a runtime buffer access instance $x[i_1, \ldots, i_n]$ tagged with q_1 states that, at the program point and thread, the access to x occurs under q_1. Moreover, the sync timelines SyncTL mentioned in Section 7.2.1, which parameterize synchronization statements, are actually defined as compositions of qualitative timelines (QualTL), as shown in Figure C.7.

7.3.3 Programs

Exo-GPU supports two types of procedures, p : Proc, which models CPU functions (possibly containing CUDA kernel launches) and g : Instr, which models hardware instructions for either CPU or CUDA. Both procedure types share three common components: a statement body that defines sequential behavior, a tuple of control parameters y^*, and a tuple of data parameters x^*. Each data parameter in Proc carries two annotations: τ_x, specifying the required data precision; and π_d, specifying the required memory type for the passed argument.

Instruction parameters (τ_a) also carry both τ_x and π_d annotations, as well as hardware-specific information, namely: out-of-order, convergence, and qualitative timeline information, which all affect only abstract-machine semantics for checking (Section 7.5). The τ_a also contains a distributed collective units tuple (τ_u^*) specifying the CUDA thread hierarchy at which each array dimension is sharded (checked by distributed-memory analysis, Section 7.4.1). The instruction (g) contains annotations for t (a CPU/CUDA flag), a collective unit τ_u specifying the convergent threads required, and π_z and τ_u^*.

7.3.4 Statements

Figure 7.2 defines statements $s \in$ Stmt. A statement can sequence $(s_1; s_2)$, iterate over loops (`seq`, `cuda_tasks`, or `cuda_threads`), and guard (`if` b `then` s). Data effects include indexed writes $x[c^*] = e$, reductions $x[c^*] \mathrel{+}= e$, and window creation $x = x[c_{\mathrm{lo}} : c_{\mathrm{hi}}]$ @ π_{w}. `CudaDeviceFunction` and `CudaWarps` control generation and selection of CUDA threads. Synchronization statements `Fence`, `Arrive`, and `Await` get lowered to CUDA synchronization and proxy fence instructions.

CUDA Kernel Structure: All Exo-GPU code is compiled as CPU code by default. A `CudaDeviceFunction` block specifies a CPU-side launch of a

```
with CudaDeviceFunction(...):
    for cta_m in cuda_tasks(...):
        for cta_n in cuda_tasks(...):
            # Device task starts here
            A_smem: f32[128, 32] @ CudaSmemLinear
            # ...
```

s : Stmt ::=	s_1 `;` s_2	sequencing
	`for` y `in seq(`c_{lo}`,`c_{hi}`) do` s	sequential loop
	`for` y `in cuda_tasks(`c_{lo}`,`c_{hi}`) do` s	cuda tasks
	`for` y `in cuda_threads(`c_{lo}`,`c_{hi}`,unit=`τ_u`) do` s	cuda threads
	`with CudaDeviceFunction(`n, e^*_{w}`) do s`	CUDA device function block
	`with CudaWarps(`n, n, r`) do s`	warp specialization block
	`if` b `then` s	guard
	$p(c^*, e^*)$	non-instruction subprocedure call
	$g(c^*, e^*)$ `>>` e_{z}	instruction subprocedure call
	x`[`c^*`] =` e	write
	x`[`c^*`] +=` e	reduce
	$x = x[w^*] \,@\, \pi_{\text{w}}$	window statement
	$x : \tau_x[c^*] \,@\, \pi_{\text{d}}$	data alloc
	$z : \tau_z[c^*] \,@\, \pi_{\text{z}}$	barrier alloc
	`free` x	data free
	`free` z	barrier free
	`Fence(`τ_s, τ_s`)`	non-split barriers
	`Arrive(`τ_s, n`)` `>>` e^*_{z}	arrive
	`Await(`e_{z}, τ_s, n`)`	await
τ_a : Arg ::=	τ_x	data type (precision)
	π_{d}	data memory type
	`out_of_order` $\in$ Bool	out-of-order execution flag
	`convergent` $\in$ Bool	implicit thread sync flag
	τ^*_u	distributed collective units
	`initial_qual_tl` $\in$ QualTL	initial qualitative timeline
	`ext_qual_tl` $\in \mathcal{P}$(QualTL)	extended qualitative timeline set
	`atomic_qual_tl` $\in \mathcal{P}$(QualTL)	atomic qualitative timeline set

p : Proc ::= `proc` $y^*, (x : \tau_x$ `@` $\pi_d)^*$ `do` s

g : Instr ::= `proc` $y^*, (x : \tau_a)^*, t, \tau_u, \pi_{\text{z}}, \tau^*_u$ `do` s

Figure 7.2: The syntax of GPU IR statements.

CUDA kernel. Its body must contain only a single statement: a nest of one or more `cuda_tasks` loops; `cuda_tasks` loops must not appear elsewhere. Each iteration of the innermost `cuda_tasks` loop comprises a device task and is assigned to one CUDA cluster for execution. The *device scope* of a statement is CPU if outside of a `CudaDeviceFunction` block and GPU if inside. At CPU scope, all loops must have loop mode `seq`. The arguments of `CudaDeviceFunction`(n, e^*_w) respectively define the `clusterDim` and, indirectly, the `blockDim` of the CUDA clusters launched. Each warp expression $e_w : r = (n, n, n)$ defines a group of warps named r, with the first n parameter being the number of warps. If nonzero, the latter two n parameters specify that the warps will adjust the register count down or up, respectively, using a `setmaxnreg.dec` or `setmaxnreg.inc` instruction [71]. We infer that `blockDim` is 32 times the total number of warps named.

Thread Control: Having described `cuda_tasks` loops (inter-cluster parallelism), we now turn to `cuda_threads` loops and `with CudaWarps` (intra-cluster parallelism). A `cuda_threads` loop takes the executing threads in-flight and subdivides them, assigning one subdivision to execute each loop iteration (possibly with left-over, inactivated threads). The `unit` parameter adjusts the number of threads per iteration. The `CudaWarps`(n, n, r) block restricts its body

statement to execute only with threads in the warp variable named by r (which is defined in the `CudaDeviceFunction`); the numeric arguments further restrict the warps selected.

Proc & Instruction Calls: Calls to p : Proc and g : Instr are syntactically similar, distinguished only by the callable's type. Exo-GPU's static analysis enforces that calls to p : Proc only appear at CPU scope, while calls to g : Instr appear at CPU or CUDA scope as appropriate for g. For CUDA instructions, calls to g must only appear at τ_u-scope, where τ_u is the collective unit specified for g. Additionally, each data parameter must have the correct precision τ_x and memory type π_d, and the trailing barrier (if required) must have the correct π_z.

```
my_warp_config = [
  CudaWarpConfig("producer", 1, 40, 0),
  CudaWarpConfig("unused",   3, 40, 0),
  CudaWarpConfig("consumer", 8, 0, 232)]
with CudaDeviceFunction(clusterDim=2,
    warp_config=my_warp_config):
```

Figure 7.3: Launching clusters of 2 CTAs each, with 1 "producer", 3 "unused", and 8 "consumer" warps per CTA (total `blockDim=384`). The consumer warps have 232 registers per thread, and the others only 40.

Synchronization Statements: The user specifies synchronization between threads and/or between async accelerator instructions by inserting synchronization statements. They take *synchronization timeline* (SyncTL) parameters, defined as compositions of qualitative timelines and a transitivity (trnstv?) flag (Figure C.7). The transitivity flag controls interaction between synchronization statements, detailed in Section 7.5.

Each τ_s : SyncTL contains two QualTL sets: the full timeline set (indicated by "full" in Figure C.7), and a temporal timeline set (indicated by "full" or "temp."). A `Fence(`τ_s^{pre}`,`τ_s^{post}`)`, or paired `Arrive(`τ_s^{pre}`,_)` and `Await(_,` τ_s^{post}`,_)` statements, resolves hazards between two accesses to the same array element when: (1) the first access precedes synchronization and uses a QualTL from τ_s^{pre}'s full timeline set, and (2) the second access follows synchronization and uses a QualTL from τ_s^{post}'s full timeline set (for reads) or temporal timeline set (for writes). In hardware terms, the distinction between the full and temporal timeline sets is due to memory fences. If the prior value of a buffer element is *overwritten*, without being read, we require only that the overwrite is temporally ordered after prior accesses to that buffer element, and memory fences may be elided.

Alloc & Free: An allocation statement begins the lifetime of a new data or barrier variable. For data allocations, the data memory π_d determines the physical backing memory (registers (RMEM), shared memory (SMEM), or global memory (GMEM)) and the mapping between multidimensional tensor coordinates and 1D memory offsets (Figure C.3). It also carries an implicit collective type δ, specifying which level of the thread hierarchy at which the physical memory type is allocated (e.g. CTA for shared memory allocations); this is consumed by distributed-memory analysis (Section 7.4.1). For barrier allocations, the BarrierComp parameter π_z controls the completion mechanism used for CUDA code translated from `Arrive` and `Await` statements (Figure C.4). The explicitly guarded barrier type is relevant only for certain configurations of `mbarrier` usage. A free statement ends the lifetime of a data or barrier variable. Currently, we forbid the user from inserting free statements manually; these are added automatically in a separate compiler pass.

Window Statement: A window statement $x_{\texttt{win}}$ `=` $x_{\texttt{data}}$`[`w^*`]` `@` π_w defines an alias to x_{data}. A window into x_{win} can be passed as an instruction argument requiring special window type π_w. Currently we use this feature only to construct `CUtensorMap` objects [71] for TMA instructions; we parameterize the `CUtensorMap` type with its swizzle mode and box shape.

7.4 Collective Analysis

Collective analysis is a static, forward dataflow analysis over a GPU IR procedure. Its results are consumed by (1) a memory analysis, (2) code generation, and (3) an abstract-machine interpreter. The analysis annotates every GPU lexical scope with a collective tiling $\omega \in \Omega$. By default, statements inherit their parents' tiling unchanged; a new tiling is derived only at: device entry (`CudaDeviceFunction`), thread loops (`cuda_threads`), and warp-specialization blocks (`CudaWarps`). The purpose of this analysis is to statically reason about the mapping between work and the threads *within* clusters, so here we ignore `cuda_tasks` loops, which distribute work across clusters. We uniquely identify a thread within a cluster by its *natural thread index*; a set of such indices comprises a *thread collective* $\mu \in \mathbb{T}$, where $\mathbb{T} \triangleq \mathcal{P}(\mathbb{N})$.

Definition 7.1 (Natural thread index)**.** The *natural thread index* for a CUDA thread is `cluster_ctarank * blockDim.x + threadIdx.x`. (Exo-GPU only parallelizes on the x dimension).

Definition 7.2 (Collective tiling)**.** A *collective tiling* $\omega \in \Omega$ for an M-dimensional domain is an M-tuple of *dimension descriptors*

$$\omega = \langle \mathcal{D}_0, \ldots, \mathcal{D}_{M-1} \rangle \qquad \text{with} \qquad \mathcal{D}_m = (D_m,\, \mathcal{O}_m),$$

where $D_m \in \mathbb{N}$ is the extent of dimension m and $\mathcal{O}_m \in \mathcal{O}^*$ is an ordered list of (possibly empty) *dimension operators* $\mathcal{O} : \mathbb{Y} \times \mathbb{N}^3$. We write the thread pitch of dimension m as $P_m \triangleq \prod_{i=m+1}^{M-1} D_i$ where $P_{M-1} = 1$.

The dimensions of a collective tiling ω capture how a cluster's threads are partitioned into hierarchical collectives (cluster, block, warp). We support *reshape* to introduce ad-hoc additional hierarchy (e.g. pairs of warps). From there, each dimension's dimension operators summarize the control variables (identified by $y \in \mathbb{Y}$) that iterate on that dimension. The thread pitch of the dimension describes the distance, in units of natural thread indices, between adjacent elements (e.g. a CTA dimension would have thread pitch `blockDim`).

During forward dataflow analysis, new collective tilings are derived for the three special statements mentioned above, while all other statements inherit their parents' tiling. It produces a child collective tiling through the following transformation:

$$\mathsf{derive}_\omega : \ (\omega^{\text{env}}, \delta^{\text{stmt}}, y, \mathsf{lo}, \mathsf{hi}, \mathsf{tileCount}) \ \rightarrow \ \omega'$$

where ω^{env} is the parent scope's collective tiling and δ^{stmt} is the collective type from the statement. The (optional) iterator y and bounds $[\mathsf{lo}, \mathsf{hi})$ describe a regular slice of linear thread space; $\mathsf{tileCount} \in \mathbb{N}$ is the number of tiles for that slice.

The `cuda_threads` loop and `CudaWarps` block use derive_ω to create a new collective tiling ω based on the one annotating the parent scope. The `CudaDeviceFunction` block derives ω based on launch parameters: ω is 1-D if `clusterDim`$= 1$ (domain D_0=`blockDim`), otherwise 2-D with (D_0, D_1)=(`clusterDim`, `blockDim`); all operator lists start empty.

The implementation of derive_ω proceeds in two steps:

1. *Shape alignment.* Make ω^{env} and δ^{stmt} compatible by reshaping via hierarchical splits so that their thread pitches P_m align.

2. *Single-dimension refinement.* After alignment, at most one dimension m is newly tiled; the derivation appends exactly one operator to $\mathcal{O}_m$.

Once a tiling ω is derived, it is compiled into a *thread mapping*: $f_\omega : \Sigma \to \mathbb{T}$, where $\Sigma \triangleq (\mathbb{Y} \to \mathbb{Z})$ which maps a control environment $\sigma \in \Sigma$ to a runtime thread collective $\mu \in \mathbb{T}$. Equivalently, let $\mathcal{F} \triangleq \{ f \mid f : \Sigma \to \mathbb{T} \}$ and $f_\omega \in \mathcal{F}$. During conversion, each Abstract Machine statement $s^\#$ is annotated with its thread mapping $f_{s^\#} \in \mathcal{F}$. In Section 7.5.3 we write $f_{s^\#}$ for the thread mapping associated with statement $s^\#$.

7.4.1 Distributed-Memory Analysis

Distributed-memory analysis is a static, program-wide consistency check that ensures every use of CUDA-scoped data and barrier variables is compatible with their memory definitions. Each memory type π_d specifies a collective type δ_{π_d}. The analysis validates that allocation and use agree on which threads "own" which memory slices, indexing respects that ownership, and all uses induce the same mapping from logical indices to CUDA thread identifiers. Analysis runs in four steps:

1. **Desugaring instruction call.** To expose the implicit collective requirement encoded by each hardware instruction to the collective analysis in **step 2**, we rewrite instruction calls $g(\cdots)$ into explicit loops. Each argument to the instruction call is wrapped around with `cuda_threads`, with their collective units specified by $g.\tau_u^*$ in the instruction specification.

2. **Collective analysis.** Run collective analysis and annotate each scope with $\omega \in \Omega$.

3. **Triple collection.** For each data or barrier variable, record the *access triple* of all of use sites. We use $\mathcal{T}_v$ to denote a set of those triples for a variable v; these sets for all the data and barrier variables are the sole inputs to **step 4**. Each triple (n^*, ω, c^*) denotes: the subdivided dimensions n^*, the use-site tiling ω, and observed index coordinates c^*. For each use site:

   - **Data.** Compute n^* from the memory's collective type δ_{π_d} and the use-site tiling ω.
   - **Barriers.** Treat all dimensions as subdivided (every axis participates in distribution).

4. **Triple analysis.** For each CUDA-scoped data variable v, we require: (i) consistency of the allocation w.r.t. the memory's δ_{π_d}, (ii) consistency of data or barrier accesses across sites (by checking all triples in $\mathcal{T}_v$), and (iii) all triples in $\mathcal{T}_v$ must derive one and only one thread-pitch tuple that summarizes how incrementing logical indices advances natural thread IDs.

As a result, distributed-memory analysis rejects programs that (i) mismatch allocation and accesses with their memory declaration, (ii) inconsistently index the same data or barrier across sites, or (iii) rely on missing/extra CUDA-thread iterators. Passing guarantees that the program's memory behavior is coherent with its collective structure and declared memory types.

7.4.2 Code Generation to CUDA Code

Lowering `cuda_threads` Loops: The correspondence between parallel loops and threads has thus far been expressed in terms of thread mappings of type $\Sigma \to \mathbb{T}$, which infer a set of active threads from the control environment σ. However, the generated CUDA C++ implements the inverse of this mapping, inferring the value of a control variable `y` from the thread index instead. Each Exo-GPU loop of the form `for y in cuda_threads(lo, hi, unit=`τ_u`) do s` lowers to CUDA C++ as shown in the inset right. Here, `f` is a function of CUDA's built-in `blockIdx` and `threadIdx` variables, and `g` is a Boolean condition on `y` that disables threads not assigned to execute any iteration of the body. Where possible, `f` is 0 and `g` is 1, to facilitate uniform branching.

```
if (int y = f(blockIdx, threadIdx); g(y)) {
    lower(s);
}
```

Warp Specialization: When a device function uses multiple warp variables $r_1, ..., r_W \in \mathbb{W}$, we generate W-many CUDA C++ code paths from a single Exo-GPU device function, each specialized for (and executed by) the threads assigned to one warp variable. Each path uses PTX's `setmaxnreg` instruction to vary per-thread register counts and omits all code paths guarded by a `with CudaWarps` block holding code not intended for the current warp. This allows the PTX assembler to statically verify that register-heavy instructions (e.g. `wgmma`) don't appear on low-register code paths.

Persistent Kernels and `cuda_tasks` Loops: The generated CUDA C++ device functions implement a persistent kernel design that launches exactly the number of thread-block clusters needed to saturate the device. Following the approach of CUTLASS grouped kernel schedulers [75], we implement a C++ task-generator object that generates a series of device-task coordinates for thread-block clusters to execute. The current implementation orders tasks lexicographically and assigns them round-robin to thread-block clusters for execution. While we leave it as future work to provide a mechanism for programmers to customize this ordering, such customization would be unconstrained by the frontend's quasi-affine indexing restrictions since `cuda_tasks` loops impose no semantic ordering, thereby enabling techniques like Morton swizzling.

Shared Memory Allocations: Device functions generated by Exo-GPU use only dynamic shared memory for SMEM allocation. The Exo-GPU compiler deduces the required SMEM allocation size and statically assigns offsets into this allocation for each SMEM-backed variable. Variables with nonoverlapping lifetimes may share the same memory locations; to ensure safe aliasing, we prohibit any thread in the cluster from passing an SMEM free statement until all reads and writes to the freed variable have retired, as enforced by the Abstract Machine (Section 7.5). This is necessary because freed SMEM may be immediately reused by any thread in the cluster. We note that the compiler's alias analysis may be suboptimal, and the cluster-wide safety requirement may be overly conservative (e.g., for CTA-local variables). Providing programmers explicit control over SMEM lifetime and aliasing is left as future work.

$\tau_v : \text{VL} ::=$	`fully_ordered` | `temporally_ordered` | `unordered` | `atomic_only` | `invisible`	visibility levels
$e^{\#} : \text{Expr}^{\#} ::=$	$e_d^{\#} \mid e_z^{\#}$ where $e_d^{\#} \triangleq x[w^*]$ $e_z^{\#} \triangleq z[w^*]$	barrier and data accesses
$s^{\#} : \text{Stmt}^{\#} ::=$	$s_1^{\#}; s_2^{\#}$	sequencing
	`for` y `in loop(`$c_{\text{lo}}, c_{\text{hi}}$`) do` $s^{\#}$	loop
	`if` b `then` $s^{\#}$	guard
	`IncTaskID`	increment the task ID
	`Alloc(`$e^{\#}$`)`	updates SyncEnv for Alloc
	`RecordRead(`$\tau_v, t, e^{\#}, \tau_q, \tau_q^*, e_z^{\#}$`)`	updates SyncEnv with Read
	`RecordMutate(`$\tau_v, t, e^{\#}, \tau_q, \tau_q^*, e_z^{\#}$`)`	updates SyncEnv with Mutate
	`ClearReads(`$e^{\#}$`)`	remove state added by `RecordRead`
	`ClearMutates(`$e^{\#}$`)`	remove state added by `RecordMutate`
	`Fence(`$t, \tau_q^*, \tau_q^*, \tau_q^*$`)`	non-split barriers
	`Arrive(`$t, \tau_q^*, e_z^{\#*}$`)`	arrive
	`Await(`$e_z^{\#}, \tau_q^*, \tau_q^*, n$`)`	await
	`CheckReads(`$\tau_v, t, e^{\#}, \tau_q^*$`)`	checks visibility of Reads in SyncEnv
	`CheckMutates(`$\tau_v, t, e^{\#}, \tau_q^*$`)`	checks visibility of Mutates in SyncEnv
	`CheckBarriers(`$e_z^{\#}$`)`	checks the consistency of arrive and await
$p^{\#} : \text{Proc}^{\#} ::=$	`proc` y^* `do` $s^{\#}$	arguments are control-only

Figure 7.4: Syntax of Abstract Machine IR (AMIR). Variables, types, and expressions (c, b, w) use the same definition as GPU IR. Visibility levels are denoted by the *type* $\tau_v \in \text{VL}$ and range over the five constructors listed above.

7.5 Abstract Machine

To guarantee sequential-parallel equivalence, Exo-GPU performs synchronization checking via Abstract Machine interpretation. This section introduces the AMIR grammar (Section 7.5.1), the translation from GPU IR (Section 7.5.2), and the AMIR execution semantics (Section 7.5.3). We translate GPU IR into AMIR solely for validation: the frontend language remains simple, while synchronization reasoning is factored into explicit AMIR operations that are orthogonal to value computation.

7.5.1 Abstract-Machine Grammar

Figure 7.4 defines AMIR's grammar. It follows GPU IR for control c, Booleans b, window coordinates w, and all variable/type definitions (Figures 7.1, 7.2). AMIR defines only windowed access expressions, written $e_d^{\#} \triangleq x[w^*]$ (data access) and $e_z^{\#} \triangleq z[w^*]$ (barrier access). AMIR introduces visibility levels $\tau_v \in \text{VL}$, which are totally ordered: `invisible` $\prec$ `atomic_only` $\prec$ `unordered` $\prec$ `temporally_ordered` $\prec$ `fully_ordered`.

AMIR has no hardware-instruction procedures (g in GPU IR) and no subprocedure calls. Procedures $p^{\#}$ take control-only arguments. Statements (Figure 7.4) fall into three classes: (i) structural-sequencing, loop, and guard; (ii) recording statements that update the synchronization environment SyncEnv–`Alloc`, `RecordRead`, `RecordMutate`, `Fence`, `Arrive`, and `Await`; and (iii) checking statements that query SyncEnv and fail if visibility obligations are unmet and are otherwise no-ops: `CheckBarrier`, `CheckReads`, and `CheckMutates`.

7.5.2 Conversion from GPU IR to Abstract Machine IR

GPU IR	**Abstract Machine IR**
$s_1; s_2$	$T_s(s_1);\ T_s(s_2)$
`for` y `in seq(`$c_{\text{lo}}, c_{\text{hi}}$`) do` s	`for` y `in loop(`$c_{\text{lo}}, c_{\text{hi}}$`) do` $T_s(s)$
`for` y `in cuda_tasks(`$c_{\text{lo}}, c_{\text{hi}}$`) do` s	`for` y `in loop(`$c_{\text{lo}}, c_{\text{hi}}$`) do (IncTaskID;` $T_s(s)$`)`
`for` y `in cuda_threads(`$c_{\text{lo}}, c_{\text{hi}}, \tau_u$`) do` s	`for` y `in loop(`$c_{\text{lo}}, c_{\text{hi}}$`) do` $T_s(s)$
$x[c^*]\ \diamond\ e$ *where* $\diamond \in \{=, +=\}$	$\underset{;e_1^\# \in T_{\text{ex}}(e)}{\cdot}$ (`CheckMutates(`$\tau_v^{\text{pre}}, \texttt{false}, e_1^\#, \{T_{\text{qin}}(e_1^\#)\}$`);` `RecordRead(`$\tau_v^{\text{post}}, \texttt{false}, e_1^\#, T_{\text{qin}}(e_1^\#), \emptyset, \emptyset$`)`) $\underset{;e_2^\# \in T_{\text{ex}}(x[c^*])}{\cdot}$ (`CheckReads(`$\tau_v^{\text{pre}}, \texttt{false}, e_2^\#, \{T_{\text{qin}}(e_2^\#)\}$`);` `CheckMutates(`$\tau_v^{\text{pre}}, \texttt{false}, e_2^\#, \{T_{\text{qin}}(e_2^\#)\}$`);` `ClearReads(`$e_2^\#$`);` `ClearMutates(`$e_2^\#$`);` `RecordMutate(`$\tau_v^{\text{post}}, \texttt{false}, e_2^\#, T_{\text{qin}}(e_2^\#), \emptyset, \emptyset$`);`), where $\tau_v^{\text{post}} =$ `fully-ordered` and $\tau_v^{\text{pre}} =$ `fully-ordered` if $+=$, else `temporally-ordered`
`if` b `then` s	`if` b `then` $T_s(s)$
`with CudaDeviceFunction(`n, e_{w}^*`) do` s	`Fence(`true, Qcs $\cup$ {`cpu_in_order_qual`}, Qcs, Tcs`);` $T_s(s)$`;` `Fence(`true, Qcs, Qcs, Tcs`);`, where Qcs $\triangleq \text{T}_{\text{qfull}}$(`cuda_stream_sync`), Tcs $\triangleq \text{T}_{\text{qtmp}}$(`cuda_stream_sync`)
`with CudaWarps(`n, n, r`) do` s	`if` true `then` $T_s(s)$
$g(c^*, e^*) \gg e_{\text{z}}$	$\underset{;e^\# \in T_{\text{ex}}(e_z)}{\cdot}$ `RecordRead(fully-ordered, false,` $e^\#, \emptyset, \emptyset, T_e(e_z)$`)` $\underset{;i=0}{\overset{;\text{len}(e^*)}{\cdot}} T_{\text{arg}}(g.p_i,\ e_i,\ e_{\text{z}})$
$x : \tau_x[c^*]\ @\ \pi_{\text{d}}$	`Alloc(`$x[c^*]$`)`
$z : \tau_z[c^*]\ @\ \pi_{\text{z}}$	`Alloc(`$z[c^*]$`)`
`free` x	$\underset{;e^\# \in T_{\text{smem}}(x[:^*])}{\cdot}$ (`CheckReads(`$\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\}$`);` `CheckMutates(`$\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\}$`);`)
`free` z	`CheckBarriers(` $T_e(z[:^*])$ `);` $\underset{;e^\# \in T_{\text{smem}}(z[:^*])}{\cdot}$ (`CheckReads(`$\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\}$`);` `CheckMutates(`$\tau_v^{\text{free}}, \texttt{false}, e^\#, \{T_{\text{qin}}(e^\#)\}$`);`)
`Fence(`$\tau_s^{\text{pre}}, \tau_s^{\text{post}}$`)`	`Fence(`$\text{T}_{\text{tr}}(\tau_s^{\text{pre}}),\ \text{T}_{\text{qfull}}(\tau_s^{\text{pre}}),\ \text{T}_{\text{qfull}}(\tau_s^{\text{post}}),\ \text{T}_{\text{qtmp}}(\tau_s^{\text{post}})$`)`
`Arrive(`τ_s, n`)` $\gg e_{\text{z}}^*$	$\underset{;e^\# \in T_{\text{ex}}(e_z^*)}{\cdot}$ `RecordRead(fully-ordered, false,` $e^\#, \text{T}_{\text{qarr}}(\tau_s), \emptyset, \emptyset$`)` `Arrive(`$\text{T}_{\text{tr}}(\tau_s),\ \text{T}_{\text{qfull}}(\tau_s),\ T_e(e_{\text{z}}^*)$`);`
`Await(`e_{z}, τ_s, n`)`	$\underset{;e^\# \in T_{\text{ex}}(e_z)}{\cdot}$ `RecordRead(fully-ordered, false,` $e^\#, T_{\text{qin}}(e^\#), \emptyset, \emptyset$`)` `Await(`$\text{T}_{\text{e}}(e_{\text{z}}),\ \text{T}_{\text{qfull}}(\tau_s),\ \text{T}_{\text{qtmp}}(\tau_s),\ n$`);`

Figure 7.5: $T_s : \mathsf{Stmt} \to \mathsf{Stmt}^\#$ (statement conversion). All *conversion* functions use the $T\_\cdot$ prefix (Figures C.9-C.11). $\tau_v^{\text{free}} =$ `temporally-ordered`

We describe how GPU IR expressions (Section 7.5.2), statements (Section 7.5.2), and instruction arguments (Section 7.5.2) are converted. Before conversion, we apply two trivial preprocessing steps to GPU IR: (i) Inline all window statements, and (ii) Inline all subprocedure calls that occur outside any `CudaDeviceFunction` block. Throughout this section (and in figures) we use the following notation. For a finite, ordered variables $R = \langle r_0, \ldots, r_{k-1} \rangle$ and a statement template $S(\cdot)$, $\fatsemi_{i=0}^{k} S(x) \triangleq S(r_0)\ ;\ S(r_1)\ ;\ \cdots\ ;\ S(r_{k-1})$. If $R = \emptyset$, the result is no-op. The intent is purely left-to-right sequential composition at AMIR conversion time. When unambiguous, we write $f(A) = \{\, f(a) \mid a \in A \,\}$ for the pointwise image of a set.

Expression Conversion

Figure C.8 defines $T_e : \mathrm{DExpr} \cup \mathrm{ZExpr} \to \mathcal{P}(\mathsf{Expr}^{\#})$, which extracts the windowed AMIR accesses $e^{\#} \in \{x[w^*],\ z[w^*]\}$: scalars d contribute nothing ($\emptyset$); a bare name x or z yields the degenerate accesses $x[\langle\rangle]$ or $z[\langle\rangle]$; explicit indexings $x[w^*]$ and $z[w^*]$ are preserved; and for arithmetic e_1 op e_2 the sets union, $T_e(e_1) \cup T_e(e_2)$. T_e is set-valued (order and duplicates are irrelevant), leaves the value expression unchanged, and merely supplies the converted $\mathsf{Expr}^{\#}$ used by `RecordRead`/`RecordMutate` and subsequent visibility checks.

Statement Conversion

Figure 7.5 gives a structural translation $T_s : \mathrm{Stmt} \to \mathrm{Stmt}^{\#}$ from GPU IR to AMIR. Helper functions referenced by Figure 7.5 are defined in Figure C.9. Sequencing and conditionals are preserved, while all loops are normalized to the abstract `loop` form. Warp/block scopes are dropped and replaced with a guard whose condition is `true`. Assignments $x[c^*] \diamond e$ ($\diamond \in \{=, +=\}$) are lowered to explicit read/write effects. We expand source and destination with T_e and remove sync-exempt expressions via T_{ex}. *Sync-exempt memories* are those whose accesses can be assumed safe without checking: `CudaGridConstant`, `CudaClusterSync`, and `CudaCommitGroup`. *Shared memories* are the following: `CudaBasicSmem`, `CudaSmemAtomicity16B`, `CudaSmemLinear`, `Sm90_SmemSwizzled(B)`, or `CudaMbarrier`.

Hardware instruction calls ignore control arguments and pair the instruction's parameter definition and the arguments via T_{arg}. The entry and exit of `CudaDeviceFunction` is bracketed with fences with `cuda_stream_sync`, capturing both the CPU-to-GPU fence implied by the kernel launch and the serialization of the CUDA kernel with respect to prior and subsequent CUDA kernels and API calls (note that Exo-GPU currently uses only one CUDA stream). During the conversion, we record the scope of each statement using a global environment $\mathsf{Scope} : \mathrm{Stmt}^{\#} \to \{\mathrm{CPU}, \mathrm{CUDA}\}$. Statements outside the `CudaDeviceFunction` block are CPU-scoped, while those inside are GPU-scoped. The scope mapping is used in Definition 7.4. `Fence`, `Arrive`, and `Await` translate a sync timeline (τ_s) into a qualitative timeline set (τ_q^*). T_{tr} queries whether a sync timeline is transitive; T_{qfull} and T_{qtmp} materialize the full and temporal QualTL sets, respectively. `Arrive` selects its arrival qualitative timeline via T_{qarr} to sync-check the barrier itself, thereby preventing use-after-free bugs.

Instruction-Argument Conversion

Figure C.10 expands the conversion $T_{\mathrm{arg}}(g.p_i, e_i, e_z)$ from a GPU IR instruction-call argument to the AMIR statements. The auxiliary projections and predicates referenced in the figure—

T_{ext}, T_{atom}, T_{init}, T_{ooo}, and T_{cvg}—are defined in Figure C.11. For each non-sync exempt expression $e^{\#} \in T_{\text{ex}}(e_i)$, the conversion emits a sequence of checks and effect-recording operations which depends on two properties of the parameter $g.p_i$: its *access mode* (read-only vs. write-only/read–write) and whether it is *atomic*. The access mode of $g.p_i$ is determined by a simple intraprocedural static analysis over instruction body (behavior) $g.s$: if $g.p_i$ is only read (and never written or reduced), it is classified as read-only; if it is written but never read or reduced, it is write-only; otherwise it is read–write. The atomic case is selected exactly when $T_{\text{atom}}(g.p_i) \neq \emptyset$.

Program Conversion

The GPU IR's program `proc` $y^*, (x : \tau_x)^*$; `do` s is translated into an AMIR program `proc` y^* ; `do` $s^{\#}$ where only control arguments are retained. Non-control arguments $(x : \tau_x)^*$ are ignored during the conversion. Control variables are preserved, since GPU IR and AMIR share the same control-variables set $\mathbb{Y}$. T_s is applied to the program body to convert s to $s^{\#}$.

7.5.3 Abstract-Machine Semantics

Definition 7.3 (Control–Value Environment)**.** Let $\mathbb{Z}$ be the set of control values and $\mathbb{Y}$ the set of control variables. $\sigma \in \Sigma \triangleq \mathbb{Y} \to \mathbb{Z}$.

Definition 7.4 (Global Threads Collective)**.** Let $\mathbb{I}$ be the set of task IDs and $\mathbb{T}$ the set of all (local) thread IDs, which we call thread collectives. The set of global thread IDs is $\mathbb{G} \triangleq \mathbb{I} \times \mathbb{T}$. For each statement $s^{\#}$, thread mapping $f_{s^{\#}} : \Sigma \to \mathbb{T}$ returns the set of local threads that execute $s^{\#}$ under σ ($f_{s^{\#}}$ defined in Section 7.4). The scope mapping $\mathsf{scope} : \text{Stmt}^{\#} \to \{\text{CPU}, \text{GPU}\}$ is initialized during the statement conversion to indicate whether each statement belongs to the CPU or GPU scope. Given σ and $\imath \in \mathbb{I}$, define the corresponding global threads

$$g_{s^{\#}}(\sigma, \imath) \triangleq \begin{cases} \{\, (\imath, t) \mid t \in f_{s^{\#}}(\sigma) \,\} & \text{if } \mathsf{scope}(s) = \text{GPU} \\ \mathbb{G} & \text{otherwise} \end{cases}$$

Definition 7.5 (Synchronization Environment)**.** Let $\mathbb{B}$ be the set of barrier names and $\mathbb{X}$ the set of data variables. Define

$$\mathbb{P} : \mathbb{B} \to \mathbb{Z}^n \to \mathcal{P}(\mathbb{N}) \quad \text{and} \quad \mathbb{S} : \mathbb{G} \to \text{QualTL} \to \text{VL},$$

where $\mathbb{P}(b, c^*)$ is the set of observed arrive counts at barrier access $b[c^*]$, and $\mathbb{S}(g, \tau_q)$ is the visibility level for the thread with global ID $g \in \mathbb{G}$ on a qualitative timeline τ_q (Figure 7.4). Let

$$\mathbb{R} \triangleq \text{QualTL} \times \mathbb{S} \times \mathbb{P}$$

be the set of *visibility records*, written (q, $\int$, p). The synchronization environment is the map

$$\rho : (\mathbb{X} \cup \mathbb{B}) \to \mathbb{Z}^n \to \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathbb{R}) \times \mathbb{N}^2,$$

sending a name and index to a tuple of (read visibility record (r): $\mathbb{R}$, mutate visibility record (m): $\mathbb{R}$, barrier count (b): n^2).

$$\frac{}{\langle n, \sigma\rangle \Downarrow_{\mathbb{Z}} n}\ \text{C-Int} \qquad \frac{\sigma(y) = n}{\langle y, \sigma\rangle \Downarrow_{\mathbb{Z}} n}\ \text{C-Read}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} n_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} n_2 \qquad \mathrm{op} \in \{+, -, *, /\}}{\langle c_1 \,\mathrm{op}\, c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} (n_1 \,\mathrm{op}\, n_2)}\ \text{C-Aff} \qquad \frac{}{\langle t, \sigma\rangle \Downarrow_{\mathsf{Bool}} t}\ \text{B-Const}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow_{\mathsf{Bool}} t_1 \qquad \langle b_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} t_2 \qquad \mathrm{op} \in \{\texttt{and}, \texttt{or}\}}{\langle b_1 \,\mathrm{op}\, b_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} \widehat{\mathrm{op}}(t_1, t_2)}\ \text{B-Log}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} n_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} n_2 \qquad \mathrm{op} \in \{==, <, \leq\}}{\langle c_1 \,\mathrm{op}\, c_2, \sigma\rangle \Downarrow_{\mathsf{Bool}} (n_1 \,\mathrm{op}\, n_2)}\ \text{B-Rel} \qquad \frac{\langle c, \sigma\rangle \Downarrow_{\mathbb{Z}} i}{\langle c, \sigma\rangle \Downarrow_{\mathcal{I}} \{\, i \,\}}\ \text{W-Point}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow_{\mathbb{Z}} i_1 \qquad \langle c_2, \sigma\rangle \Downarrow_{\mathbb{Z}} i_2}{\langle c_1..c_2, \sigma\rangle \Downarrow_{\mathcal{I}} \{\, i \in \mathbb{Z} \mid i_1 \leq i \leq i_2 \,\}}\ \text{W-Range} \qquad \frac{\forall j \in \{1..n\}.\ \langle w_j, \sigma\rangle \Downarrow_{\mathcal{I}} W_j}{\langle w_1, \ldots, w_n, \sigma\rangle \Downarrow_{\mathcal{I}^n} W_1 \times \cdots \times W_n}\ \text{W-Tuple}$$

$$\frac{a \in \mathbb{X} \cup \mathbb{B} \qquad \langle w_1, \ldots, w_n, \sigma\rangle \Downarrow_{\mathcal{I}^n} \mathbf{W}}{\langle a[w_1, \ldots, w_n], \sigma\rangle \Downarrow_{\mathsf{Acc}} \{\, \{a \mapsto \vec{i}\} \mid \vec{i} \in \mathbf{W} \,\}}\ \text{E\#-Acc}$$

Figure 7.6: The big-step operational semantics for expressions.

The semantics follows a standard big-step style where judgments take the form $\langle e^{\#}, \sigma\rangle \Downarrow v$ for expressions and $\langle s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma', \rho'$ for statements, indicating that expression $e^{\#}$ evaluates to value v in state σ, and statement $s^{\#}$ transforms state σ to σ', ρ to ρ', and $\imath$ to $\imath'$, respectively.

Expression Semantics

Figure 7.6 gives big-step judgements $\langle e, \sigma\rangle \Downarrow_{\tau} v$. Integer and Boolean literals evaluate to themselves (C-Int, B-Const); variables read from the store (C-Read). Arithmetic evaluates both operands to integers and then applies $\mathrm{op} \in \{+, -, *, /\}$ (C-Aff). Boolean connectives apply the usual truth functions for `and`/`or` (B-Log), and comparisons $\mathrm{op} \in \{==, <, \leq\}$ compare integer results to yield Booleans (B-Rel). Windows denote index sets $\mathcal{I}$: a point yields $\{i\}$ (W-Point); a range $c_1..c_2$ yields the inclusive set $\{\, i \in \mathbb{Z} \mid i_1 \leq i \leq i_2 \,\}$ (W-Range); and an n-tuple of windows denotes the Cartesian product $W_1 \times \cdots \times W_n$ (W-Tuple). The only nonstandard part is E#-Acc: given $a \in \mathbb{X} \cup \mathbb{B}$ and a window tuple $\mathbf{W}$, it *unpacks* the window into the set of concrete accesses $\{\, \{a \mapsto \vec{i}\} \mid \vec{i} \in \mathbf{W} \,\}$ of type $\mathbb{X} \cup \mathbb{B} \to \mathbb{Z}^*$.

Statement Semantics

Figure C.12 gives big-step operational semantics for a loop, sequencing, an explicit task–ID increment, and a guard. For `for` y in `loop`$(c_{\mathrm{lo}}, c_{\mathrm{hi}})$ do $s^{\#}$, the bounds evaluate in σ to integers m and n; the loop ranges over the *half-open* interval $[m, n)$: when $m < n$ (LoopStep) the body executes once with $y \mapsto m$, producing $(\imath', \sigma', \rho')$, and the loop recurs on $(m{+}1) \ldots n$; when $m \geq n$ (LoopDone) it terminates with no effect. Sequencing (Seq) threads the triple $(\imath, \sigma, \rho)$ from the first statement into the second. The guard (IF-True/IF-False) evaluates

$$\frac{\begin{array}{c}\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e^{\#}} \\ \exists(\_, \int, \_) \in \rho(W_{e^{\#}}).r\,((t = \text{false} \wedge \exists g \in g_{s^{\#}}(\sigma, \imath).\, \forall \tau_q \in \tau_q^* \mid \int(g, \tau_q) \not\succeq \tau_v) \\ \vee (t = \text{true} \wedge \forall g \in g_{s^{\#}}(\sigma, \imath).\, \forall \tau_q \in \tau_q^* \mid \int(g, \tau_q) \not\succeq \tau_v)) \Downarrow_{\mathsf{Bool}} \text{true}\end{array}}{\langle \text{CheckReads}(\tau_v, t, e^{\#}, \tau_q^*), \imath, \sigma, \rho\rangle \Downarrow \text{error}}\ \textsc{CheckReads}$$

$$\frac{\begin{array}{c}\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e^{\#}} \\ \exists(\_, \int, \_) \in \rho(W_{e^{\#}}).m\,((t = \text{false} \wedge \exists g \in g_{s^{\#}}(\sigma, \imath).\, \forall \tau_q \in \tau_q^* \mid \int(g, \tau_q) \not\succeq \tau_v) \\ \vee (t = \text{true} \wedge \forall g \in g_{s^{\#}}(\sigma, \imath).\forall \tau_q \in \tau_q^* \mid \int(g, \tau_q) \not\succeq \tau_v)) \Downarrow_{\mathsf{Bool}} \text{true}\end{array}}{\langle \text{CheckMutates}(\tau_v, t, e^{\#}, \tau_q^*), \imath, \sigma, \rho\rangle \Downarrow \text{error}}\ \textsc{CheckMutates}$$

$$\frac{\langle e_z^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e_z^{\#}} \qquad (\exists(a, w) \in \rho(W_{e_z^{\#}}).b \mid a \neq w) \Downarrow_{\mathsf{Bool}} \text{true}}{\langle \text{CheckBarrier}(e_z^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow \text{error}}\ \textsc{CheckBarrier}$$

Figure 7.7: Big-step rules for access visibility and barrier consistency. Successful checks leave $(\imath, \sigma, \rho)$ unchanged; any violation reduces the configuration to error state.

b in σ and either executes $s^{\#}$ or behaves as a no-op. The only nonstandard rule, InctaskID, increments the task identifier ($\imath' = \imath + 1$) while leaving σ and ρ unchanged.

Figure C.13 shows the rest of statement semantics. Alloc evaluates each extent c_i to an integer n_i (under $\Downarrow_{\mathbb{Z}}$), forms the index set $I = \{(v_1, \ldots, v_k) \mid 0 \leq v_i < n_i\}$, and initializes the per-access visibility record for x by updating ρ so that, for every $\mathbf{n} \in I$, $\rho(x, \mathbf{n}) = (\emptyset, \emptyset, (0, 0))$; the thread id $\imath$ and store σ are unchanged. ClearReads first resolves the accessed locations via $\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e^{\#}}$ and, for each $(x, \mathbf{n}) \in W_{e^{\#}}$, replaces $\rho(x, \mathbf{n}) = (r, m, b) \in \mathbb{R}$ with $(\emptyset, m, b)$. ClearMutates is analogous but replaces it with $(r, \emptyset, b)$. RecordRead and RecordMutate first evaluate the accessor $e^{\#}$ and $e_z^{\#}$ to access sets $W_{e^{\#}}$ and $W_{e_z^{\#}}$ (via $\Downarrow_{\mathsf{Acc}}$). They then update the visibility map ρ at each key $(x, \mathbf{n}) \in W_{e^{\#}}$ using the new record constructor $\mathcal{R}$ defined in Appendix C.2.

`Fence`, `Arrive`, and `Await` update ρ solely via $\text{lift}(\rho, \lambda)$ (defined in Appendix C.2), which applies λ pointwise to both the read and mutate visibility sets. In the following, $\mathcal{W}$ decides when augmentation is permitted, $\mathcal{A}$ raises visibility to `fully_ordered` or `temporally_ordered`. For $\texttt{Fence}(t, \tau_q^{\text{pre}*}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*})$, $\lambda(r) = \mathcal{A}\big(r,\, g^{\text{exec}*},\, \tau_q^{\text{full}*},\, \tau_q^{\text{temp}*}\big)$ iff the witness $\mathcal{W}\big(t,\, g^{\text{exec}*},\, \tau_q^{\text{pre}*},\, r\big)$ holds, otherwise $\lambda(r) = r$; thus visibility is raised by joining $f \vee f_{\text{aug}}$. $\texttt{Arrive}(t, \tau_q^{\text{pre}*}, e_z^{\#*})$ and $\texttt{Await}(e_z^{\#}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*}, n)$ are defined similarly in Appendix C.2, updating the access visibility appropriately.

Checks

Figure 7.7 gives three pure checks that either signal error or leave $(\imath, \sigma, \rho)$ unchanged. CheckReads evaluates $e^{\#}$ to accesses $W_{e^{\#}}$ and raises error when some $(\_, \int, \_) \in \rho(W_{e^{\#}}).r$ fails to supply visibility at least τ_v for all timelines $\tau_q \in \tau_q^*$. CheckMutates is analogous but ranges over mutate visibility record $\rho(W_{e^{\#}}).m$. CheckBarrier evaluates $e_z^{\#}$ and signals error iff some $(a, w) \in \rho(W_{e_z^{\#}}).b$ has $a \neq w$, i.e., the recorded *arrive* and *await* counts disagree.

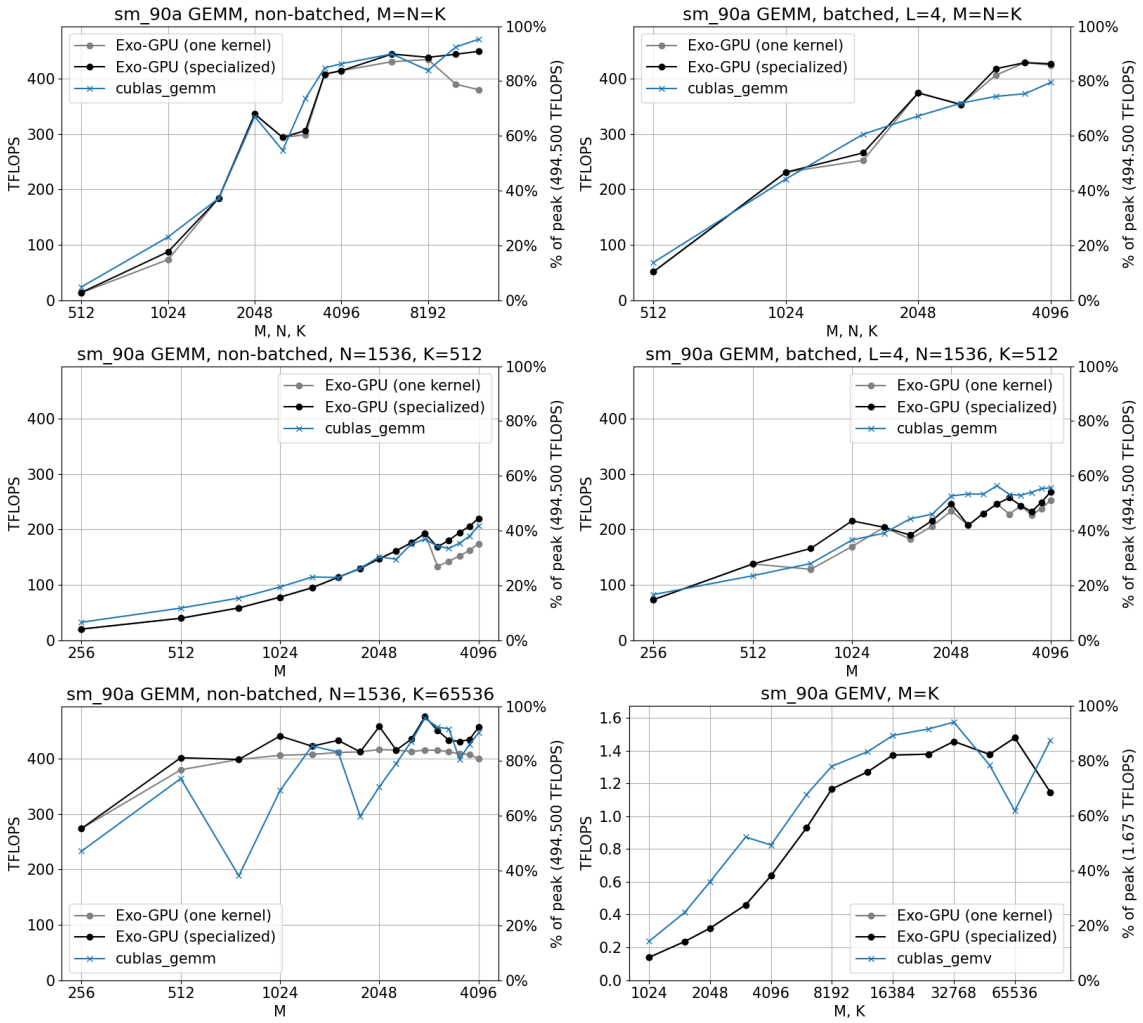


Figure 7.8: Performance by varying problem size on a sm_90a GPU.

Program semantics

Finally, the abstract machine executes the program `proc` y^* ; `do` $s^{\#}$. The synchronization check is exposed to the user, who invokes the abstract-machine interpreter with concrete integer values corresponding to the control variables y^*. The control-value environment $\sigma : \mathbb{Y} \to \mathbb{Z}$ is initialized based on the user's input. The statement $s^{\#}$ is then evaluated according to the semantics defined previously. Note that the task id $\imath$ is initialized to 0, and the synchronization ρ is initially empty.

7.6 Performance Results

We implemented tf32-precision GEMM kernels for the sm_90a architecture, using `wgmma` and TMA instructions. We also implemented a full-precision fp32 GEMV kernel using warp shuffles and scalar math [76]; this did not use the accelerator instructions. We compared the Exo-GPU kernels to cuBLAS's `cublasSgemm`, `cublasSgemmStridedBatched`, and `cublasSgemv`

functions, all with the math mode set to `CUBLAS_TF32_TENSOR_OP_MATH`. Our sm_90a-compatible test hardware contained an NVIDIA H100 80GB SXM5 GPU and an AMD EPYC 9454 48-Core Processor, and we used nvcc `Build cuda_12.3.r12.3/compiler.33492891_0` to compile the CUDA C++ sources generated by Exo-GPU. For GEMM, a compute-bound workload, theoretical peak is 494.5 TFLOPS [77]. For GEMV, a memory-bound workload, we derive the theoretical peak from the memory bandwidth: $3.35\frac{\text{TB}}{s}\frac{2\text{ FLOP}}{4\text{ B}} = 1.675\frac{\text{TFLOP}}{s}$, with one multiply and one add done for each 4 byte tf32 element loaded from the matrix (the bandwidth from loading the vector is amortized).

We tested each problem size and hardware combination over 105 iterations: 5 warm-up iterations and 100 timed iterations, with the pcg3d hash algorithm [78] used to generate "random" input matrices. Each iteration tested, in a randomized order, the cuBLAS kernel and all applicable Exo-GPU kernel variants for that problem size. In particular, we tested non-split-k and split-k variants of the GEMM kernel, with the split-k kernels using `cp.reduce.async.bulk` (TMA) and split factors of 1, 2, 4, 8, and 16. For each (problem size, hardware) combination, we collected 100 time samples per tested kernel and reported the mean of the inter-quartile range (middle 50 samples). For each plot in Figure 7.8, we report the best-performing Exo-GPU kernel for each problem size (plotted as "Exo-GPU (specialized)") and the single best-performing Exo-GPU kernel ("Exo-GPU (one kernel)").

For large, square matrix GEMM problem sizes, we see very similar performance between Exo-GPU and cuBLAS; this is expected, as Exo-GPU is able to express all optimizations required for a peak-performance sm_90a gemm: warp specialization, cluster & TMA multicast support, `wgmma` support, and split-barrier synchronization using commit group, mbarrier, and cluster sync mechanisms. For more unusual GEMM problem sizes, such as small matrices and `K=65536` workloads, we see more variation between Exo-GPU and cuBLAS, possibly due to different problem-size thresholds for observing wave-quantization effects. For GEMV, Exo-GPU and cuBLAS follow a similar performance trend, with cuBLAS having an almost-consistent performance advantage.

Abstract Machine performance. We report the performance of the synchronization checking alone, which is implemented as an interpreter for the abstract machine (Section 7.5). We benchmarked using only one thread of a laptop with an "`11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz`" CPU. In Figure 7.9, we report times for concrete square matrix problem sizes for the GEMV kernel and the sm_90a GEMM kernel described earlier (both the split-k and non-split-k versions). The batch size `L` is 1. We use one untimed warmup run for each problem size, and report the average of five timed runs, as reported by the `timeit` module of Python 3.10.12.

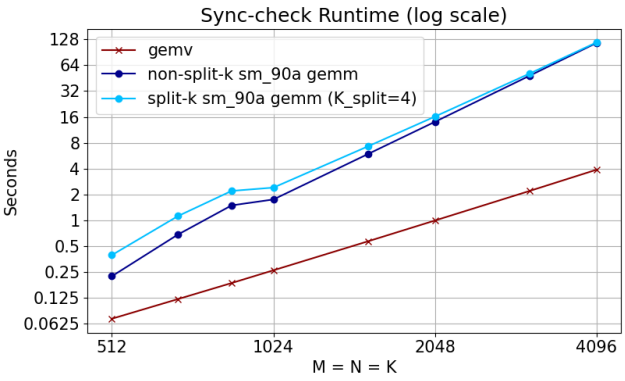


Figure 7.9: Sync-check runtime by problem size.

We observe that runtimes are linear in the number of memory operations interpreted. The number of memory operations is cubic in the problem size for GEMM and quadratic for GEMV. We see this pattern followed cleanly by the measured trend (e.g. 2/16/128 seconds for GEMM for 1024/2048/4096 problem size), indicating no unexpected overhead for large

problem sizes. The bump for small GEMM problem sizes is due to imperfect division by the per-cluster tile size. Even if the expected problem size for a deployed kernel is large, during development of an Exo-GPU kernel, it generally suffices to test only with a small problem size (e.g. 768) for immediate feedback on bugs, with the option of testing the finalized kernel with the true, larger problem size for additional confidence.

Chapter 8

Related Work

This thesis develops one continuous line of language design across Chapters 3–7, unified by the *Exocompilation* philosophy: an imperative, rewrite-based core; user-defined hardware targets and scheduling libraries; value-sensitive analysis; and explicit GPU concurrency. In the discussion that follows, the recurring theme is where compiler automation should end and user control should begin, and what static analyses are needed to keep that control safe as the hardware it targets grows more expressive.

8.1 Metaprogramming

Homogeneous and heterogeneous metaprogramming Scheduling is fundamentally a metaprogramming process where the schedule is a metaprogram that derives optimized implementations from a base object program. Tim Sheard introduced a distinction between homogeneous and heterogeneous metaprogramming systems: homogeneous systems use the same language for both metaprograms and object programs, whereas heterogeneous systems use different languages for the two [79]. Staged metaprogramming has been extensively explored in homogeneous systems for code generation and building DSLs since the 1990s, pioneered by languages such as MetaML, MetaHaskell, MetaOCaml, Lisp, and Racket [80–87] and remains an active research area in modern systems [88–91]. The Terra-Lua system [92,93] introduced a heterogeneous metaprogramming approach, using Lua [94], a dynamic, high-level language, to metaprogram Terra, an embedded, lower-level object language. User-schedulable languages (USLs) are heterogeneous systems since the algorithm and scheduling languages have separate syntax and semantics but are fundamentally different from Terra because they are *transformative*.

Generative and transformative metaprogramming We propose a new taxonomy within metaprogramming languages, distinguishing between *generative* and *transformative* metaprogramming. Generative metaprograms create new object code by constructing and interpolating code fragments, while transformative metaprograms deconstruct and transform a code object into another code object. Lisp-inspired metaprogramming systems and Terra are predominantly generative, creating new code at compile time using constructs like `quote` and `unquote`. On the other hand, jQuery [95], XSLT [96], ELEVATE [46], and its

predecessor strategy languages [55,56,97] focus on transformative metaprogramming. jQuery enables users to navigate and manipulate the HTML DOM using methods like `.parent()` and `.prev()`, while XSLT allows users to transform XML documents via XPath pattern matching. ELEVATE and strategy languages use pattern matching to obtain references to the object code. Racket, with `syntax-case` and `syntax-parse`, supports both generative and transformative metaprogramming by allowing code transformations while preserving lexical scoping and hygiene. Inspired by transformative approaches, Exo supports jQuery-like navigation and a `.find()` operation (Section 5.4). USLs are a subset of transformative systems which preserve functional equivalence.

8.2 User-Schedulable Languages

USLs describe optimization as rewriting programs into new programs that compute the same result but execute much faster. Rather than relying on opaque compiler heuristics, programmers directly control these rewrites by specifying scheduling operations. Halide [43,47] popularized the idea of USLs, and subsequent Halide-inspired languages explored different target applications and control-abstraction boundaries [44,45,48,49,98–102]. For instance, TVM offers accelerator abstraction through `tensorize`, TACO separates tensor definitions from sparse formats, and Taichi abstracts memory layout and management with `fields`. Several systems have extended user-scheduling to support modern GPU features: TVM's TensorIR [103] provides intrinsics for manipulating low-level constructs like split barriers, while Cypress [104] lets programmers describe computations as tasks and map them to different resources. This explicit control over compiler transformations was foreshadowed earlier in many script- or pragma-based compiler tools in HPC [105–109] and in the definition of parametric optimization spaces in SPIRAL [110], all of which have been applied to matrix-matrix multiply and related kernels. The polyhedral loop optimization community has simultaneously explored similar ideas in its own context [111–116].

Rewrite vs. Lowering-based In sharp contrast to Halide, Exo implements scheduling as algebraic rewrites within a small core language. Like Lift and ELEVATE [45,46], but unlike Halide and TVM, each Exo scheduling operator rewrites one program into another rather than supplying an argument to a single monolithic lowering pass. This separation matters: because every primitive is defined and verified independently of the others, the implementation stays substantially simpler and easier to maintain. Where Exo departs from these earlier rewrite-based systems is in the object language they manipulate. Lift and ELEVATE rewrite restricted *functional* programs, for which equivalence across a rewrite is straightforward to establish and, in practice, often left unchecked [45,46]. Exo instead rewrites *imperative* code, as discussed below.

Functional vs. Imperative The Halide lineage of USLs represents object code *functionally*, expressing algorithms as pure functions. This yields elegant abstractions and concise scheduling operators, but it hides performance-critical details such as hardware state and explicit memory instructions. Exo and Allo [58] instead adopt *imperative* object code that makes these details visible and directly manipulable, which lets Exo expose configuration

state (Chapter 4), distributed GPU memory, and explicit synchronization (Chapter 7) as ordinary program constructs. Sliding-window optimization shows both sides of this tradeoff: Halide can apply it only *implicitly*, leaving programmers to rely on the backend to introduce it during code generation, whereas an imperative representation lets them express it explicitly as an ordinary loop transformation. This expressiveness comes at a cost, however, since mutable state complicates analysis and can restrict which transformations are provably sound; confronting that cost is the subject of the effect analysis of Section 3.4 and the value-sensitive analysis of Chapter 6.

Reference in USLs Chapter 5 identified three limitations that existing USLs share: they offer little or no ability to *inspect* the object code (Section 5.3), they rely on inflexible "nominal" referencing schemes, and they provide no support for multiple, time-stable references (Section 5.4). For instance, MLIR's Transform dialect [117] rewrites one MLIR program into another through transformative metaprogramming, and, although it does not guarantee that its transformations are safe, it exposes a USL-like interface organized around references it calls *handles*. In our terminology, handles are *one-time, nominal* references: they are labels attached to the object IR rather than a means of navigating it, and because any transformation invalidates them—consistent with the SSA nature of MLIR—each must be reassigned a fresh name after every step. Exo's original pattern-matching references (e.g., `"for i in _:_"`) are one-time and nominal in the same sense and likewise provide no cursor forwarding, relative navigation, or object-code inspection.

This referencing model is the crux of the problem. Because a scheduling function cannot be parameterized over the references it operates on, it is difficult or impossible to encapsulate user-defined transformations into libraries of *new scheduling operators* (Section 5.5). Schedules must therefore be written as repeated, explicit instantiations of a scheduling pattern rather than as invocations of a single encapsulated generalization of it, which makes them harder to maintain and to adapt to new purposes. Exo closes this gap: unlike prior USLs, which let users write new *schedules*, Exo also lets them write new *scheduling operators* and assemble them into reusable libraries, enabled by its cursor, inspection, and reference mechanisms. This goal is inspired by Guy Steele's "Growing a Language" [51], which argues that a language should not be static but should give its users the means to introduce new abstractions—here, a scheduling language designed to scale to large projects through libraries of user-defined scheduling operators.

Auto-schedulers A complementary line of work automates scheduling rather than exposing it. Auto-schedulers for Halide [17–19] and TVM [20–22] search the scheduling space automatically, and equality saturation [23,24] explores program optimizations by construction. Recognizing that pure automation is not always sufficient, guided optimization [118] and guided equality saturation [119] fold user guidance back into these automatic techniques. Exo shares the premise that user input is essential to scheduling but approaches it from the opposite direction: instead of guiding an automatic search top-down, it builds automation bottom-up from fine-grained, equivalence-preserving primitives (Section 5.2) that users compose into the reusable scheduling operators and libraries of Section 5.5, dropping to lower-level primitives wherever needed.

8.3 Program Analysis and Verification

Our framework for verifying the equivalence and safety of Exo programs rests on two analyses: the effect analysis of Section 3.4, which justifies a rewrite by proving that statements touch disjoint locations; and the value-*sensitive* analysis of Chapter 6, which reasons about the values statements write when disjointness alone is too weak. Both draw on long lines of work in type systems, dependence analysis, and abstract interpretation. Dependently typed arrays, especially as adapted in the formalization of Halide, inform our treatment of memory safety [42,50,120,121]. Dependence analysis, especially on static-control programs, forms a common basis for reasoning about the safety of loop transformations [41,122]; combined with reasoning about affine indexing, this is the basis of polyhedral compilation [123]. Our effect analysis instead builds on the effect types of Gifford and Lucassen [124], though the earliest foundations of dependencies for program parallelization already define conditions on read and write sets closely related to our effects [125]. The value-sensitive analysis of Chapter 6 departs from all of these by tracking array *contents* rather than access sets and so builds instead on abstract interpretation for arrays, which we discuss next.

Exo can nonetheless be seen as a polyhedral compiler, in the sense that it is built on linear integer arithmetic and static-control programs. Its program analysis goes beyond what is normally called "polyhedral analysis" in two respects, both driven by the needs of configuration-state accelerators: it tracks mutable control state, for which we rely on an approximating symbolic dataflow analysis (Section 3.4.3); and it justifies code deletion and insertion (Section 3.4.7), which dependence-only reasoning cannot. The value-sensitive analysis of Chapter 6 generalizes both along two axes. First, it replaces the coarse loop summarization of Section 3.4.3 (which collapses any loop-varying configuration field to an unknown value) with reasoning about the values a loop actually writes, extending control-sensitivity from loop-invariant to loop-varying state. Second, it lifts that reasoning from scalar configuration fields to arbitrary array contents, which is what lets it certify the value-dependent deletions and insertions of Chapter 6, such as redundant-write elimination and value-dependent loop fusion. Tracking mutable control state also forced us to adopt ternary logic at the base of the analysis, so that the dataflow approximations could be propagated soundly. Were configuration state eliminated, Exo would more closely resemble traditional polyhedral compilers, focused purely on reordering statement instances.

Array Reasoning Reasoning about programs that manipulate arrays poses a fundamental challenge in static analysis [59,62,126,127]. Arrays may have unbounded lengths, and loops can execute an unbounded number of iterations, complicating the inference of properties such as the absence of out-of-bounds accesses [128,129], invariants over array contents [62,130], and loop termination [126]. Over the past two decades, a wide range of techniques have been developed to analyze arrays and loops in a sound manner. Abstract interpretation typically offers a general static-analysis framework by interpreting program constructs over abstract domains [131]. Several frameworks summarize sections of arrays within numeric domains [59], while others partition arrays into segments to track segment-specific properties [61]. Also, symbolic abstract domains have been proposed for array reasoning to better capture memory updates [62,130], and noncontiguous array-partitioning techniques have been designed to infer precise invariants even when similar elements are dispersed across an array [132]. Instead of

targeting general pointers or containers, we target a domain-specific language that excludes pointer manipulations and restricts control flows to affine expressions. Languages with these restrictions are still expressive enough for optimizing many HPC kernels like BLAS [69], which includes widely used linear algebra kernels like GEMM, GEMV, and SAXPY/DAXPY. The CAD tree domain and the flood-fill widening in Chapter 6 are designed to maximally exploit this domain-specificity.

The parameterized *decision-tree* abstract domain of Urban & Miné [126] is the closest in spirit to our use of trees: internal nodes test linear constraints, and leaves range over a chosen numerical domain. They target conditional termination, where leaves denote (pieces of) ranking functions, and widening is based on left-unification plus leaf extrapolation. In contrast, Prexo is designed around *array relations*. Prexo's leaves carry value information including symbolic reads of input arrays (e.g., $x[\zeta^*]$), which lets us relate results back to inputs without knowing concrete contents—precisely the kind of value sensitivity needed for array transforms. A second difference is how disjunctions arise: instead of growing trees only from syntactic tests, we build trees by CAD over iterator/offset variables. CAD introduces *semantic* splitters that may not appear in the code (e.g., the $\lfloor N/2 \rfloor$ boundary in our reverse kernels, Section 6.7.1), yielding strictly finer partitions and, in practice, more precise array-content invariants. Finally, whereas Urban & Miné widen by unifying tree shapes, we use the flood-fill widening on space-time arrays: after a join, we propagate leaf values forward along the iteration direction, efficiently carrying array values through the fixpoint. These choices specialize the tree structure and payload base domains to value-sensitive array analysis and explain the precision gains we observe on sliding-window and reverse-access patterns.

Polyhedral Compilers Polyhedral compilers analyze nested loops using polyhedra to represent iteration spaces and dependencies, enabling affine transformations like loop fusion, tiling, and parallelization. They ensure soundness by preserving all dependencies, so that $T(s) \leq T(e)$ for each dependency edge $s \to e$ under transformation T. Some systems generate affine transformations automatically [14–16], while others accept user-provided ones [114]. As noted above, Exo's effect analysis already sits in this tradition. The value-sensitive analysis of Chapter 6 tracks not only *which* memory locations are accessed but also *what* values are written to them. Because classical polyhedral frameworks reason only about dependencies between statement instances, they cannot justify optimizations whose correctness depends on the data values computed (such as the redundant-write elimination and value-dependent loop fusion of Chapter 6) which is addressed by our analysis.

Safety Guarantees Prior work in USLs has established various methods to guarantee functional equivalence between original and optimized code. In functional USLs, Halide lacks sound compiler analysis and relies on scheduling operation-specific compiler passes with interval analysis, while ATL provides foundational verification of scheduling transformations using Rocq [133]. In imperative USLs, original Exo primarily reasons about access sets and symbolic scalar configuration state. Allo [58] uses an external equivalence checker based on symbolic execution [134], though this approach is restricted to handling bounded loops and lacks both termination guarantees for infinite loops and soundness guarantees.

Despite their differences, USLs share a fundamental constraint: each USL must prove

the soundness of its supported transformations, with the scope of possible transformations inherently bounded by the capabilities of the language's analysis engine. Imperative USLs face additional challenges in reasoning about state mutations and side effects. Chapter 6 tackles this challenge via value-sensitive analysis—determining when program statements can be safely modified without affecting program output. Prexo aims to extend imperative USLs' transformation expressiveness by incorporating a precise abstract value information into the safety checks of scheduling operations.

Theorem Provers Our bottom-up approach to building safe scheduling abstractions is inspired by theorem provers. In the 1970s, to address the issue of LCF [135] sometimes combining incorrect proofs and asserting their validity [136,137], Robin Milner developed ML to only allow users to construct valid proofs within the language using a set of trusted axioms [138]. This "LCF approach" has led to modern theorem provers like Coq/Rocq [139]. Exo adopts the same discipline: the equivalence-preserving primitives of Chapter 4 play the role of trusted axioms, and the user-defined scheduling operators and libraries of Chapter 5 compose them into complex, still-trustworthy scheduling programs—the mechanism by which this thesis reintroduces automation without enlarging the trusted core.

8.4 Code Generation and Library Targets

Instruction Selection Exo's instruction/procedure-mapping mechanism is related to the classic problem of instruction selection [140]. Traditional instruction selection applies local pattern-matching rules to replace small IR fragments with equivalent instructions, but this struggles to exploit accelerator instructions, which correspond to large, complex program fragments. Recent work applies more powerful search techniques to target complex SIMD instructions using program synthesis [141] and equality saturation [142]. Exo's `replace` operator and its unification-modulo-affine-equalities (Section 4.2.7) instead substitute much larger program fragments with arbitrary equivalent procedures under explicit programmer control; and because instruction selection is externalized to user code rather than confined to the compiler backend, these substitutions can be interleaved with the further scheduling transformations of Chapter 5. TVM provides a related "tensorization" directive for replacing loop fragments with instructions asserted as equivalent [44], but it lacks the combination of automation and checking that Exo's unification procedure provides.

HPC Libraries BLAS pioneered the standardization of HPC kernels [69,143–145]. Many BLAS libraries have been developed by hardware vendors and optimized for their particular hardware, such as Intel's oneMKL [146], Apple's Accelerate Framework [147], and NVIDIA's cuBLAS [148]. Open-source implementations like ATLAS [149], GotoBLAS [36,150] (now OpenBLAS [6]), and BLIS [37] have introduced various techniques for high-performance, portable BLAS implementations. Early attempts to generate GEMM microkernels in Exo produced schedules with limited parametrization [151], for want of the reference and inspection mechanisms later introduced in Chapter 5. Unlike BLIS, which reuses *optimized object code* (a microkernel) across kernels, the scheduling libraries of Chapter 5 reuse *optimization strategies* as parameterized metaprogramming functions that transform the object code, adding a second

layer of reuse across kernels. BLIS's microkernel, for instance, is hand-written in assembly for each size, target, and precision; in contrast, our library (Section 5.5.2) parameterizes microkernel generation and produces these variants from a *single* library function. Moreover, the BLAS level-1 library function can be reused to optimize the inner loop of level-2 kernels, since the level-2 inner loop shares the same optimization strategy as level 1 (Section 5.5.2). Such reuse of scheduling strategies is impossible without the metaprogramming capabilities of Chapter 5.

8.5 GPU Kernel-Authoring Languages

While CUDA offers fine-grained control, it remains low-level and error-prone; the bugs it admits—the dataflow, instruction, and synchronization errors catalogued in Section 7.1—motivate the safety guarantees Exo-GPU provides (Chapter 7). Prior work confronts this same tension between low-level control and correctness and resolves it at different levels of abstraction.

Higher-level Abstractions Tile-level DSLs like Triton [8] and Pallas [9] abstract away low-level details by enabling programmers to express computations at the block level, with compilers automatically handling the thread-level mapping. However, these abstractions can hide performance-critical details. When Triton performance fell far behind state-of-the-art for Blackwell GPUs, the team had to invent Gluon [26], a lower-level language that gives users direct control over tile layouts, memory allocation, data movement, and asynchrony, essentially abandoning the higher-level automation to regain the fine-grained control necessary for peak performance. Rather than starting with automation that may later prove inadequate, Exo-GPU deliberately maintains a thin abstraction layer over CUDA. Performance-critical constructs, including control flow and data copies across the memory hierarchy, are made explicit statements in Exo-GPU's imperative object code. This design is consistent with the Exocompilation philosophy of maximizing user control.

Template-based Libraries CUTLASS [7] and ThunderKittens [74] are C++ template-based libraries that raise the abstraction of GPU programming while preserving performance. Instead of hiding CUDA behind rigid abstractions, they provide composable, architecture-aware primitives that integrate with raw CUDA code. This approach lets programmers work at the tile level when convenient yet seamlessly drop to thread-level operations when fine control is needed. CuTe [73], introduced in CUTLASS 3.0, offers flexible abstraction for defining tensor layouts and transformations. While these libraries are effective engineering toolkits, they do not provide safety guarantees or whole-program analyses for asynchronous instructions and synchronization statements. Achieving both peak performance and safety remains a fundamental challenge. Existing approaches either provide limited safety guarantees or lack support for modern GPU features. For instance, Descend [152] introduces Rust-like borrow checking to ensure race freedom but does not support key GPU features such as tensor cores and split barriers. The aforementioned tile- and template-based languages are practical tools yet do not offer safety guarantees. This motivated the collective analysis (Section 7.4) and synchronization checking via abstract machine (Section 7.5).

Exo-GPU differs from these approaches in two key ways. First, Exo-GPU exposes low-level control to the programmer; there is little hidden control flow in GPU IR. In this sense, Exo-GPU follows a WYSIWYG (what you see is what you get) philosophy: the structure of the program directly reflects its execution. This contrasts with Cypress [104], which hides warp specialization as an internal compiler optimization; and with TensorIR [103], which focuses on automatically generating schedules with lowering-based intrinsics. Second, Exo-GPU guarantees sequential-parallel equivalence through a synchronization checker that detects data races, hazards, and other concurrency bugs.

Chapter 9

Concluding Remarks

9.1 Impact in Academia and Industry

Since its open-source release in 2022, Exo has been adopted and extended by researchers at several institutions. In collaboration with the Polytechnic University of Valencia, we published a series of papers optimizing GEMM microkernels for Neon [151] and RISC-V [153]. At the University of Cambridge, I helped build an MLIR backend for Exo, part of an ongoing effort to support Exo-like scheduling within the MLIR ecosystem. Independently, researchers at Toyohashi University of Technology used Exo to build an autotuner [154] and to benchmark SVE microkernels on the Fujitsu A64FX [155]—work we are now extending to Arm SME. And at UC Berkeley, Exo served as the foundation for LLM-driven code generation targeting Gemmini [156].

Exo has also reached production through collaboration with industry partners. Exo-generated kernels ship in Apple's iPhone and Vision Pro as part of their HPC library.

9.2 Limitations and Future Work

Externalizing the threading model. Exo already parallelizes multicore CPU loops soundly. `parallelize_loop()` marks a loop as parallel, and the effect analysis proves its iterations are free of cross-iteration RAW and WAW dependences (Section 4.2.8), but, unlike hardware targets, the threading model itself is not yet externalized. The semantics and primitives of runtimes such as OpenMP, pthreads, and MPI remain fixed in the compiler rather than definable in user code. Externalizing them would close one gap where Exo's backends are still built-in rather than user-defined.

Externalizing numeric types and quantization. In Exo, precision is already a scheduling decision rather than a property of the source algorithm. A kernel is written against the abstract numeric type `R`, and `set_precision` specializes it to a concrete type such as `f32` or `i8`, with casts inserted and checked late in the backend (Chapter 3). What remains built into the compiler is the *vocabulary* of numeric types and their conversions: adding a format, or changing how values are rounded, saturated, or rescaled, means editing the compiler rather than writing a library. This is the numeric counterpart of the memories, instructions, and

configuration state externalized for hardware in Chapter 4 and one of the last places an Exo backend is built-in rather than user-defined. The gap matters most for machine learning, where quantization schemes evolve rapidly and low-bit integer, block-scaled, and microscaling floating-point formats each carry distinct conversion and accumulation semantics. Since both the choice of format and the placement of conversions are performance-critical, letting users define numeric types and their conversion behavior in user code, and schedule where conversions occur, would bring quantization under the same externalized, safety-checked discipline as the rest of Exo.

Expressivity of scheduling primitives. Exo is designed for growth, but it does not solve the problem of expressivity in user-schedulable languages: the finest level of control is still dictated by the set of scheduling primitives. If no composition of primitives achieves a desired transformation, modifying the compiler remains necessary, and a complete set of primitives is unlikely to exist, since each scheduling operation encodes an axiom of program equivalence—an undecidable problem in general. Nonetheless, our primitives proved expressive enough to build libraries that match state-of-the-art HPC performance on multiple platforms.

Scheduling and analysis performance. Composing many fine-grained primitives that guarantee functional equivalence through static analysis can increase compilation time relative to higher-level, built-in operations. This was most evident in complex schedules: our GEMM and unsharp-masking schedules in Chapter 5 took roughly 30 seconds and 2 minutes, respectively. Further overhead comes from our reliance on SMT solvers to check functional equivalence and from Exo being implemented in Python. Value-sensitive analysis (Chapter 6) is more expensive still: the CAD abstract domain is doubly exponential in the worst case, and our slowest benchmark (array initialization with 2D nonconstant loop bounds) took 20 minutes. Restricting the analysis to variables of interest helps, but the cost becomes prohibitive for heavily nested, multidimensional arrays. We plan to reduce these overheads through caching, incremental reanalysis across scheduling steps, and more efficient analysis algorithms.

Integrating value-sensitive reasoning into Exo-GPU. The value-sensitive analysis of Chapter 6 and the GPU backend of Chapter 7 are currently developed independently. Prexo reasons about the values that statements write and composes with Exo's equivalence-preserving primitives, but its abstract values are not yet consulted by Exo-GPU's rewrite and synchronization checks, which remain purely dependency-based. Integrating the two would let value information justify transformations that reasoning about disjoint locations alone cannot, and this is increasingly necessary on modern accelerators. Features such as Hopper's asynchronous TMA copies and RVV's masked instructions let programmers choose what happens at the boundaries of a computation (writing zero, a NaN, or leaving elements undisturbed), and those values then propagate through later stages; only a value-sensitive analysis can certify that special values such as NaNs never silently contaminate valid data.

Reasoning about GPU parallelism. Our reasoning about parallelism is incomplete in several ways. Our programming model and generated CUDA synchronization code are

based on our reading of NVIDIA's natural-language PTX documentation [71] rather than any formal CUDA semantics, so we do not prove our generated code correct with respect to such a semantics. We do not yet support Blackwell tensor cores (`tcgen05`), nor platforms without CUDA C++ and inline PTX support. Finally, our synchronization checker—which translates the program to an abstract-machine IR and interprets it in software—handles only concrete problem sizes and limits compiler performance; static analysis of the abstract-machine program could remove this restriction.

Automatic scheduling. We have not yet built autoschedulers for Exo [17,18,20,21], but we expect Exo autoscheduling to differ from prior systems in two essential ways. First, because hardware targets are externalized, idiosyncratic, and often proprietary, no single autoscheduling strategy is likely to work across all accelerators. Second, because Exo schedules are composable rewrites rather than monolithic, autoscheduling can be built compositionally, on top of scheduling operators that already exist. Chapter 5 shows that libraries of reusable mid-level scheduling abstractions can be assembled from primitives and shared across dozens of kernels. Today these libraries are written by hand; the open question is how to *generate* them, and the schedules that apply them, automatically. Classical autoscheduling search and, increasingly, LLM-driven synthesis are natural candidates, and early work already drives code generation for Gemmini starting from Exo-generated code [156]. Because every step remains a safety-checked rewrite, such automation inherits Exo's equivalence guarantees instead of having to reestablish them.

9.3 Closing Vision

Performance is the currency of modern computing, and realizing it on today's accelerators has meant hand-writing and endlessly retuning low-level C and assembly, a burden that grows with every new hardware generation. This thesis argued that the way forward is not more automation but a cleaner division of labor. Following the Exocompilation philosophy, Exo keeps built-in compiler automation minimal, limited to safety analysis, and hands the decisions that determine performance—instruction selection, data movement across the memory hierarchy, and scheduling—to the performance engineer. Automation is not abandoned but reintroduced external to the compiler as reusable user-defined libraries built from safety-checked primitives.

The two externalizations at the heart of this approach move hardware definitions (Chapters 3, 4, and 7) and optimization strategies (Chapters 5 and 6) out of the compiler and into user code. Together they keep the core compiler small, auditable, and predictable, while letting richer automation grow in libraries that anyone can extend. Using this approach, we externalized backends for a wide range of real vector, matrix, and GPU engines: AVX2, AVX-512, Arm Neon, RVV, Arm SME, Gemmini, Apple's outer-product engine, Intel AMX, and Hopper tensor cores. These matched or exceeded state-of-the-art libraries while reducing code size, and on top of them we built scheduling libraries and value-sensitive analyses that make optimizations such as vectorization, bound inference, and sliding-window transformations extensible in user code rather than frozen in the compiler.

More than any single kernel or benchmark, we hope the lasting contribution of this

work will be the idea of building reusable libraries from fine-grained primitives: libraries of hardware definitions and of scheduling abstractions, not just individual kernels tuned one-at-a-time. We believe this can reshape how high-performance code is written. Instead of performance engineers optimizing kernels one-by-one while chasing correctness bugs, we envision a workflow that separates three responsibilities: language designers provide primitives and guarantee their safety; scheduling-library authors write target- and application-specific optimization strategies; and HPC-library authors compose these building blocks to optimize kernels productively. As accelerators continue to proliferate and expose ever more of their internals to software, we believe this separation offers a future-proof foundation for building reliable, high-performance systems across a rapidly changing hardware landscape: maximal control for the programmer, minimal and verifiable automation in the compiler, and reusable optimization living outside it.

Appendix A

Externalizing Optimization from the Compiler

A.1 Matrix-multiply on Gemmini

This section provides a step-by-step guide to optimizing matrix multiplication for the Gemmini hardware accelerator. We start with a simple GEMM object code and progressively transform it to leverage Gemmini's unique hardware features. The code snippets on the right illustrate both the object and the scheduling code at each stage of the optimization process. Exo object code is enclosed in boxes, while Exo scheduling code has a vertical line on the left. Most of the scheduling functions used in this example are written in user-code, as part of Exo's Gemmini library.

Initial Object Code. The starting point is a standard GEMM object code with additional post-processing steps specific to machine learning applications. These steps include scaling, clamping, and applying a ReLU activation function before storing the final result. The scaling and clamping operations ensure correct quantization for integer matrix multiplication, while the activation function is a common component in machine learning pipelines.

```
def matmul_on_gemmini(N: size, M: size, scale: f32,
    act:bool, A:i8[N,512], B:i8[512,M], C:i8[N,M]):
  assert N % 256 == 0
  assert M % 256 == 0
  for i in seq(0, N):
    for j in seq(0, M):
      res: i32 @ DRAM
      res = 0.0
      for k in seq(0, 512):
        a2: i32 @ DRAM
        b2: i32 @ DRAM
        a2 = A[i, k]
        b2 = B[k, j]
        res += a2 * b2
      src_tmp: i32 @ DRAM
      src_tmp = res
      tmp_res1: f32 @ DRAM
      acc_scale(src_tmp, tmp_res1, scale)
      tmp_res2: i8 @ DRAM
      clamp(tmp_res1, tmp_res2)
      if act == True:
        tmp_res2 = relu(tmp_res2)
      C[i, j] = tmp_res2
```

Hardware Constraints and Cursors. Gemmini has a software-managed scratchpad of size 256KB and an accumulator of size 16KB. To efficiently utilize these hardware resources, the memory layout of the scratchpad and accumulator must have innermost dimensions of 16. We obtain cursors to key points in the object code, which will be used throughout the scheduling process to navigate and transform the code.

```
# Parameters
accum_size= 16 * 1024  # 16 KB
sc_size   = 256 * 1024 # 256 KB
# Grab cursors
i         = p.find_loop("i")
j         = p.find_loop("j")
k         = p.find_loop("k")
res_load  = p.find("res = 0.0")
a_assign  = p.find("a2 = A[_]")
b_assign  = p.find("b2 = B[_]")
res_alloc = res_load.prev()
```

Tiling for Scratchpad and Accumulator. To ensure that the working set fits into the scratchpad and accumulator, we tile the loops using the `tile_loops`, which is a user-level Gemmini library function. This function divides each loop into an outer and inner loop, rearranges the outer loops to be outside the inner loops, and returns cursors to the inner loops. In the provided schedule, we apply two levels of tiling to the `i` and `j` loops, while the `k` loop is simply divided by 16.

```
p, _ = tile_loops(p, [(i,16), (j,16)], perfect=True)
p, [_, j_outer] = tile_loops(p, [(i,16), (j,16)],
                                                perfect=True)
p, _ = tile_loops(p, [(k, 16)])
```

Memory Allocation and Lifting. `autolift_alloc` and `bind_and_lift` allocate buffers in the scratchpad and accumulator. `autolift_alloc` lifts and expands buffer dimensions within the specified threshold (accumulator and scratchpad sizes). `bind_and_lift` creates and lifts a temporary buffer using `autolift_alloc`. We allocate an `i32[16, 16, 16]` buffer in the accumulator, matching its size. For input matrices `A` and `B`, we bind them to temporary scratchpad memory and lift the allocations. The resulting `A_tmp` and `B_tmp` buffers each occupy 128KB, half of the scratchpad.

```
p = autolift_alloc(p, res_alloc, max_size=accum_size)
p, a_load, a_alloc = bind_and_lift(p,
         p.forward(a_assign).rhs(), max_size=sc_size)
p, b_load, b_alloc = bind_and_lift(p,
         p.forward(b_assign).rhs(), max_size=sc_size)
```

Utilizing Blocked Load Instructions. Although Gemmini's matrix multiplication instruction operates on 16×16 tiles, it supports larger blocked loads of size $4 \times 16 \times 16$ in a single cycle. To take advantage of this feature, we further divide the `k` and `j` loops by 4.

```
p, _ = tile_loops(p, [(k_loop, 4)])
p, [j_imost] = tile_loops(p, [(j_outer, 4)],
                                    perfect=True)
```

Loop Fission and Instruction Mapping. We apply loop fission to each code block, isolating individual chunks that correspond to Gemmini instructions.

```
p = fission_as_much_as_possible(p, res_load)
p = fission_as_much_as_possible(p, k)
p = fission_as_much_as_possible(p, a_load)
p = fission_as_much_as_possible(p, b_load)
```

Data Layout Transformation. The dimensions of `A_tmp` and `B_tmp` buffers need to be reordered to match the expected layout of Gemmini instructions. Specifically, we use the `rearrange_dim` function to ensure that the innermost two dimensions of `A_tmp` are along the `i` and `k` axes, while dimensions for `B_tmp` are along the `k` and `j` axes.

```
p = rearrange_dim(p, a_alloc, [0, 2, 1, 3])
p = rearrange_dim(p, b_alloc, [2, 0, 3, 1])
p = reorder_loops_from_idx(p, a_load)
p = reorder_loops_from_idx(p, b_load)
p = reorder_loops_from_idx(p,
                  p.forward(a_assign).rhs())
p = remove_redundant_loops(p, a_load, num=2)
p = remove_redundant_loops(p, b_load, num=1)
```

Replacing with Gemmini Instructions and Hoisting Configuration. Gemmini instructions require setting certain configuration states. In Exo, the instructions are represented by pairs of functionally equivalent procs, with one version having additional configuration state updates. For instance, `ld_i8_block_id1_v2` writes to a configuration stride register before calling `ld_i8_block_id1`. We use the `replace_and_inline` function to first insert calls to the `_v2` instructions, and then inline to replace it with a configuration update and the original instructions. Then, the configuration instructions are hoisted out of the loops to minimize their overhead (see Section 5.5.1 for user-level scheduling functions that support configuration hoisting).

```
p = set_memory(p, res_alloc, GEMM_ACCUM)
p = set_memory(p, a_alloc, GEMM_SCRATCH)
p = set_memory(p, b_alloc, GEMM_SCRATCH)
tuples = [
    (zero_acc_i32, zero_acc_i32_v2),
    (ld_i8_block_id1, ld_i8_block_id1_v2),
    (ld_i8_block_id2, ld_i8_block_id2_v2),
    (matmul_acc_i8, matmul_acc_i8_v2),
    (st_acc_i8, st_acc_i8_v2)]
for t in tuples:
    p = simplify(replace_and_inline(p, t))
```

Optimizing Memory Access. We add guards to avoid issuing redundant load instructions while keeping the loop structure intact for further optimization.

```
p = add_guard(p, p.find("do_ld_i8_block_id1(_)"))
p = add_guard(p, p.find("do_ld_i8_block_id2(_)"))
```

Loop Fusion and Unrolling. To prevent the reorder buffer (ROB) from being filled with only load instructions, we fuse the loops to achieve a balanced mix of load, store, and compute instructions. This improves locality and exploits instruction-level parallelism. We also unroll all the innermost loops to reduce loop overhead.

```
p = fuse_all_loops(p, p.body()[0])
p = unroll_all(p, p.find_loop(j_imost.name(),
                        many=True))
```

Final object code. The final object code, just before the unrolling step, is shown on the right. This function takes arbitrary N and M as long as they are multiples of 256 (lines 3-4). As we showed in Section 5.5.1, configuration instructions are hoisted at the beginning of the kernel, eliminating the overhead of reconfiguration (lines 5-9). Buffers for A, B, and C are allocated in the Gemmini accumulator and scratchpad memories (lines 10-12). In the main loop, it first initializes the accumulator to zero (lines 17-19), loads A and B to the scratchpad (lines 21-33), and performs matrix multiplication on the accumulator (lines 34-38). Finally, it stores the result from the accumulator to C (lines 39-44). This optimized code efficiently utilizes Gemmini's scratchpad and accumulator memories and leverages its instructions and configuration states to achieve high-performance matmul.

```
1  def matmul_on_gemmini(N: size, M: size, scale: f32,
2      act:bool, A:i8[N,512], B:i8[512,M], C:i8[N,M]):
3    assert N % 256 == 0
4    assert M % 256 == 0
5    config_st_acc_i8(scale, stride(C, 0), act)
6    config_matmul()
7    config_ld_i8_id2(stride(B, 0))
8    config_ld_i8_id1(stride(A, 0))
9    config_zero()
10   res: i32[16, 16, 16] @ GEMM_ACCUM
11   A_tmp: i8[16, 32, 16, 16] @ GEMM_SCRATCH
12   B_tmp: i8[32, 16, 16, 16] @ GEMM_SCRATCH
13   for ioo in seq(0, N / 256):
14     for joo in seq(0, M / 256):
15       for ioi in seq(0, 16):
16         for joio in seq(0, 4):
17           for joii in seq(0, 4):
18             do_zero_acc_i32(16, 16,
19                         res[joii+4*joio, 0:16, 0:16])
20           for koo in seq(0, 8):
21             if joo == 0:
22               if joio == 0:
23                 do_ld_i8_block_id1(16, 4,
24                   A[16*ioi+256*ioo:16+16*ioi+256*ioo,
25                                     64*koo:64+64*koo],
26                   A_tmp[ioi,4*koo:4+4*koo,0:16,0:16])
27             for koi in seq(0, 4):
28               if ioi == 0:
29                 do_ld_i8_block_id2(16, 4,
30                   B[16*koi+64*koo:16+16*koi+64*koo,
31                   64*joio+256*joo:64+64*joio+256*joo],
32                   B_tmp[koi+4*koo, 4*joio:4+4*joio,
33                                            0:16, 0:16])
34               for joii in seq(0, 4):
35                 do_matmul_acc_i8(16, 16, 16,
36                   A_tmp[ioi, koi+4*koo, 0:16, 0:16],
37                   B_tmp[koi+4*koo, joii+4*joio, 0:16,
38                   0:16], res[joii+4*joio, 0:16, 0:16])
39           for joii in seq(0, 4):
40             do_st_acc_i8(16, 16,
41                   res[joii+4*joio, 0:16,0:16],
42                   C[16*ioi+256*ioo:16+16*ioi+256*ioo,
43                     16*joii+64*joio+256*joo:
44                     16+16*joii+64*joio+256*joo])
```

A.2 Matrix-multiply on AVX-512

Initial Object Code. This section provides a step-by-step guide for optimizing matrix multiplication on AVX-512. Our starting point is the simple triple nested-loop matrix multiplication implemented as an outer product. This section does not include every code snippet evaluated in Section 5.5.2. Specifically, it omits the code for the bottom and right panels. However, these panels follow a strategy similar to that of the main panel.

```
@proc
def SGEMM(M: size, N: size, K: size,
          A: f32[M, K], B: f32[K, N],
          C: f32[M, N]):
    for k in seq(0, K):
        for i in seq(0, M):
            for j in seq(0, N):
                C[i, j] += A[i, k] * B[k, j]
```

Hardware Constraints. Before diving into the optimization process, we define hardware constraints such as the vector width (`VEC_W`), register blocking factors (`M_REG_BLK`, `N_REG_BLK`), and L1 cache tiling factors (`M_L1_FAC`, `N_L1_FAC`, `K_L1_BLK`). Our optimization strategy consists of two main stages. First, we tile the loops to optimize for the memory hierarchy, improving data locality and reducing cache misses. Second, we compute the matrix-multiplication of these tiles by performing register-tiling.

```
VEC_W = 16
M_REG_BLK = 6
N_REG_BLK = 4 * VEC_W
M_L1_FAC = 44
N_L1_FAC = 1
M_L1_BLK = M_REG_BLK * M_L1_FAC
N_L1_BLK = N_REG_BLK * N_L1_FAC
K_L1_BLK = 512
```

Memory System Blocking and Staging. Memory-level tiling decomposes the overall multiplication into `M_L1_BKL x N_L1_BLK x K_L1_BLK` subproblems. We start from the triple-nested outer product matrix multiplication, and we tile it to generate the main tile case and the various tail cases. Then, we stage the `B` tile into local static memory for each of the cases. We only stage the `A` tile for the main tile. We then hoist the loop nest of the load stage of `A` since it is idempotent with respect to the innermost loop. Finally, we replace all the main tile case and all the tail cases with the scheduled matrix-multiplication that calls to the various microkernels that are scheduled in the following paragraphs.

```
p = rename(SGEMM, "sgemm_exo")
p = tile_loops_bottom_up(p, p.body()[0],
                         (K_L1_BLK, M_L1_BLK, N_L1_BLK))
for i in range(0, 8):
  B_name = "B_cache"
  p, (alloc, _, _, _) = auto_stage_mem(p,
      p.find_loop(f"ki #{i}"), "B", B_name, rc=True)
  p = resize_dim(p, alloc, 0, K_L1_BLK, 0)
  p = resize_dim(p, alloc, 1, N_L1_BLK, 0)
  p = set_memory(p, alloc, DRAM_STATIC)
p, (alloc, _, _, _) = auto_stage_mem(p,
        p.find_loop("jo"), "A", "A_cache", rc=True)
p = resize_dim(p, alloc, 0, M_L1_BLK, 0)
p = resize_dim(p, alloc, 1, K_L1_BLK, 0)
p = set_memory(p, alloc, DRAM_STATIC)
p = fission(p, p.find_loop("jo").after(), n_lifts=2)
p = replace_all(p, SGEMM_WINDOW)
p = repeat(call_eqv)(p, SGEMM_WINDOW,
                                  sgemm_above_kernel)
```

Micro-Kernel Generation Function. Register-level tiling (micro-kernel) computes `N_REG_BKL x M_REG_BLOCK` of `C` as an outer product of micropanels from matrices `A` and `B`. Our primary micro-kernel computes a 6×64 microtile of C. For tail cases, we generate additional micro-kernels of size $m \times 16n$ where either $m = 6$ and $n \leq 4$, or $m \leq 6$ and $n = 4$. Microtiles with the second dimension with not multiples of 16 are rounded up and dispatched to the nearest appropriate micro-kernel.

A single, parameterized function `gen_ukernel` generates schedules for all micro-kernels. The parameter `cond` represents the multiple of 16 to which n rounds up. The schedule first asserts conditions about `cond`, rounds the j loop to a multiple of 16, stages the `C` microtile, and vectorizes the loops for loading, outer product computation, and storing. After applying dead code elimination to remove unnecessary branches, it replaces SIMD loops with AVX-512 instructions.

```
def gen_ukernel(p, cond):
  og_p = p.add_assertion(cond)
  p = round_loop(og_p, og_p.find_loop("j"), VEC_W)
  p = simplify(specialize(p, p.find_loop("k"), cond))
  p = dce(p)

  p = simplify(auto_stage_mem(p, p.find_loop("k"),
                                        "C", "C_reg"))
  p = simplify(divide_dim(p, "C_reg", 1, VEC_W))
  p = set_memory(p, "C_reg", AVX512)

  rules = [fma_rule]
  for it in ("i1", "j", "i1"):
    p = vectorize(
      p, p.find_loop(it), VEC_W, "f32", AVX512,
                        rules=rules, tail="perfect")
  p = hoist_from_loop(p, p.find_loop("jo"))
  for it in ("i1o", "jo", "i1o"):
    p = unroll_loop(p, it)

  p = simplify(dce(p))
  p = replace_all(p, AVX512F_instructions)
  return og_p, simplify(p)
```

Generating Micro-Kernels. We use the `gen_ukernel` function to create all necessary micro-kernels. This is done by calling the function with procedures that are partially evaluated for different values of M. For the main and bottom micro-kernels, which don't require predicates, we set `cond` to True (`"0 == 0"`). When generating right panels, we use an actual condition for `cond`.

The resulting micro-kernels are stored in the `basic_kernel_Mx4` and `sgemm_kernel_avx512_Mx4` dictionaries for later use.

```
basic_kernel_Mx4 = {}
sgemm_kernel_avx512_Mx4 = {}
for M in range(1, M_REG_BLK + 1):
  def make_basic(p):
    p = rename(p, f"basic_kernel_{M}x4")
    p = p.partial_eval(M, N_REG_BLK)
    p = simplify(p)
    return p

  p = make_basic(SGEMM_WINDOW)
  p, p_sched = gen_ukernel(p, "0 == 0")
  basic_kernel_Mx4[M] = p
  sgemm_kernel_avx512_Mx4[M] = p_sched
```

Micro-Kernel Object Code. The code on the right shows an example of a generated micro-kernel for a 6×64 micro-tile. The kernel assumes that the input matrices A and B, as well as the output matrix C, are stored in row-major order with unit stride (lines 3-6). The micro-tile of C is staged into a local register array `C_reg` (line 7), which is then used for loading from C (lines 8-12), accumulating the results of the outer product between micropanels of A and B (lines 13-32), and storing the result back to C (lines 33-37). The computation is fully vectorized using AVX-512 intrinsics, with each iteration of the innermost loop processing a 16-element vector. By applying the aforementioned optimizations, the matrix multiplication code is efficiently mapped to the AVX-512 instruction set, taking full advantage of the available vector registers and SIMD parallelism.

```
1  def basic_kernel_6x4(K: size, A: [f32][6, K] @ DRAM,
2       B: [f32][K, 64] @ DRAM, C: [f32][6, 64] @ DRAM):
3    assert K >= 1
4    assert stride(A, 1) == 1
5    assert stride(B, 1) == 1
6    assert stride(C, 1) == 1
7    C_reg: f32[6, 4, 16] @ AVX512
8    for i0 in seq(0, 6):
9      mm512_loadu_ps(C_reg[i0, 0, 0:16], C[i0, 0:16])
10     mm512_loadu_ps(C_reg[i0, 1, 0:16], C[i0, 16:32])
11     mm512_loadu_ps(C_reg[i0, 2, 0:16], C[i0, 32:48])
12     mm512_loadu_ps(C_reg[i0, 3, 0:16], C[i0, 48:64])
13   for k in seq(0, K):
14     for i in seq(0, 6):
15       var0: f32[16] @ AVX512
16       mm512_set1_ps(var0[0:16], A[i, k:1 + k])
17       var1: f32[16] @ AVX512
18       mm512_loadu_ps(var1[0:16], B[k, 0:16])
19       mm512_fmadd_ps(var0[0:16], var1[0:16],
20                                     C_reg[i, 0, 0:16])
21       var1_1: f32[16] @ AVX512
22       mm512_loadu_ps(var1_1[0:16], B[k, 16:32])
23       mm512_fmadd_ps(var0[0:16], var1_1[0:16],
24                                     C_reg[i, 1, 0:16])
25       var1_2: f32[16] @ AVX512
26       mm512_loadu_ps(var1_2[0:16], B[k, 32:48])
27       mm512_fmadd_ps(var0[0:16], var1_2[0:16],
28                                     C_reg[i, 2, 0:16])
29       var1_3: f32[16] @ AVX512
30       mm512_loadu_ps(var1_3[0:16], B[k, 48:64])
31       mm512_fmadd_ps(var0[0:16], var1_3[0:16],
32                                     C_reg[i, 3, 0:16])
33   for i0 in seq(0, 6):
34     mm512_storeu_ps(C[i0, 0:16], C_reg[i0, 0, 0:16])
35     mm512_storeu_ps(C[i0, 16:32],C_reg[i0, 1, 0:16])
36     mm512_storeu_ps(C[i0, 32:48],C_reg[i0, 2, 0:16])
37     mm512_storeu_ps(C[i0, 48:64],C_reg[i0, 3, 0:16])
```

A.3 BLAS

A.3.1 Level 1 Specific Scheduling Operator

```
1  def optimize_level_1(proc, loop, precision, machine, interleave_factor,
2                                                  vec_tail=None, inter_tail="recursive"):
3      vec_width = machine.vec_width(precision)
4      instrs = machine.get_instructions(precision)
5      memory = machine.mem_type
6      patterns = machine.patterns
7
8      if vec_tail is None:
9          vec_tail = "cut_and_pred" if machine.supports_predication else "cut"
10
11     loop = proc.forward(loop)
12     proc = CSE(proc, loop.body(), precision)
13
14     proc, (loop,) = vectorize(proc, loop, vec_width, memory, patterns, vec_tail)
15
16     proc, (_, loop) = LICM(proc, loop, rc=True)
17
18     proc = interleave_loop(proc, loop, interleave_factor, memory, inter_tail)
19
20     proc = cleanup(proc)
21     proc = replace_all_stmts(proc, instrs)
22     return proc
```

`optimize_level_1` is a scheduling operator we defined to optimize the entire BLAS level 1, achieving competitive performance against existing libraries across the board (Figures A.1, A.2, and A.3). This operator optimizes a loop for a target machine at a given precision, taking arguments such as the interleaving factor and tail strategies. The implementation closely mirrors the body of the chapter as discussed in Section 5.5.2. First, it extracts machine information, including vector width, instructions, and memory type (lines 3-6). For simplicity, CSE is performed before vectorization (line 12), and the `vectorize` function autovectorizes the loop as described in Section 5.5.1 (line 14). LICM then hoists broadcast instructions after vectorization (line 16). Loop iterations may be interleaved (line 18) to exploit SIMD vector parallelism. Finally, `replace_all_stmts` is called to replace the code with machine-specific vector instructions.

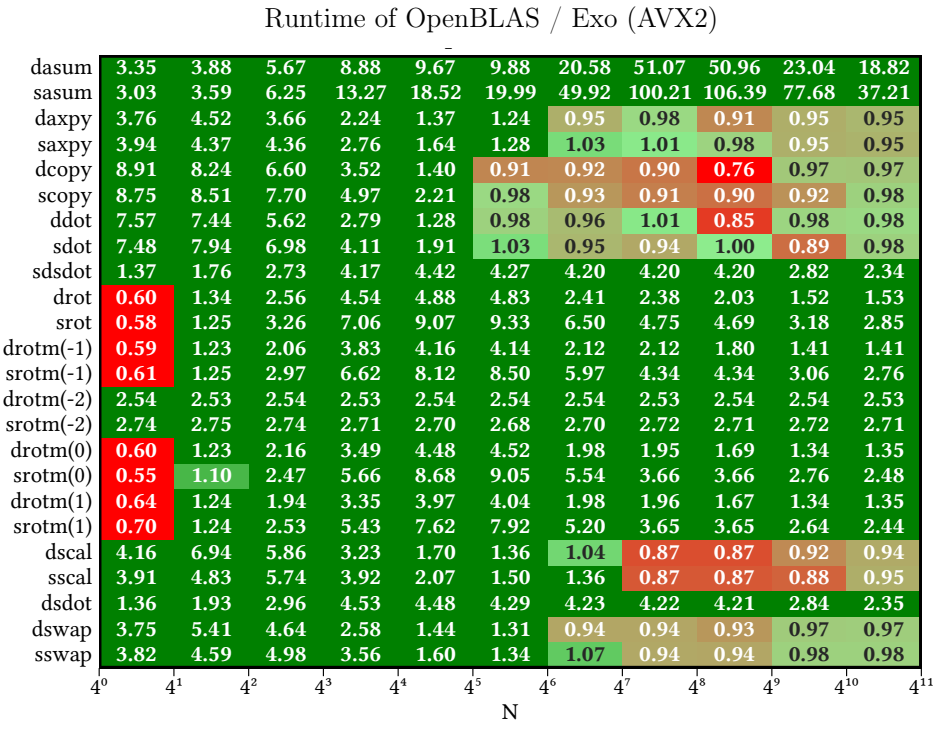


Figure A.1: Exo BLAS level 1 performance compared to OpenBLAS (top) and BLIS (bottom) using AVX2. Each heatmap cell represents the geometric mean over problems fitting into the respective x-axis bucket. Higher is better.

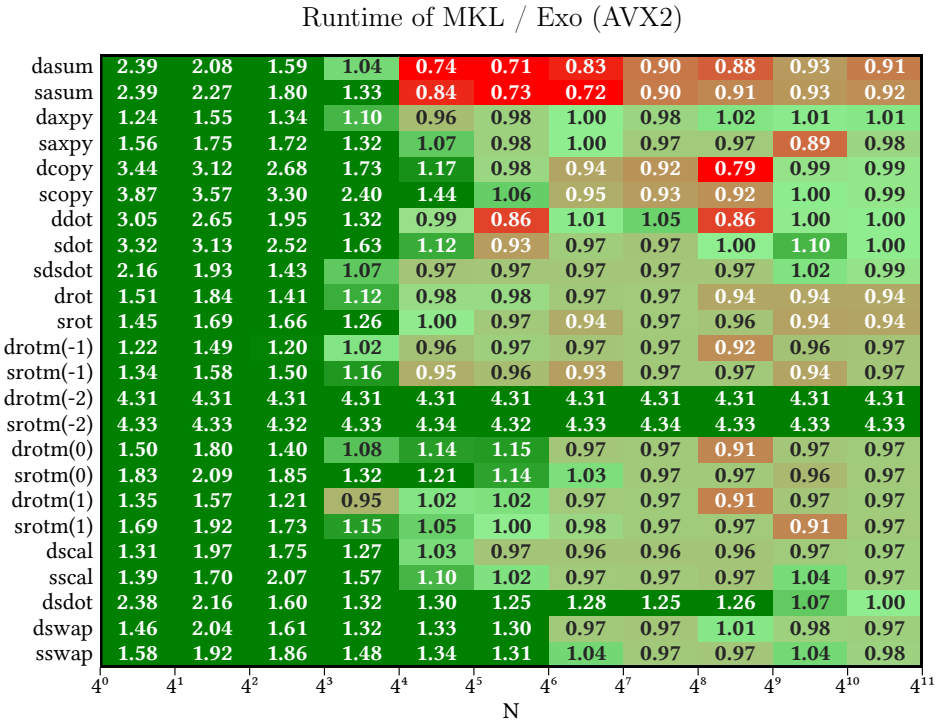
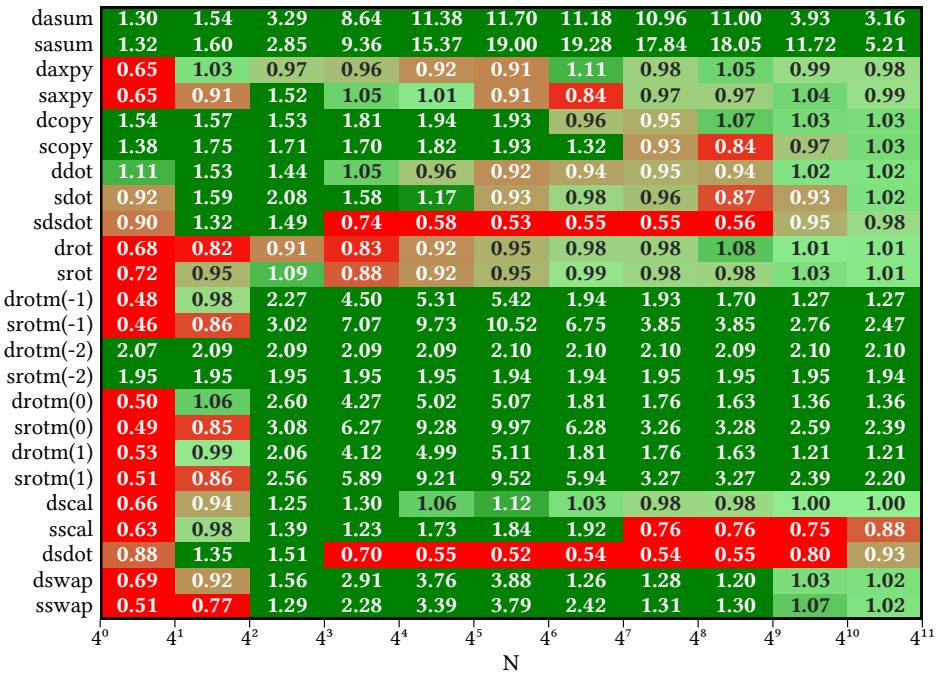


Figure A.2: Exo BLAS level 1 performance compared to MKL using AVX2 (top) and AVX-512 (bottom).

Runtime of OpenBLAS / Exo (AVX-512)



Runtime of BLIS / Exo (AVX-512)

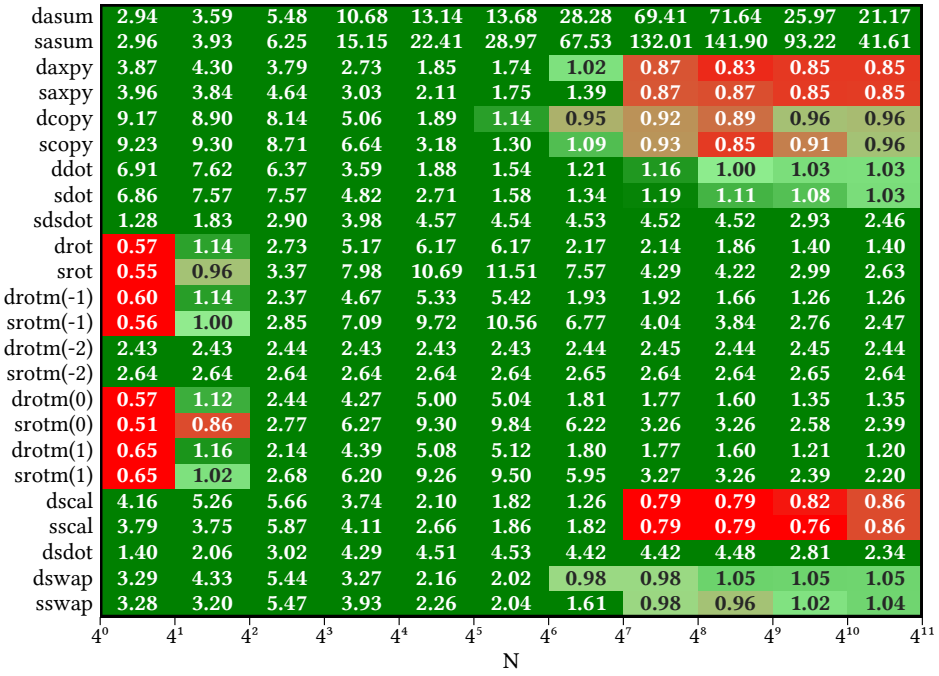


Figure A.3: Exo BLAS level 1 performance compared to OpenBLAS (top) and BLIS (bottom) using AVX-512.

A.3.2 Level 2 Specific Scheduling Operator

```
1  def optimize_level_2_general(proc, o_loop, precision, machine,
2                                              r_fac, c_fac, round_up=None):
3      o_loop = proc.forward(o_loop)
4
5      proc, (o_loop,) = adjust_triang(proc, o_loop, machine, r_fac, round_up)
6
7      proc = unroll_and_jam(proc, o_loop, r_fac)
8      inner_loop = get_inner_loop(proc, o_loop)
9      proc = optimize_level_1(proc, inner_loop, precision, machine, c_fac)
10
11     tail = get_inner_loop(proc, proc.forward(o_loop).next())
12     proc = optimize_level_1(proc, tail, precision, machine, c_fac)
13     return cleanup(proc)
```

`optimize_level_2_general` optimizes 50 kernel variants, supporting most BLAS level-2 operations (excluding banded operations and packed-triangular formats) across all configurations. The performance is on par with existing libraries, as shown in Figures A.4, A.5, and A.6. The function takes the outer loop (`o_loop`) of the level 2 kernel as input, along with the precision at which the kernel is computed and the target machine. Additionally, users can specify the number of rows (`r_fac`) to be batched and the number of columns (`c_col`) to be interleaved. A `round_up` parameter is provided to allow users to override the rounding behavior in triangular matrices. The implementation is simple and closely follows the body of the chapter, as discussed in Section 5.5.2. First, the iteration space is adjusted for triangular matrices (line 5). Then, unroll-and-jam is performed to transform the loop into a batched program of multiple dot products (line 7). Finally, `optimize_level_1` is called on the resulting inner loop (line 12).

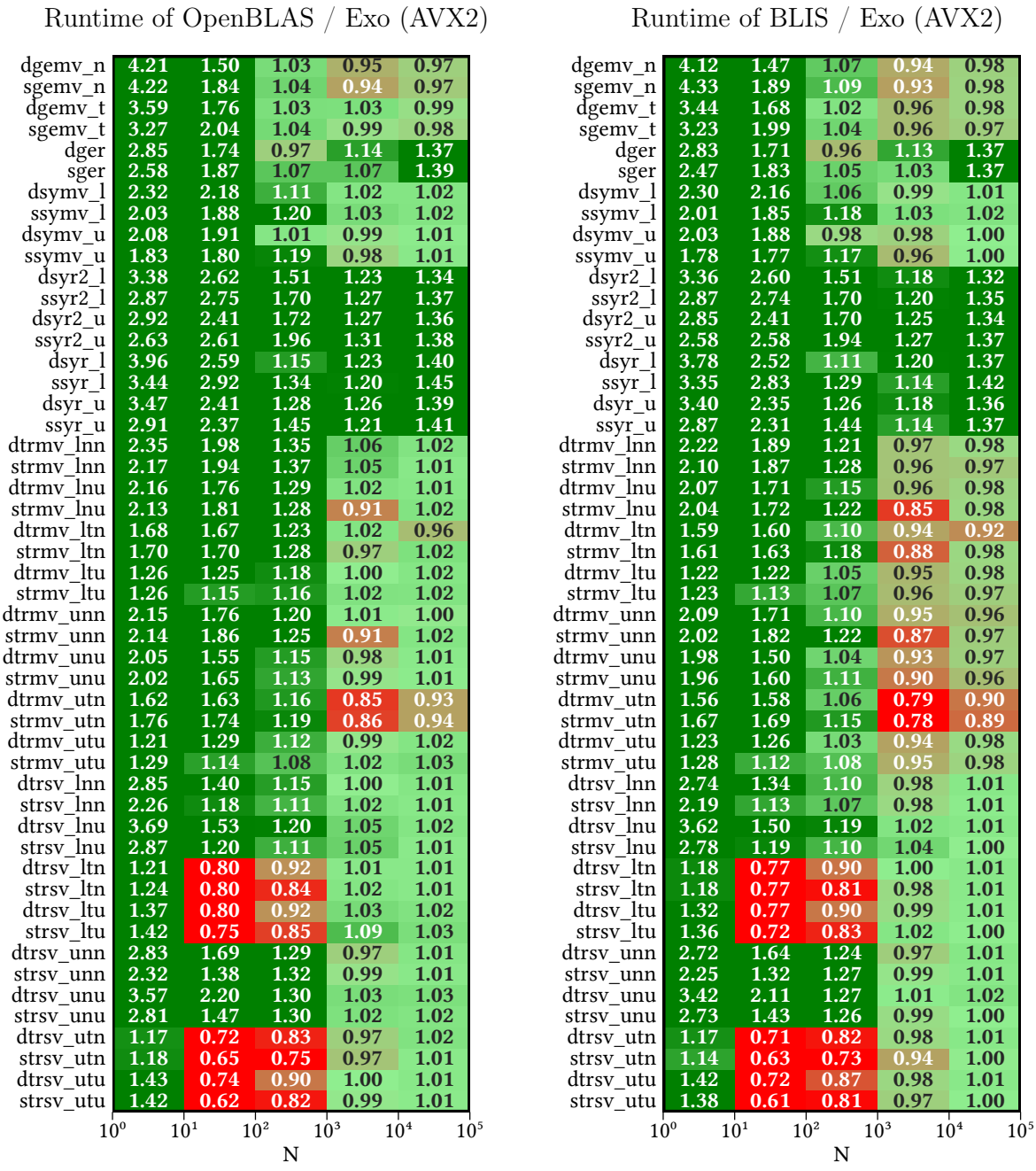


Figure A.4: Exo BLAS level 2 performance compared to OpenBLAS (left) and BLIS (right) using AVX2.

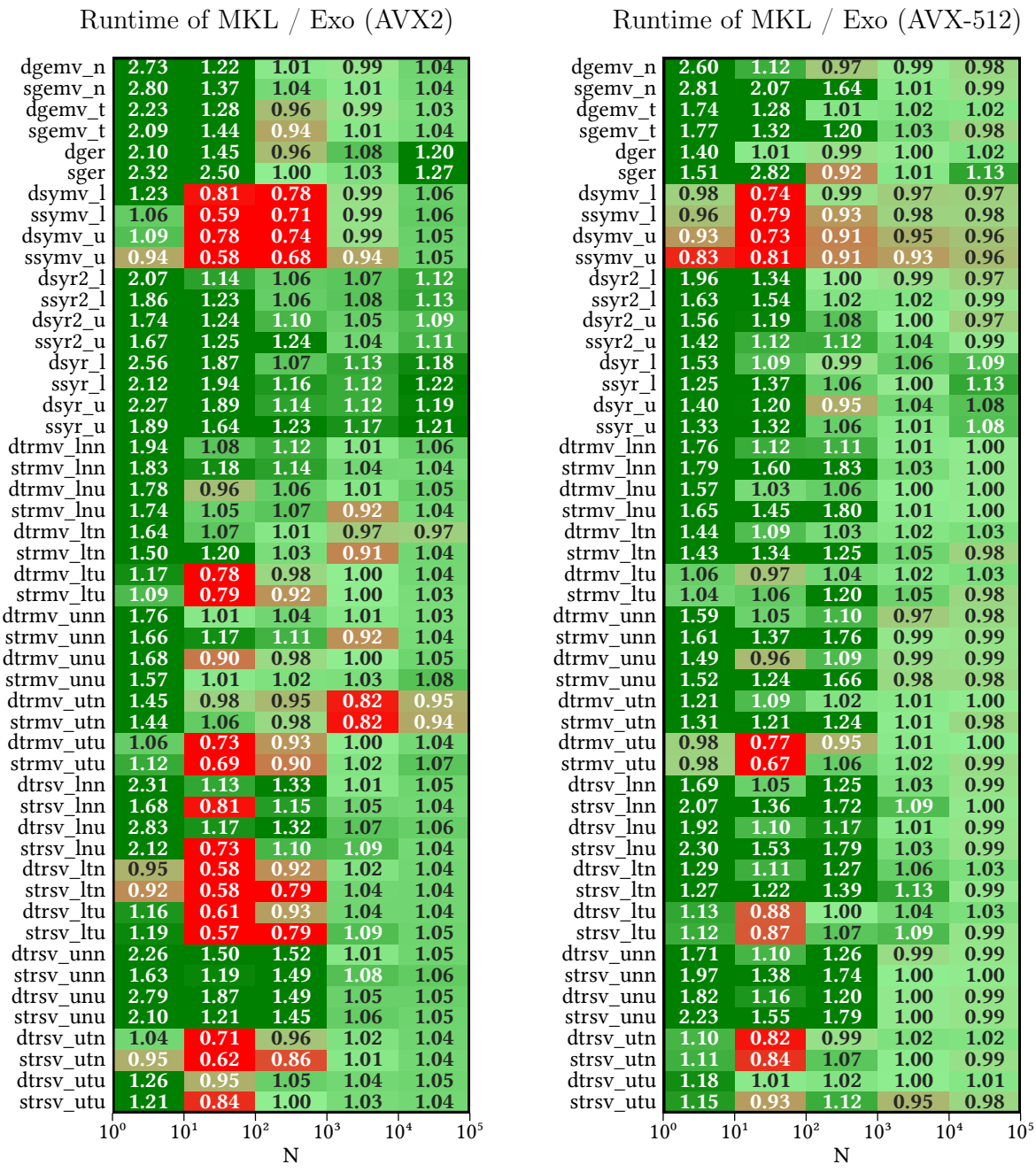


Figure A.5: Exo BLAS level 2 performance compared to MKL using AVX2 (left) and AVX-512 (right).

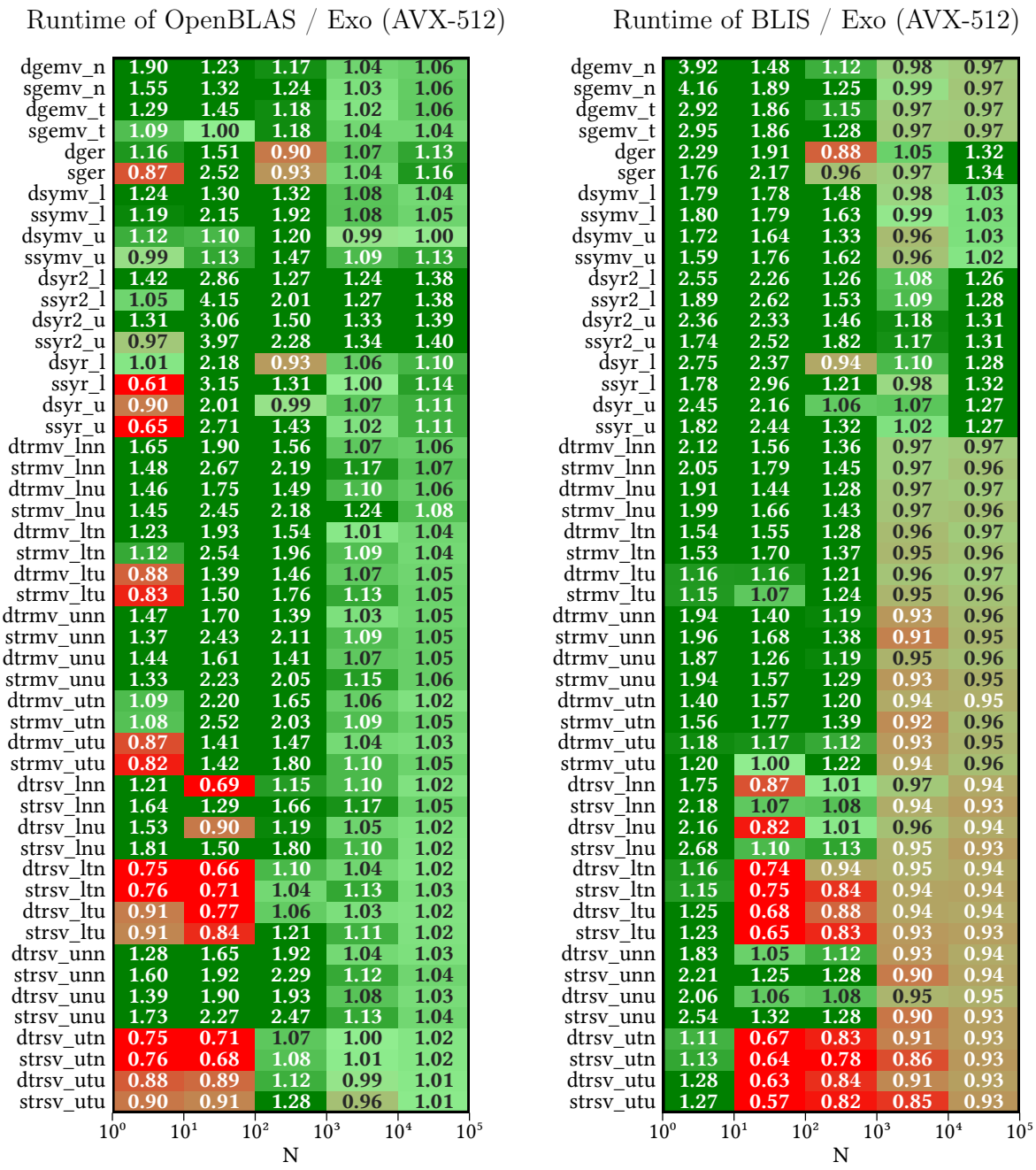


Figure A.6: Exo BLAS level 2 performance compared to OpenBLAS (left) and BLIS (right) using AVX-512.

Appendix B

Value-Sensitive Analysis for Looping Array Code

B.1 Monotonicity Proof

Lemma B.1 (Monotonicity of expression semantics)**.** *For every control expression c, $\mathcal{E}_c^e[\![c]\!] : \Sigma_c \to \mathbb{Z}_\perp^\top$ is monotone. For every data expression e, $\mathcal{E}_d^e[\![e]\!] : \Sigma_c \times \Sigma_d \to \mathbb{D}_\perp^\top$ is monotone. For every boolean expression b, $\mathcal{E}_b[\![b]\!] : \Sigma_c \to \mathbb{B}_\perp^\top$ is monotone.*

Proof. By structural induction. Constants are constant; variables are store lookups, hence monotone. Composite forms apply lifted operators/relations and the lifted array read, each monotone in its arguments by case analysis on $\perp \sqsubseteq v \sqsubseteq \top$ (strict in $\perp$, absorbing at $\top$). □

Lemma B.2 (Monotonicity of statement semantics)**.** *For every statement s, $\mathcal{E}_c[\![s]\!] : \Sigma_c \to \Sigma_c$ is monotone, and $\mathcal{E}_d[\![s]\!] : \Sigma_c \times \Sigma_d \to \Sigma_d$ is monotone in both arguments.*

Proof. By induction on s. Sequential composition uses composition of monotone maps. For a control assignment $y = \lambda\imath^*.(b?c_t : c_f)$, the updated array a_y is defined pointwise via ite applied to monotone subterms (Lemma B.1), hence the store update is monotone. For a data assignment $x = \lambda\imath^*.\lambda\eta^*.(b?e_t : e_f)$, by pointwise order on arrays a_x the store update $(\sigma_c, \sigma_d) \mapsto \sigma_d[x \mapsto a_x]$ is monotone in both arguments. Control assignments leave σ_d unchanged and data assignments leave σ_c unchanged, which is trivially monotone. □

Lemma B.3 (Existence of the least fixed points)**.** *Let $F_c(\sigma_c) \triangleq \sigma_c^0 \sqcup_c \mathcal{E}_c[\![s]\!](\sigma_c)$ and $F_d^{\sigma_c}(\sigma_d) \triangleq \sigma_d^0 \sqcup_d \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Then $\mathrm{lfp}(F_c)$ exists, and for every σ_c, $\mathrm{lfp}(F_d^{\sigma_c})$ exists.*

Proof. By Lemma B.2, $\mathcal{E}_c[\![s]\!]$ and $\mathcal{E}_d[\![s]\!]$ are monotone; joins with fixed stores σ_c^0, σ_d^0 preserve monotonicity, so F_c and $F_d^{\sigma_c}$ are monotone. The environments Σ_c and Σ_d are complete lattices (flat lifts with $\perp, \top$ are complete; finite products over complete lattices are complete under pointwise order). By the Tarski-Knaster theorem, $\mathrm{lfp}(F_c)$ and, for each σ_c, $\mathrm{lfp}(F_d^{\sigma_c})$ exist. □

B.2 Preliminaries: Cylindrical Algebraic Decomposition

Cylindrical algebraic decomposition (CAD) decomposes $\mathbb{R}^n$ into finitely many semi-algebraic cells where each input polynomial has constant sign [65,66]. The algorithm takes as input

a tuple of ordered variables (which we term "vars") $(y_1, \dots, y_n)$ and a set of polynomials in those variables ("cuts") $F = \{f_1, \dots, f_m\} \subset \mathbb{Q}[y_1, \dots, y_n]$, and returns a CAD tree structure ("tree") that encodes the decomposition.

Definition B.4 (Cylindrical Algebraic Decomposition)**.** Let $F \subset \mathbb{Q}[y_1, \dots, y_n]$ be a finite set of polynomials with variable ordering $y_1 \prec \cdots \prec y_n$.

Projection. Set $P_n := F$. For $k = n, \dots, 2$ define

$$P_{k-1} \;=\; \Big\{ \operatorname{coeff}_{y_k}(p),\, \operatorname{disc}_{y_k}(p),\, \operatorname{res}_{y_k}(p, q) \;\Big|\; p, q \in P_k,\; p \neq q \Big\} \subset \mathbb{Q}[y_1, \dots, y_{k-1}],$$

where $\operatorname{coeff}_{y_k}(p)$ denotes every coefficient of p viewed as a univariate in y_k, $\operatorname{disc}_{y_k}(p)$ is the discriminant of p with respect to y_k, and $\operatorname{res}_{y_k}(p, q)$ is the resultant of p and q with respect to y_k.

Lifting.

1. (*Base*) Isolate the ordered real roots $r_1 \prec \cdots \prec r_s$ of the polynomials in P_1 and form $\mathcal{D}_1 = \{(-\infty, r_1), \{r_1\}, (r_1, r_2), \dots, \{r_s\}, (r_s, \infty)\}$.

2. (*Induction*) For $k = 2, \dots, n$: for each $C \in \mathcal{D}_{k-1}$, choose a sample point $s_C \in C$ and substitute it into all $p \in P_k$. Let $\alpha_1 \prec \cdots \prec \alpha_m$ be the real roots of the resulting polynomials in y_k. Partition $C \times \mathbb{R}$ into:

   - *sections*: $C \times \{\alpha_i\}$ for $i = 1, \dots, m$
   - *sectors*: $C \times (\alpha_{i-1}, \alpha_i)$ for $i = 0, \dots, m + 1$

   where $\alpha_0 = -\infty$ and $\alpha_{m+1} = \infty$. $\mathcal{D}_k$ is the collection of all these sections and sectors.

Output. $\mathrm{CAD}(F) := \mathcal{D}_n$, a finite partition of $\mathbb{R}^n$ into connected semi-algebraic cells that is *cylindrical* (the projection of any cell onto the first k coordinates is a union of cells of $\mathcal{D}_k$ for every $k < n$) and *sign-invariant* for F (each $p \in F$ has constant sign on each cell).

The algorithm first computes the projection sets $P_{n-1}, \dots, P_1$ by successively eliminating the variables $y_n, \dots, y_2$; each P_{k-1} collects all coefficients, discriminants, and pairwise resultants required to mark where the original polynomials can change sign in the lower-dimensional space. In the lifting phase, space is rebuilt coordinate by coordinate. Beginning with the line decomposition $\mathcal{D}_1$ obtained from the real roots of P_1, the algorithm iteratively refines: for every cell $C \in \mathcal{D}_{k-1}$ a sample point s_C is chosen, substituted into the polynomials of P_k, and the resulting real roots $\alpha_1 \prec \cdots \prec \alpha_m$ in y_k are isolated. These roots define *sections* $C \times \{\alpha_i\}$ and the open intervals between them called *sectors* $C \times (\alpha_{i-1}, \alpha_i)$, yielding $\mathcal{D}_k$. Repeating for $k = 2, \dots, n$ produces the cylindrical family $\mathcal{D}_n$; its cells and their sample points form a rooted tree whose edges record how each higher-dimensional cell projects onto its parent, fully encoding the cylindrical structure.

Numerous projection operators have been introduced to reduce the size of the projection sets, such as McCallum's reduced projection [157] and Brown–McCallum's minimal projection [158]. We use Hong's projector [159] in Prexo, which guarantees correctness without additional well-orientedness conditions while often yielding smaller sets than Collins's original projector.

B.3 Soundness of α and γ

Theorem B.5 (Soundness of α and γ)**.** *For every set of concrete states $S \subseteq \Sigma$, $S \subseteq \gamma(\alpha(S))$. Equivalently, for each $\sigma \in S$ we have $\sigma \models \mathcal{L}^{\#}(\alpha(S))$.*

Proof. Let $S \subseteq \Sigma$ and fix $\sigma = (\sigma_c, \sigma_d) \in S$. Because $\mathcal{L}^{\#}$ conjoins the per-variable formulas (Def. 6.8), it suffices to prove, for each variable $r \in \mathbb{X} \cup \mathbb{Y}$, that

$$\sigma \models \mathcal{L}^{\#}_{\mathcal{T}\langle\pi\rangle}(r, \hat{\alpha}(\sigma)(r)),$$

where $\pi = \zeta$ for control variables and $\pi = \delta$ for data variables. The theorem will then follow by conjoining these per-variable facts and lifting from $\hat{\alpha}(\sigma)$ to $\alpha(S)$ via join-weakening.

Control $y \in \mathbb{Y}$. By the definition of α_c (Def. 6.7),

$$\alpha_c(\sigma_c)(y) \;=\; \bigsqcup_{\vec{i} \in \mathbb{Z}^{\mathrm{itr}(y)}} \Big[\, \vec{\iota} \;\Big|\; \bigwedge_r (\iota_r = i_r) \;\Rightarrow\; \Big\lceil \sigma_c(y)(\vec{i}) \Big\rceil_C \,\Big],$$

where $\vec{\iota}$ is the tuple of control variables and $\vec{i}$ ranges over integer tuples of the same arity. Applying the lifting $\mathcal{L}^{\#}$ (Fig. 6.9), a top-level $\mathsf{LinSplit}$ yields

$$\mathcal{L}^{\#}_{\mathcal{T}\langle\zeta\rangle}(y, \alpha_c(\sigma_c)(y)) \;=\; \bigvee_{\vec{i}} \Big(\mathcal{L}^{\#}_g(\textstyle\bigwedge_r(\iota_r = i_r)) \;\wedge\; \mathcal{L}^{\#}_\zeta(y, \Big\lceil \sigma_c(y)(\vec{i}) \Big\rceil_C) \Big).$$

Let $\vec{i}^{\sigma}$ be the concrete integer tuple chosen by σ (Def. 6.3). Then the guard $\bigwedge_r(\iota_r = i^{\sigma}_r)$ is true under that valuation, whereas all guards for $\vec{i} \neq \vec{i}^{\sigma}$ are false. Hence exactly one disjunct remains relevant; its leaf payload is $\Big\lceil \sigma_c(y)(\vec{i}^{\sigma}) \Big\rceil_C$, and the leaf clause of $\mathcal{L}^{\#}$ yields a base constraint (e.g. $y = \sigma_c(y)(\vec{i}^{\sigma})$) that holds in σ by construction. Therefore $\sigma \models \mathcal{L}^{\#}_{\mathcal{T}\langle\zeta\rangle}(y, \alpha_c(\sigma_c)(y))$.

Data $x \in \mathbb{X}$. If $x \notin \mathrm{Lhs}_p$ (a free variable), α_d produces a single *unguarded* leaf with payload $\lceil x[\vec{\eta}] \rceil_D$; the corresponding base clause in the lifting is tautological (e.g. $x[\vec{j}] = x[\vec{j}]$), so it holds in every σ. Otherwise,

$$\alpha_d(\sigma_c, \sigma_d)(x) \;=\; \bigsqcup_{\vec{i},\vec{j}} \Big[\, \vec{\iota}\,\vec{\eta} \;\Big|\; (\textstyle\bigwedge_r \iota_r = i_r) \wedge (\bigwedge_s \eta_s = j_s) \;\Rightarrow\; \Big\lceil \sigma_d(x)(\vec{i}, \vec{j}) \Big\rceil_D \,\Big].$$

Let $(\vec{i}^{\sigma}, \vec{j}^{\sigma})$ be the concrete indices selected by σ. As above, only the disjunct with that guarded pair can be true, and at its leaf the payload is exactly $\Big\lceil \sigma_d(x)(\vec{i}^{\sigma}, \vec{j}^{\sigma}) \Big\rceil_D$, so the resulting atomic constraint holds in σ.

Combining the two cases, we obtain $\sigma \models \mathcal{L}^{\#}(\hat{\alpha}(\sigma))$.

□

B.4 Soundness of $S^{\#}$

Theorem B.6 (Soundness of $S^{\#}$)**.** *For every statement s, concrete stores $(\sigma_c, \sigma_d) \in \Sigma$ and abstract store $\sigma^{\#} \in \Sigma^{\#}$,*

$$\hat{\alpha}(\sigma_c, \sigma_d) \;\sqsubseteq_{\Sigma^{\#}}\; \sigma^{\#} \;\Longrightarrow\; \hat{\alpha}(\,\mathcal{E}_c[\![s]\!](\sigma_c),\; \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)\,) \;\sqsubseteq_{\Sigma^{\#}}\; S^{\#}[\![s]\!]\,\sigma^{\#}.$$

Proof. We proceed by structural induction on s. We rely on the following standard properties (all hold for the given definitions; proofs are routine inductions and omitted for brevity).

(F1) *Pointwise soundness of $C^\#$:* If $\hat{\alpha}(\sigma_c, \sigma_d) \sqsubseteq \sigma^\#$ and c is a control expression, then for $\imath^* = \mathbb{FV}(c)$ and for every integer tuple $z^* \in \mathbb{Z}$,

$$\lceil \mathcal{E}_c^e[\![c]\!](\sigma_c[\imath^* \mapsto z^*]) \rceil_C \;\sqsubseteq_{\mathcal{B}_C}\; (C^\#[\![c]\!]\,\sigma^\#[\imath^* \mapsto z^*])\langle\rangle.$$

When all control variables in the control expression maps to a concrete integer ($\imath^* \mapsto z^*$), all the branch in $C^\#[\![c]\!]\,\sigma^\#$ should be either trivially false or true, thus leading to only one leaf value. Therefore, applying an empty tuple $\langle\rangle$ yields one and only reachable base domain value. The proof is by induction on c, using the given clauses for $C^\#$ and $A^\#$.

(F2) *Componentwise update for $\hat{\alpha}$:* If $\hat{\alpha}(\sigma_c, \sigma_d) \sqsubseteq \sigma^\#$ and for some y we have a function $f : \mathbb{Z}^{\mathrm{itr}(y)} \to \mathbb{Z}_\perp^\top$ and a tree $\tau \in \mathcal{T}_C$ such that

$$\forall z^*.\; \lceil f(z^*) \rceil_C \;\sqsubseteq_{\mathcal{B}_C}\; \tau(z^*),$$

then $\hat{\alpha}(\sigma_c[y \mapsto f], \sigma_d) \;\sqsubseteq_{\Sigma^\#}\; \sigma^\#[y \mapsto \tau]$.

(Sequence) $s \equiv s_1; s_2$. From the IH for s_1 and the premise $\hat{\alpha}(\sigma_c, \sigma_d) \sqsubseteq \sigma^\#$ we get

$$\hat{\alpha}(\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c, \sigma_d)) \;\sqsubseteq\; S^\#[\![s_1]\!]\,\sigma^\#.$$

Applying the IH to s_2 with concrete input $(\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c, \sigma_d))$ and abstract input $S^\#[\![s_1]\!]\,\sigma^\#$ yields

$$\hat{\alpha}\,(\mathcal{E}_c[\![s_2]\!](\mathcal{E}_c[\![s_1]\!](\sigma_c)),\; \mathcal{E}_d[\![s_2]\!](\mathcal{E}_c[\![s_1]\!](\sigma_c),\; \mathcal{E}_d[\![s_1]\!](\sigma_c, \sigma_d))) \;\sqsubseteq\; S^\#[\![s_2]\!](S^\#[\![s_1]\!]\,\sigma^\#).$$

This is the desired inequality since both concrete and abstract semantics compose.

(Control-assign) $s \equiv y = \lambda\imath^*.(b?c_t : c_f)$. Let

$$\tau_t = C^\#[\![c_t]\!]\sigma^\#, \quad \tau_f = C^\#[\![c_f]\!]\sigma^\#, \quad \Pi = B^\#[\![b]\!]\sigma^\#.$$

Build the CAD skeleton from the branches (cf. Figure 6.11):

$$\tau_{\mathrm{skel}} = \mathrm{CAD}_\perp^\top(\mathrm{vars}, \mathrm{cuts}), \quad \mathrm{vars} = \tau_t.\mathrm{vars} \;\uplus\; \tau_f.\mathrm{vars} \;\uplus\; \imath^*, \quad \mathrm{cuts} = \tau_t.\mathrm{cuts} \;\uplus\; \tau_f.\mathrm{cuts} \;\uplus\; \Pi.$$

If any component is imprecise (unknown iterator or a $\top_\zeta/\perp_\zeta$ cut), then by definition $\mathrm{CAD}_\perp^\top$ returns $\top_{\mathcal{T}_C}$; since $\top_{\mathcal{T}_C}$ is greatest, the target inequality holds trivially.

Henceforth assume the precise case. By the concrete semantics (Figure 6.6),

$$\mathcal{E}_c[\![s]\!](\sigma_c) = \sigma_c[y \mapsto a_y], \quad a_y(z^*) = \mathrm{ite}(\,\mathcal{E}_b[\![b]\!](\sigma_c[\imath^* \mapsto z^*]),\; \mathcal{E}_c^e[\![c_t]\!](\sigma_c[\imath^* \mapsto z^*]),\; \mathcal{E}_c^e[\![c_f]\!](\sigma_c[\imath^* \mapsto z^*])\,).$$

The abstract transformer constructs $\tau_{\mathrm{new}}^y = \tau_{\mathrm{skel}}$ and then performs the leaf substitution

$$\tau_{\mathrm{new}}^{y\,\prime} = \tau_{\mathrm{new}}^y[\mathrm{Leaf}(\_, \hat{n}) \mapsto \mathrm{Leaf}(v(\hat{n}), \hat{n})], \quad v(\hat{n}) = \begin{cases} \tau_t(\hat{n}) & \text{if } \mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \texttt{true}, \\ \tau_f(\hat{n}) & \text{otherwise.} \end{cases}$$

By (F2), it suffices to show the *pointwise* inequality

$$\forall z^*.\ \lceil a_y(z^*)\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ v(z^*). \tag{B.7}$$

Case $\mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \texttt{true}$. Then $a_y(z^*) = \mathcal{E}_c^e[\![c_t]\!](\{\imath^* \mapsto \hat{n}\})$. By (F1) with $c = c_t$,

$$\lceil a_y(z^*)\rceil_C = \lceil \mathcal{E}_c^e[\![c_t]\!](\{\imath^* \mapsto \hat{n}\})\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ \tau_t(z^*) \ = \ v(z^*).$$

Case $\mathcal{E}_b[\![b]\!](\{\imath^* \mapsto \hat{n}\}) = \texttt{false}$. Symmetrically, $a_y(z^*) = \mathcal{E}_c^e[\![c_f]\!](\{\imath^* \mapsto \hat{n}\})$ and (F1) with $c = c_f$ gives $\lceil a_y(z^*)\rceil_C \ \sqsubseteq_{\mathcal{B}_C}\ \tau_f(z^*) = v(z^*)$.

Thus (B.7) holds for all z^*. Applying (F2) yields

$$\hat{\alpha}(\sigma_c[y \mapsto a_y], \sigma_d) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#[y \mapsto \tau^{y}_{\text{new}}{}'] \ = \ S^\#[\![s]\!]\,\sigma^\#.$$

Finally, by the (implicit) definition of $\mathcal{E}_d$ for control-assign, the data store is unchanged: $\mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) = \sigma_d$, and the claim follows.

Proof for the data assignment statement is similar. □

B.5 Soundness of $P^\#$

Lemma B.8 (Fixpoint transfer)**.** *$(\Sigma, \sqsubseteq_\Sigma)$ and $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$ are complete lattices. $\overline{F} : \Sigma \to \Sigma$ and $F^\# : \Sigma^\# \to \Sigma^\#$ are monotone, and $\hat{\alpha} : \Sigma \to \Sigma^\#$ preserves arbitrary joins. If*

$$\hat{\alpha} \circ \overline{F} \ \sqsubseteq_{\Sigma^\#}\ F^\# \circ \hat{\alpha},$$

then

$$\hat{\alpha}(\text{lfp}(\overline{F})) \ \sqsubseteq_{\Sigma^\#}\ \text{lfp}(F^\#).$$

Proof. For each $\sigma^\# \in \Sigma^\#$, define the candidate set

$$\Sigma_{\sigma^\#} \ \triangleq\ \{\ \sigma \in \Sigma \mid \hat{\alpha}(\sigma) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#\ \},$$

and its join

$$\sigma_{\text{join}} \ \triangleq\ \bigsqcup_{\Sigma}\{\ \sigma \in \Sigma \mid \hat{\alpha}(\sigma) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#\ \}.$$

Note that $\Sigma_{\sigma^\#} \neq \varnothing$ because $\hat{\alpha}$ preserves the empty join, hence $\hat{\alpha}(\bot_\Sigma) = \bot_{\Sigma^\#} \sqsubseteq_{\Sigma^\#} \sigma^\#$ and thus $\bot_\Sigma \in \Sigma_{\sigma^\#}$. Since $\hat{\alpha}$ preserves arbitrary joins,

$$\hat{\alpha}(\sigma_{\text{join}}) \ = \ \hat{\alpha}\left(\bigsqcup_{\Sigma}\{\sigma \in \Sigma \mid \hat{\alpha}(\sigma) \sqsubseteq_{\Sigma^\#} \sigma^\#\}\right) \ = \ \bigsqcup_{\Sigma^\#}\{\hat{\alpha}(\sigma) \mid \hat{\alpha}(\sigma) \sqsubseteq_{\Sigma^\#} \sigma^\#\} \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#,$$

so σ_{join} is the greatest element of Σ whose $\hat{\alpha}$-image lies below $\sigma^\#$.

Now fix any *pre-fixed point* $\sigma^\#$ of $F^\#$, i.e. $F^\#(\sigma^\#) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#$. Using the assumption $\hat{\alpha} \circ \overline{F} \sqsubseteq_{\Sigma^\#} F^\# \circ \hat{\alpha}$ and monotonicity of $F^\#$, we have

$$\hat{\alpha}\left(\overline{F}\left(\sigma_{\text{join}}\right)\right) \ \sqsubseteq_{\Sigma^\#}\ F^\#\left(\hat{\alpha}\left(\sigma_{\text{join}}\right)\right) \ \sqsubseteq_{\Sigma^\#}\ F^\#(\sigma^\#) \ \sqsubseteq_{\Sigma^\#}\ \sigma^\#.$$

Hence $\overline{F}\,(\sigma_{\text{join}}) \in \Sigma_{\sigma^\#}$, which implies

$$\overline{F}\,(\sigma_{\text{join}}) \;\sqsubseteq_\Sigma\; \sigma_{\text{join}}.$$

Thus σ_{join} is a pre-fixed point of $\overline{F}$, and by Park's induction (least pre-fixed–point rule) [160]

$$\text{lfp}(\overline{F}) \;\sqsubseteq_\Sigma\; \sigma_{\text{join}}.$$

Applying monotonicity of $\hat{\alpha}$ gives

$$\hat{\alpha}(\text{lfp}(\overline{F})) \;\sqsubseteq_{\Sigma^\#}\; \hat{\alpha}\,(\sigma_{\text{join}}) \;\sqsubseteq_{\Sigma^\#}\; \sigma^\#.$$

Since this holds for every pre-fixed point $\sigma^\#$ of $F^\#$, we conclude

$$\hat{\alpha}(\text{lfp}(\overline{F})) \;\sqsubseteq_{\Sigma^\#}\; \bigsqcap_{\Sigma^\#}\{\, \sigma^\# \in \Sigma^\# \mid F^\#(\sigma^\#) \;\sqsubseteq_{\Sigma^\#}\; \sigma^\# \,\} \;=\; \text{lfp}(F^\#).$$

□

Theorem B.9 (Soundness of $P^\#$)**.** *Let $p \equiv \text{Fix}\, s$. Let initial concrete stores $\sigma_c^0 \in \Sigma_c$, $\sigma_d^0 \in \Sigma_d$ and an initial abstract store $\sigma^{\#0} \in \Sigma^\#$ satisfy $\hat{\alpha}(\sigma_c^0, \sigma_d^0) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#0}$. Write the concrete program semantics from Figure 6.6 as $\langle \sigma_c^*, \sigma_d^* \rangle \;=\; \langle\, \text{lfp}(F_c)\;,\; \text{lfp}(F_d^{\,\text{lfp}(F_c)}) \,\rangle$, where $F_c(\sigma_c) = \sigma_c^0 \;\sqcup_c\; \mathcal{E}_c[\![s]\!](\sigma_c)$ and $F_d^{\sigma_c}(\sigma_d) = \sigma_d^0 \;\sqcup_d\; \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d)$. Let the abstract program semantics be $\sigma^{\#*} = \text{lfp}(F^\#)$ with $F^\#(\sigma^\#) \triangleq \sigma^{\#0} \;\sqcup_{\Sigma^\#}\; S^\#[\![s]\!](\sigma^\#)$ (Def. 6.10). Then*

$$\hat{\alpha}(\sigma_c^*, \sigma_d^*) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#*}.$$

Proof. Define the *simultaneous concrete operator* $\overline{F} : \Sigma \to \Sigma$ by

$$\overline{F}(\langle \sigma_c, \sigma_d \rangle) \;\triangleq\; \Big\langle\; \sigma_c^0 \;\sqcup_c\; \mathcal{E}_c[\![s]\!](\sigma_c)\;,\; \sigma_d^0 \;\sqcup_d\; \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \;\Big\rangle.$$

For any $\langle \sigma_c, \sigma_d \rangle \in \Sigma$,

$$\begin{aligned}
\hat{\alpha}(\overline{F}\langle \sigma_c, \sigma_d \rangle) &= \hat{\alpha}\Big(\; \langle \sigma_c^0, \sigma_d^0 \rangle \;\sqcup_\Sigma\; \langle \mathcal{E}_c[\![s]\!](\sigma_c),\; \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \rangle \;\Big) \\
&= \hat{\alpha}\langle \sigma_c^0, \sigma_d^0 \rangle \;\sqcup_{\Sigma^\#}\; \hat{\alpha}\Big\langle \mathcal{E}_c[\![s]\!](\sigma_c),\; \mathcal{E}_d[\![s]\!](\sigma_c, \sigma_d) \Big\rangle \quad (\hat{\alpha} \text{ is pointwise and join-preserving}) \\
&\sqsubseteq_{\Sigma^\#}\; \sigma^{\#0} \;\sqcup_{\Sigma^\#}\; S^\#[\![s]\!](\hat{\alpha}\langle \sigma_c, \sigma_d \rangle) \quad (\text{init. assumption and Thm. B.6}) \\
&= F^\#(\hat{\alpha}\langle \sigma_c, \sigma_d \rangle).
\end{aligned}$$

Hence $\hat{\alpha} \circ \overline{F} \;\sqsubseteq\; F^\# \circ \hat{\alpha}$. By Lemma B.8, $\hat{\alpha}(\text{lfp}(\overline{F})) \;\sqsubseteq_{\Sigma^\#}\; \text{lfp}(F^\#) \;=\; \sigma^{\#*}$. From Bekic's theorem [161], the least fixpoint of $\overline{F}$ is

$$\text{lfp}(\overline{F}) \;=\; \Big\langle\; \text{lfp}(F_c)\;,\; \text{lfp}(F_d^{\,\text{lfp}(F_c)}) \;\Big\rangle,$$

which is exactly $\langle \sigma_c^*, \sigma_d^* \rangle$ from Figure 6.6. This gives $\hat{\alpha}(\sigma_c^*, \sigma_d^*) \;\sqsubseteq_{\Sigma^\#}\; \sigma^{\#*}$, as required. □

B.6 Soundness and Stabilization of ∇

Lemma B.10 ($\mathsf{Floodfill}_n$ is inflationary and idempotent w.r.t. its first argument)**.** *Fix $\pi \in \{\zeta, \delta\}$ and $c \in \mathcal{B}_\pi$. For every node n in $\mathcal{T}\langle\pi\rangle$, let $\mathsf{Floodfill}_n(n, c) = (m, c')$. Then:*

1. *Inflationary: $n \sqsubseteq_{\mathcal{T}\langle\pi\rangle} m$.*
2. *Idempotent (at fixed carry): $\mathsf{Floodfill}_n(m, c) = (m, c')$.*

Proof. By structural induction on n.

Leaf. If $n = \mathrm{Leaf}(\ell, \mathbf{s})$ then $m = \mathrm{Leaf}(\ell', \mathbf{s})$ with $\ell' = \ell$ when $\ell \neq \bot_\pi$ and $\ell' = c$ when $\ell = \bot_\pi$. In the flat base lattice, $\ell \sqsubseteq_\pi \ell'$; thus inflationarity holds. Reapplying $\mathsf{Floodfill}_n$ with the same c does nothing (if $\ell' = \bot_\pi$ it stays $\bot_\pi$; otherwise it stays ℓ'), so idempotence holds.

Split. If $n = \mathrm{LinSplit}((g_i, n_i)_{i=1}^k)$, write $c_0 = c$ and $(m_i, c_i) = \mathsf{Floodfill}_n(n_i, c_{i-1})$ for $i = 1, \ldots, k$; then $m = \mathrm{LinSplit}(\mathrm{Cell}(g_1, m_1), \ldots, \mathrm{Cell}(g_k, m_k))$. By the IH applied to each child, $n_i \sqsubseteq_{\mathcal{T}\langle\pi\rangle} m_i$, hence (by the partial order of Def. 6.3) $n \sqsubseteq_{\mathcal{T}\langle\pi\rangle} m$. For idempotence, apply the IH again: $\mathsf{Floodfill}_n(m_i, c_{i-1}) = (m_i, c_i)$ for each i, so the second pass reconstructs exactly the same children and carries, yielding (m, c_k). □

Corollary B.11 ($\mathsf{Floodfill}$ is inflationary and idempotent)**.** $\mathsf{Floodfill}_t$ *is inflationary and idempotent, and* $\mathsf{Floodfill} : \Sigma^\# \to \Sigma^\#$ *(Def. 6.12) is inflationary and idempotent.*

Theorem B.12 (Soundness of Widening)**.** *For all $X, Y \in \Sigma^\#$,*

$$(X \sqcup_{\Sigma^\#} Y) \sqsubseteq_{\Sigma^\#} X \nabla Y = \mathsf{Floodfill}(X \sqcup_{\Sigma^\#} Y).$$

Proof. By Cor. B.11, $\mathsf{Floodfill}$ is inflationary; thus $(X \sqcup_{\Sigma^\#} Y) \sqsubseteq_{\Sigma^\#} \mathsf{Floodfill}(X \sqcup_{\Sigma^\#} Y)$. □

Lemma B.13 (Monotonicity of the widening iteration)**.** *For any initial $\widetilde{\sigma}_0^\# \in \Sigma^\#$, define $\widetilde{\sigma}_{i+1}^\# \triangleq \mathsf{Floodfill}(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma}_i^\#))$. Then $\widetilde{\sigma}_i^\# \sqsubseteq_{\Sigma^\#} \widetilde{\sigma}_{i+1}^\#$ for all i.*

Proof. By Cor. B.11, $\mathsf{Floodfill}$ is inflationary. Hence $\widetilde{\sigma}_i^\# \sqsubseteq_{\Sigma^\#} \mathsf{Floodfill}(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma} i^\#)) = \widetilde{\sigma}_{i+1}^\#$. □

Lemma B.14 ($\Sigma^\#$ is finite height)**.** $\Sigma^\#$ *is finite height.*

Proof. By construction, the base domains $\mathcal{B}_\pi$ ($\pi \in \{\zeta, \delta\}$) are flat lattices and hence finite height. The total number of cells in any CAD is bounded by a finite constant $M \leq t^{2^{O(n)}}$, where n is the size of vars and t is the size of cuts. Since only a finite number of program variables and cuts can exist for a given program p, both $\mathcal{T}_C$ and $\mathcal{T}_D$ are finite-height lattices. By Definition 6.4, $\Sigma^\# \triangleq \Sigma_c^\# \times \Sigma_d^\#$ with $\Sigma_c^\# \triangleq (\mathbb{Y} \to \mathcal{T}_C)$ and $\Sigma_d^\# \triangleq (\mathbb{X} \to \mathcal{T}_D)$. Since there are only finite number of variables in a program and $\mathcal{T}_C$ and $\mathcal{T}_D$ are finite-height lattices, $\Sigma^\#$ is a finite height lattice. □

Theorem B.15 (Stabilization of ∇)**.** *For any initial $\widetilde{\sigma}_0^\# \in \Sigma^\#$, define $\widetilde{\sigma}_{i+1}^\# \triangleq \mathsf{Floodfill}(\widetilde{\sigma}_i^\# \sqcup_{\Sigma^\#} F^\#(\widetilde{\sigma}_i^\#))$. Then $(\widetilde{\sigma}_i^\#)_{i \geq 0}$ stabilizes in finitely many steps.*

Proof. $F^\#$ is monotone by Definition 6.10, and the widening iteration is monotone by Lemma B.13. Therefore, $(\widetilde{\sigma}_i^\#)_{i \geq 0}$ is an ascending chain in $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$. By Lemma B.14, $(\Sigma^\#, \sqsubseteq_{\Sigma^\#})$ has finite height. Every ascending chain in a finite-height poset stabilizes after finitely many steps; hence so does $(\widetilde{\sigma}_i^\#)_{i \geq 0}$. □

Appendix C

Exo-GPU for Tensor Cores

C.1 GPU IR Type Definitions

τ_x : DataType ::=	`f32`	32-bit floating point
	| `f64`	64-bit floating point
	| `f16`	16-bit floating point
	| `i32`	32-bit integer
	| `ui16`	16-bit unsigned integer
	| `i8`	8-bit integer
	| `ui8`	8-bit unsigned integer
τ_z : BarrierType ::=	`barrier`	non-explicitly guarded barrier
	| `barrier(z)`	explicitly guarded barrier

Figure C.1: List of variable types.

	domain	*box*
τ_u : CollUnit ::= `standalone_thread`	`(1,)`	`(1,)`
| n_1 `* cuda_thread`	`(blockDim,)`	(n_1`,)`
| `cuda_quadpair`	`(blockDim/16, 16)`	`(2, 4)`
| n_1 `* cuda_warp`	`(blockDim,)`	(n_1 `* 32,)`
| n_1 `* cuda_warpgroup`	`(blockDim,)`	(n_1 `* 128,)`
| n_1 `* cuda_threads_strided(`n_2`, `n_3`)`	`(blockDim/`n_3`, `n_3`)`	(n_1`, `n_2`)`
| n_1 `* cuda_warp_in_cluster`	`(clusterDim, blockDim)`	(n_1`, 32)`
| n_1 `* cuda_cta_in_cluster`	`(clusterDim * blockDim,)`	(n_1 `* blockDim,)`
| `cuda_cluster`	`(clusterDim * blockDim,)`	`(clusterDim * blockDim,)`
| n_1 `* cuda_cta_in_cluster_strided(`n_3`)`	`(clusterDim/`n_3`, `n_3`, blockDim)`	(n_1`, 1, blockDim)`
| n_1 `* cuda_warp_in_cluster_strided(`n_3`)`	`(clusterDim/`n_3`, `n_3`, blockDim)`	(n_1`, 1, 32)`
| `cuda_agnostic_sub_cta`	`(clusterDim, blockDim)`	`(1, _)`
| `cuda_agnostic_intact_cta`	`(clusterDim, blockDim)`	`(_, blockDim)`

Figure C.2: Collective units defined in the frontend language. These are parameterized by a collective type δ, which consists of the *domain* and *box* shown here.

π_d : DataMemory	::= `CudaBasicDeviceVisible`	base type for CUDA-visible allocations
	| `CudaBasicSmem`	base type for CUDA SMEM allocations
	| `CudaDeviceVisibleAtomicity16B`	base type for 16B-atomicity layout CUDA-visible allocations
	| `CudaDeviceVisibleLinear`	base type for linear-order CUDA-visible allocations
	| `CudaGridConstant`	CUDA grid constant memory
	| `CudaGmemLinear`	CUDA global memory, linear-order
	| `CudaSmemAtomicity16B`	any CUDA shared memory with 16B-atomicity layout
	| `CudaSmemLinear`	CUDA shared memory with linear-order layout
	| `CudaRmem`	per-thread CUDA registers
	| `Sm80_RmemMatrixA(M, K)`	matrix tile for sm_80+ warp MMA A operand
	| `Sm80_RmemMatrixB(N, K)`	matrix tile for sm_80+ warp MMA B operand
	| `Sm80_RmemMatrixD(M, N)`	matrix tile for sm_80+ MMA acumulator operands
	| `Sm90_SmemSwizzled(B)`	B-byte swizzled shared memory, in TMA/wgmma-accepted format
	| `Sm90_RmemMatrixA(M)`	register tile for wgmma A operand (not implemented)
	| `Sm90_RmemMatrixD(M, N)`	register tile for wgmma D operand

Figure C.3: CUDA Memory Types – Data Allocations (pre-existing Exo Memory not listed).

π_z : BarrierComp	::= `CudaDeviceBarrier`	base type for CUDA-allocated barriers
	| `CudaMbarrier`	CUDA `mbarrier`
	| `CudaCommitGroup`	CUDA `commit_group` mechanism
	| `CudaClusterSync`	`barrier.cluster` sync

Figure C.4: CUDA Barrier (completion) mechanisms.

π_w : SpecialWindow	::= `Sm90_tensorMap(swizzle, *smem_box)` `CUtensorMap` parameterized by swizzle mode and box size

Figure C.5: CUDA Special Window Types (for Window Statement).

τ_q : QualTL	::= `cpu_in_order_qual`	ordinary CPU reads/writes (cpu)
	| `cpu_cuda_stream_qual`	stream-ordered CUDA API calls (strm)
	| `cuda_in_order_rmem_qual`	non-async CUDA instrs' accesses to registers (cuda1)
	| `cuda_in_order_ram_qual`	non-async CUDA instrs' accesses to SMEM/GMEM (cuda2)
	| `Sm80_cp_async_qual`	non-bulk `cp.async` memory accesses (Sm80)
	| `tma_to_smem_async_qual`	`cp.async.bulk` GMEM reads, SMEM writes (tmaS)
	| `tma_to_gmem_async_qual`	`cp.async.bulk` SMEM reads, GMEM writes/reduces (tmaG)
	| `wgmma_async_rmem_a_qual`	`wgmma.mma_async` reads from A register parameter (wgA)
	| `wgmma_async_rmem_d_qual`	`wgmma.mma_async` accesses to D (accumulator) (wgD)
	| `wgmma_async_smem_qual`	`wgmma.mma_async` access to SMEM, A or B (wgS)
	| `wgmma_zero_qual`	special case used to model the `wgmma` scale-d parameter (wg0)

Figure C.6: List of qualitative timelines.

τ_s : SyncTL	trnstv?	cpu	strm	cuda1	cuda2	Sm80	tmaS	tmaG	wgA	wgD	wgS	wg0
empty_sync_tl												
cpu_in_order	y	full										
cuda_stream_sync	y		full	full	full	full	full	full	full	full	full	
cuda_in_order	y		temp.	full	full	temp.	temp.	temp.	temp.	temp.	temp.	temp.
cuda_temporal			temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.	temp.
Sm80_cp_async						full						
Sm80_generic			temp.	full	full	full	temp.	temp.	temp.	temp.	temp.	temp.
tma_to_smem_async							full					
tma_to_gmem_async								full				
wgmma_async_smem											full	
wgmma_fence_1				full					full	full		
wgmma_fence_2									full	full		
wgmma_async									full	full	full	
cuda_async_proxy							full	full			full	
cuda_async_proxy_wgmma							full	full	full	full	full	
cuda_generic_and_async			temp.	full	full	full	full	full	temp.	temp.	full	temp.

Figure C.7: SyncTL are defined as composition of QualTL.

C.2 Abstract Machine Conversion and Semantics Definitions

A new record Constructor $\mathcal{R}$ is used to define `RecordRead` and `RecordMutate` in Figure C.13. RecordRead and RecordMutate first evaluate the accessor $e^{\#}$ and $e_z^{\#}$ to access sets $W_{e^{\#}}$ and $W_{e_z^{\#}}$ (via $\Downarrow_{\mathsf{Acc}}$). They then update the visibility map ρ at each key $(x, \mathbf{n}) \in W_{e^{\#}}$ using the new record constructor $\mathcal{R}$ defined below: RecordRead adds it to the *read* visibility set, setting $\rho'(x, \mathbf{n}).r = \rho(x, \mathbf{n}).r \cup \mathcal{R}(\cdots)$ while leaving $\rho(x, \mathbf{n}).m$ and $\rho(x, \mathbf{n}).b$ unchanged; RecordMutate adds it to the *mutate* visibility set, setting $\rho'(x, \mathbf{n}).m = \rho(x, \mathbf{n}).m \cup \mathcal{R}(\cdots)$ while preserving $\rho(x, \mathbf{n}).r$ and $\rho(x, \mathbf{n}).b$. In both cases, neither the control $\imath$ nor the store σ changes.

Definition C.1 (New Record Constructor $\mathcal{R}$)**.** Define $\mathcal{R}$ as follows:

$$\mathcal{R} : \ \mathrm{VL} \times \mathrm{Bool} \times \mathrm{QualTL} \times \mathcal{P}(\mathrm{QualTL}) \times \mathsf{Expr}^{\#} \times \rho \times \mathbb{G} \ \longrightarrow \ \mathcal{P}(\mathbb{R})$$

$$\mathcal{R}(\tau_v, t, \tau_q, \tau_q^*, E^{\#}, \rho, G) \ = \ \begin{cases} \{\, (\tau_q,\ S_G,\ P_{E^{\#},\rho}) \,\}, & t = \mathsf{true}, \\ \{\, (\tau_q,\ S_{\{g\}},\ P_{E^{\#},\rho}) \mid g \in G \,\}, & t = \mathsf{false}, \end{cases}$$

where, for any $H \subseteq \mathbb{G}$, $S_H \in \mathbb{S}$ and $P_{E^{\#},\rho} \in \mathbb{P}$ are given pointwise by

$$S_H(g', \tau_q') \ = \ \max_{\mathrm{VL}}\left(\begin{cases} \mathsf{atomic\_only}, & \tau_q' \in \tau_q^*, \\ \mathsf{invisible}, & \text{otherwise} \end{cases}, \begin{cases} \tau_v, & g' \in H \ \wedge \ \tau_q' = \tau_q, \\ \mathsf{invisible}, & \text{otherwise} \end{cases} \right),$$

$$P_{E^{\#},\rho}(z, \mathbf{n}) \ = \ \begin{cases} \{a\}, & (z, \mathbf{n}) \in E^{\#} \text{ where } (a, \_) = \rho(z, \mathbf{n}).b, \\ \emptyset, & \text{otherwise.} \end{cases}$$

`Fence`, `Arrive`, and `Await` (Figure C.13) update ρ solely via $\mathrm{lift}(\rho, \lambda)$ (defined below), which applies λ pointwise to both the read and mutate visibility sets. For $\texttt{Fence}(t, \tau_q^{\mathrm{pre}*}, \tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*})$, $\lambda(r) = \mathcal{A}\big(r,\ g^{\mathrm{exec}*},\ \tau_q^{\mathrm{full}*},\ \tau_q^{\mathrm{temp}*}\big)$ iff the witness $\mathcal{W}\big(t,\ g^{\mathrm{exec}*},\ \tau_q^{\mathrm{pre}*},\ r\big)$ holds, otherwise $\lambda(r) = r$; thus visibility is raised by joining $f \vee f_{\mathrm{aug}}$. For $\texttt{Arrive}(t, \tau_q^{\mathrm{pre}*}, e_z^{\#*})$, the accessed sets yield a unique home barrier $\{(z, \mathbf{n})\} = \bigcap_i W_{e_{zi}^{\#}}$; guarded by $\mathcal{W}$, λ adds the existing arrive count a from $\rho(z, \mathbf{n})$ into $r.p$ at every $(z, \mathbf{n}) \in \mathbf{W}$, and home barrier's arrive count is incremented by $(a' + 1, w')$. For $\texttt{Await}(e_z^{\#}, \tau_q^{\mathrm{full}*}, \tau_q^{\mathrm{temp}*}, n)$, with $(a, w) = \rho(z, \mathbf{n}).b$, compute Max and New as in the rule; set $\lambda(r) = \mathcal{A}\big(r,\ g_{s^{\#}}(\rho),\ \tau_q^{\mathrm{full}*},\ \tau_q^{\mathrm{temp}*}\big)$ iff some observed arrive count $i \in r.p(z, \mathbf{n})$ satisfies $i \leq \mathrm{Max}$, else $\lambda(r) = r$; after lift, update the wait count to (a', New). In short, $\mathcal{W}$ decides when augmentation is permitted, $\mathcal{A}$ raises visibility to `fully_ordered` or `temporally_ordered`.

Definition C.2 (Lift)**.** For $\lambda : \mathbb{R} \to \mathbb{R}$, define $\mathrm{lift}(\rho, \lambda)$ pointwise by

$$\mathrm{lift}(\rho, \lambda)(x, \mathbf{v}) \ = \ (\{\, \lambda(r) \mid r \in R \,\},\ \{\, \lambda(r) \mid r \in M \,\},\ (a,\ w)) \quad \text{where } (R, M, (a, w)) = \rho(x, \mathbf{v}).$$

Definition C.3 (Synchronization helpers)**.** Let $r = (\tau_q^{\mathrm{orig}}, \int, p) \in \mathbb{R}$ with $\int \in \mathbb{S}$. For $\int_1, \int_2 \in \mathbb{S}$, define the pointwise join $(\int_1 \vee \int_2)(g, \tau_q) \ = \ \max_{\mathrm{VL}}\{\int_1(g, \tau_q),\ \int_2(g, \tau_q)\}$.

(i) Witness $\mathcal{W}$. For $t \in \text{Bool}$, $g^{\text{exec}*} \subseteq \mathbb{G}$, and $\tau_q^{\text{pre}*} \subseteq \text{QualTL}$, set

$$\mathcal{W}(t, g^{\text{exec}*}, \tau_q^{\text{pre}*}, r) = \begin{cases} \exists g \in g^{\text{exec}*}\, \exists \tau_q \in \tau_q^{\text{pre}*} :\ \int(g, \tau_q) \succeq \texttt{unordered}, & t = \mathsf{true}, \\ (\tau_q^{\text{orig}} \in \tau_q^{\text{pre}*}) \ \wedge\ \exists g \in g^{\text{exec}*} :\ \int(g, \tau_q^{\text{orig}}) \succeq \texttt{unordered}, & t = \mathsf{false}. \end{cases}$$

(ii) Augment $\mathcal{A}$. Given $g^{\text{exec}*} \subseteq \mathbb{G}$ and $\tau_q^{\text{full}*}, \tau_q^{\text{temp}*} \subseteq \text{QualTL}$, define

$$\mathcal{A}(r, g^{\text{exec}*}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*}) \ =\ (\tau_q^{\text{orig}},\ \textstyle\int \vee \int_{\text{aug}},\ p),$$

$$\textstyle\int_{\text{aug}}(g, \tau_q) = \begin{cases} \texttt{fully\_ordered}, & g \in g^{\text{exec}*} \ \wedge\ \tau_q \in \tau_q^{\text{full}*}, \\ \texttt{temporally\_ordered}, & g \in g^{\text{exec}*} \ \wedge\ \tau_q \in \tau_q^{\text{temp}*} \setminus \tau_q^{\text{full}*}, \\ \texttt{invisible}, & \text{otherwise.} \end{cases}$$

Data expressions $e \in \text{DExpr}$	$\mathcal{P}(\mathsf{Expr}^{\#})$
d	$\emptyset$
x	$\{\, x[\langle\rangle] \,\}$
$x[w^*]$	$\{\, x[w^*] \,\}$
e_1 op e_2 $\quad(\text{op} \in \{+, -, *, /\})$	$T_e(e_1) \ \cup\ T_e(e_2)$
Barrier expressions $e_{\text{z}} \in \text{ZExpr}$	
z	$\{\, z[\langle\rangle] \,\}$
$z[w^*]$	$\{\, z[w^*] \,\}$

Figure C.8: Expression conversion $T_e : \text{DExpr} \cup \text{ZExpr} \to \mathcal{P}(\mathsf{Expr}^{\#})$ collects the distinct reads in an expression and returns them as a set of AMIR reads ($\mathsf{Expr}^{\#}$).

Type	Definition
$T_{\text{smem}} : \mathsf{Expr} \to \mathcal{P}(\mathsf{Expr}^{\#})$	$T_{\text{smem}}(e) \triangleq \{\, e^{\#} \in T_e(e) \mid \text{isSmem}(e^{\#}) \,\}$.
$T_{\text{ex}} : \mathsf{Expr} \to \mathcal{P}(\mathsf{Expr}^{\#})$	$T_{\text{ex}}(e) \triangleq \{\, e^{\#} \in T_e(e) \mid \neg\, \text{isEx}(e^{\#}) \,\}$.
$\text{isEx} : \mathsf{Expr}^{\#} \to \mathsf{Bool}$	$\text{isEx}(e^{\#}) \triangleq$ `true` iff $e^{\#}$ is annotated as sync-exempt memory; `false` otherwise.
$\text{isSmem} : \mathsf{Expr}^{\#} \to \mathsf{Bool}$	$\text{isSmem}(e^{\#}) \triangleq$ `true` iff $e^{\#}$ is annotated as shared memory; `false` otherwise.
$T_{\text{tr}} : \text{SyncTL} \to \mathsf{Bool}$	$T_{\text{tr}}(\tau_s) \triangleq$ `true` iff τ_s is transitive; `false` otherwise.
$T_{\text{qfull}} : \text{SyncTL} \to \mathcal{P}(\text{QualTL})$	$T_{\text{qfull}}(\tau_s) \triangleq \{\, \tau'_q \in \text{QualTL} \mid \tau'_q$ annotated as "full" in $\tau_s \,\}$.
$T_{\text{qtmp}} : \text{SyncTL} \to \mathcal{P}(\text{QualTL})$	$T_{\text{qtmp}}(\tau_s) \triangleq \{\, \tau'_q \in \text{QualTL} \mid \tau'_q$ annotated as "temporal" or "full" in $\tau_s \,\}$.
$T_{\text{qarr}} : \text{SyncTL} \to \text{QualTL}$	$T_{\text{qarr}}(\tau_s) \triangleq \begin{cases} \texttt{Sm80\_cp\_async\_qual}, & \text{if } \texttt{Sm80\_cp\_async\_qual} \in T_{\text{qfull}}(\tau_s), \\ \texttt{cuda\_in\_order\_qual}, & \text{otherwise.} \end{cases}$
$T_{\text{qin}} : \mathsf{Expr}^{\#} \to \text{QualTL}$	$T_{\text{qin}}(e^{\#}) \triangleq \begin{cases} \texttt{cpu\_in\_order\_qual}, & \text{if } e^{\#} \text{ is host/CPU scoped,} \\ \texttt{cuda\_in\_order\_rmem\_qual}, & \text{if } e^{\#} \text{ is register memory,} \\ \texttt{cuda\_in\_order\_ram\_qual}, & \text{otherwise.} \end{cases}$

Figure C.9: Helper functions used by Fig. 7.5 statement conversion. All sets are finite.

Case for $g.p_i$	Abstract Machine IR
Non-atomic *read-only*	$\underset{e^{\#} \in T_{\text{ex}}(e_i)}{;} \Big($ `CheckMutates(full_ordered,` $T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{ext}}(g.p_i))$`;` `RecordRead(` $\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{init}}(g.p_i),\ \emptyset, T_e(e_z))$`)`

Non-atomic *non-read-only*	$\underset{e^{\#} \in T_{\text{ex}}(e_i)}{;} \Big($ `CheckMutates(` $\tau_v^{\text{pre}}, T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{ext}}(g.p_i))$`;`
	 `CheckReads(temporal_ordered,` $T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{ext}}(g.p_i))$`;`
	 `ClearReads(` $e^{\#}$`);`
	 `ClearMutates(` $e^{\#}$`);`
	 `RecordMutate(` $\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{init}}(g.p_i),\ \emptyset, T_e(e_z))$`)`
	 where τ_v^{pre} = `temporal_ordered` if $g.p_i$ is write-only, else `full_ordered`.

Atomic	$\underset{e^{\#} \in T_{\text{ex}}(e_i)}{;} \Big($ `CheckMutates(atomic_only,` $T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{ext}}(g.p_i))$`;`
	 `CheckReads(temporal_ordered,` $T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{ext}}(g.p_i))$`;`
	 `ClearReads(` $e^{\#}$`);` *(note lack of ClearMutates)*
	 `RecordMutate(` $\tau_v^{\text{post}}, T_{\text{cvg}}(g.p_i), e^{\#},\ T_{\text{init}}(g.p_i), T_{\text{atom}}(g.p_i), T_e(e_z))$`)`

Figure C.10: Argument conversion $T_{\text{arg}}(g.p_i, e_i, e_z)$. τ_v^{post} = `unordered` if $T_{\text{ooo}}(g.p_i)$, else `full_ordered`.

Type	Definition
$T_{\text{ext}} : \mathsf{Expr} \to \mathcal{P}(\text{QualTL})$	$T_{\text{ext}}(e) \triangleq \{\, \tau_q \in \text{QualTL} \mid \tau_q \in e.\texttt{ext\_qual\_tl} \,\}$.
$T_{\text{atom}} : \mathsf{Expr} \to \mathcal{P}(\text{QualTL})$	$T_{\text{atom}}(e) \triangleq \{\, \tau_q \in \text{QualTL} \mid \tau_q \in e.\texttt{atomic\_qual\_tl} \,\}$.
$T_{\text{init}} : \mathsf{Expr} \to \text{QualTL}$	$T_{\text{init}}(e) \triangleq e.\texttt{initial\_qual\_tl}$.
$T_{\text{ooo}} : \mathsf{Expr} \to \mathsf{Bool}$	$T_{\text{ooo}}(e) \triangleq e.\texttt{out\_of\_order}$
$T_{\text{cvg}} : \mathsf{Expr} \to \mathsf{Bool}$	$T_{\text{cvg}}(e) \triangleq e.\texttt{convergent}$

Figure C.11: Helper functions used by Fig. C.10 argument conversion.

$$\frac{\langle c_{\text{lo}}, \sigma\rangle \Downarrow_{\mathbb{Z}} m \quad \langle c_{\text{hi}}, \sigma\rangle \Downarrow_{\mathbb{Z}} n \quad m < n \quad \langle s, \imath, \sigma[y \mapsto m], \rho\rangle \Downarrow \imath', \sigma', \rho' \quad \langle \text{for } y \text{ in } (m{+}1) \ldots n \text{ do } s^{\#}, \imath', \sigma', \rho'\rangle \Downarrow \imath'', \sigma'', \rho''}{\langle \text{for } y \text{ in } \texttt{loop}(c_{\text{lo}}, c_{\text{hi}}) \text{ do } s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath'', \sigma'', \rho''}\ \textsc{LoopStep}$$

$$\frac{\langle c_{\text{lo}}, \sigma\rangle \Downarrow_{\mathbb{Z}} m \quad \langle c_{\text{hi}}, \sigma\rangle \Downarrow_{\mathbb{Z}} n \quad m \geq n}{\langle \text{for } y \text{ in } \texttt{loop}(c_{\text{lo}}, c_{\text{hi}}) \text{ do } s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath, \sigma, \rho}\ \textsc{LoopDone}$$

$$\frac{\langle s_1^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma', \rho' \qquad \langle s_2^{\#}, \imath', \sigma', \rho'\rangle \Downarrow \imath'', \sigma'', \rho''}{\langle s_1^{\#};\ s_2^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath'', \sigma'', \rho''}\ \textsc{Seq}$$

$$\frac{\imath' = \imath + 1}{\langle \text{InctaskID}, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma, \rho}\ \textsc{InctaskID} \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\mathsf{Bool}} \text{true} \quad \langle s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma', \rho'}{\langle \text{if } b \text{ then } s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath', \sigma', \rho'}\ \text{IF-}\textsc{True}$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\mathsf{Bool}} \text{false}}{\langle \text{if } b \text{ then } s^{\#}, \imath, \sigma, \rho\rangle \Downarrow \imath, \sigma, \rho}\ \text{IF-}\textsc{False}$$

Figure C.12: Big-step semantics for a loop, sequencing, an explicit task–ID increment, and a guard.

$$\frac{c^* = (c_1, \ldots, c_k) \quad \langle c_i, \sigma\rangle \Downarrow_{\mathbb{Z}} n_i \ \text{ for } i = 1..k \qquad I \ = \ \{\, (v_1, \ldots, v_k) \mid 0 \le v_i < n_i \,\} \qquad \rho' \ = \ \rho[(x, \mathbf{n}) \mapsto (\emptyset, \emptyset, (0, 0))]_{\mathbf{n} \in I}}{\langle \text{Alloc}(x[c^*]),\ \imath,\ \sigma,\ \rho\rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{Alloc}$$

$$\frac{\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e\#} \qquad \rho' \ = \ \rho[(x, \mathbf{n}) \mapsto (\,\emptyset,\ \rho(x, \mathbf{n}).m,\ \rho(x, \mathbf{n}).b\,)]_{\ (x,\mathbf{n}) \in W_{e\#}}}{\langle \text{ClearReads}(e^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow\ \imath,\ \sigma,\ \rho'} \ \textsc{ClearReads}$$

$$\frac{\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e\#} \qquad \rho' \ = \ \rho[(x, \mathbf{n}) \mapsto (\,\rho(x, \mathbf{n}).r,\ \emptyset,\ \rho(x, \mathbf{n}).b\,)]_{\ (x,\mathbf{n}) \in W_{e\#}}}{\langle \text{ClearMutates}(e^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow\ \imath,\ \sigma,\ \rho'} \ \textsc{ClearMutates}$$

$$\frac{\begin{array}{c}\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e\#} \quad \langle e_z^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e_z^{\#}} \\ \rho' \ = \ \rho[(x, \mathbf{n}) \mapsto (\,\rho(x, \mathbf{n}).r \cup \mathcal{R}(\tau_v,\ t,\ \tau_q,\ \tau_q^*,\ W_{e_z^{\#}}, \rho, g_{s\#}(\sigma, \imath)), \\ \rho(x, \mathbf{n}).m,\ \rho(x, \mathbf{n}).b\,)]_{\ (x,\mathbf{n}) \in W_{e\#}}\end{array}}{\langle \text{RecordRead}(\tau_v,\ t,\ e^{\#},\ \tau_q,\ \tau_q^*,\ e_z^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{RecordRead}$$

$$\frac{\begin{array}{c}\langle e^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e\#} \quad \langle e_z^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e_z^{\#}} \\ \rho' \ = \ \rho[(x, \mathbf{n}) \mapsto (\,\rho(x, \mathbf{n}).r,\ \rho(x, \mathbf{n}).m \cup \mathcal{R}(\tau_v,\ t,\ \tau_q,\ \tau_q^*,\ W_{e_z^{\#}}, \\ \rho, g_{s\#}(\sigma, \imath)),\ \rho(x, \mathbf{n}).b\,)]_{\ (x,\mathbf{n}) \in W_{e\#}}\end{array}}{\langle \text{RecordMutate}(\tau_v,\ t,\ e^{\#},\ \tau_q,\ \tau_q^*,\ e_z^{\#}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow \imath, \sigma, \rho'} \ \textsc{RecordMutate}$$

$$\frac{\lambda \ = \ (r \mapsto \mathcal{A}(r, g_{s\#}(\sigma, \imath), \tau_q^{\text{full}*}, \tau_q^{\text{temp}*}) \text{ if } \mathcal{W}(t, g_{s\#}(\sigma, \imath), \tau_q^{\text{pre}*}, r) \text{ else } r)}{\langle \texttt{Fence}(t, \tau_q^{\text{pre}*}, \tau_q^{\text{full}*}, \tau_q^{\text{temp}*}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow\ \imath,\ \sigma,\ \text{lift}(\rho, \lambda)} \ \textsc{Fence}$$

$$\frac{\begin{array}{c}\langle\langle e_{z1}^{\#}, \cdots, e_{zk}^{\#}\rangle, \sigma\rangle \Downarrow_{\mathsf{Acc}} W_{e_{z1}^{\#}}, \cdots, W_{e_{zk}^{\#}} \quad \{(z, \mathbf{n})\} = \bigcap_{i=1}^{k} W_{e_{zi}^{\#}} \quad \mathbf{W} = \bigcup_{i=1}^{k} W_{e_{zi}^{\#}} \\ \lambda \ = \ (r \mapsto (r.q,\ r.f,\ r.p[(z, \mathbf{n}) \mapsto r.p(z, \mathbf{n}) \cup \{a\}]_{z,\mathbf{n} \in \mathbf{W}}^{(a,\_) = \rho(z,\mathbf{n})}) \ \text{ if } \ \mathcal{W}(t, g_{s\#}(\sigma, \imath), \tau_q^{\text{pre}*}, r) \ \text{ else } \ r\,) \\ \rho' = \text{lift}(\rho, \lambda) \qquad (R', M', (a', w')) \ = \ \rho'(z, \mathbf{n}) \qquad \rho'' \ = \ \rho'[(z, \mathbf{n}) \mapsto (R', M', (a' + 1, w'))]\end{array}}{\langle \texttt{Arrive}(t, \tau_q^{\text{pre}*}, e_z^{\#*}),\ \imath,\ \sigma,\ \rho\rangle \Downarrow\ \imath,\ \sigma,\ \rho''} \ \textsc{Arrive}$$

$$\frac{\begin{array}{c}\langle e_z^{\#}, \sigma\rangle \Downarrow_{\mathsf{Acc}} \{(z, \mathbf{n})\} \quad (a, w) = \rho(z, \mathbf{n}).b \\ \text{Max} \ \triangleq \ \text{if } n \ge 0 \text{ then } a - (n + 1) \text{ else } w + n + 1 \\ \text{New} \ \triangleq \ \text{if } n \ge 0 \text{ then } \max(w,\ \text{Max} + 1) \text{ else } w + 1 \\ \lambda \ = \ \Big(r \mapsto \mathcal{A}(r,\ g_{s\#}(\rho),\ \tau_q^{\text{full}*},\ \tau_q^{\text{temp}*}) \text{ if } \ \exists i.\ i \le \text{Max} \wedge i \in r.p(z, \mathbf{n}) \ \text{ else } \ r\Big) \\ \rho' = \text{lift}(\rho, \lambda) \qquad (R', M', (a', \_)) \ = \ \rho'(z, \mathbf{n}) \qquad \rho'' \ = \ \rho'[(z, \mathbf{n}) \mapsto (R', M', (a', \text{New}))]\end{array}}{\langle \texttt{Await}(e_z^{\#},\ \tau_q^{\text{full}*},\ \tau_q^{\text{temp}*},\ n),\ \imath,\ \sigma,\ \rho\rangle \Downarrow\ \imath,\ \sigma,\ \rho''} \ \textsc{Await}$$

Figure C.13: The big-step operational semantics for Abstract Machine statements with sync environment updates.

References

[1] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. "Compute Trends Across Three Eras of Machine Learning." In: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–8. DOI: 10.1109/ijcnn55064.2022.9891914. URL: http://dx.doi.org/10.1109/IJCNN55064.2022.9891914.

[2] Epoch AI. *Data on AI Models*. Accessed: 2026-06-16. URL: https://epoch.ai/data/ai-models.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. ACM, 2017, pp. 1–12. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[4] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. Whitepaper. Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[5] J. L. Hennessy and D. A. Patterson. "A new golden age for computer architecture." *Commun. ACM*, **62**(2), Jan. 2019, pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[6] Z. Xianyi. *OpenBLAS*. Accessed: 2026-06-16. 2011. URL: https://www.openmathlib.org/OpenBLAS/.

[7] NVIDIA. *CUTLASS 4.6.0*. Accessed: 2026-06-18. 2026. URL: https://github.com/NVIDIA/cutlass.

[8] P. Tillet, H. T. Kung, and D. Cox. "Triton: an intermediate language and compiler for tiled neural network computations." In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.

[9] The JAX Authors. *Pallas: a JAX Kernel Language*. Accessed: 2026-06-17. JAX, 2024. URL: https://docs.jax.dev/en/latest/pallas/index.html.

[10] NVIDIA Corporation. *CuTe DSL*. NVIDIA CUTLASS Documentation; last updated 2026-04-08; accessed 2026-06-16. URL: https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html.

[11] NVIDIA Corporation. *cuTile Python*. Version 1.4.0 documentation; accessed: 2026-06-17. NVIDIA, 2025. URL: https://docs.nvidia.com/cuda/cutile-python/.

[12] Modular Inc. *Mojo*. Accessed: 2026-06-16. URL: https://mojolang.org/.

[13] PyTorch Team at Meta. *Helion: A High-Level DSL for Performant and Portable ML Kernels*. Accessed: 2026-06-16. Oct. 2025. URL: https://pytorch.org/blog/helion/.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595. URL: https://doi.org/10.1145/1375581.1375595.

[15] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. "The Polyhedral Model Is More Widely Applicable Than You Think." In: *Compiler Construction*. Ed. by R. Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–303. ISBN: 978-3-642-11970-5.

[16] T. Grosser, A. Größlinger, and C. Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation." *Parallel Process. Lett.*, **22**, 2012. URL: https://api.semanticscholar.org/CorpusID:18533155.

[17] A. Adams et al. "Learning to optimize Halide with tree search and random programs." *ACM Trans. Graph.*, **38**(4), 2019, 121:1–121:12. DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

[18] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. "Automatically scheduling halide image processing pipelines." *ACM Trans. Graph.*, **35**(4), 2016, 83:1–83:11. DOI: 10.1145/2897824.2925952. URL: https://doi.org/10.1145/2897824.2925952.

[19] L. Anderson, A. Adams, K. Ma, T.-M. Li, T. Jin, and J. Ragan-Kelley. "Efficient automatic scheduling of imaging and vision pipelines for the GPU." *Proc. ACM Program. Lang.*, **5**(OOPSLA), Oct. 2021. DOI: 10.1145/3485486. URL: https://doi.org/10.1145/3485486.

[20] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning." In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '20. 2020, pp. 863–879. DOI: 10.5555/3488766.3488815.

[21] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. "Learning to Optimize Tensor Programs." In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 2018, pp. 3393–3404. URL: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.

[22] J. Shao, X. Zhou, S. Feng, B. Hou, R. Lai, H. Jin, W. Lin, M. Masuda, C. H. Yu, and T. Chen. "Tensor program optimization with probabilistic programs." In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[23] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. "Equality Saturation: A New Approach to Optimization." In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. New York, NY, USA: Association for Computing Machinery, 2009, pp. 264–276.

[24] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. "egg: Fast and extensible equality saturation." *Proc. ACM Program. Lang.*, **5**(POPL), Jan. 2021. DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

[25] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA, 2013, pp. 519–530.

[26] Triton Project. *Gluon Overview*. Triton documentation; accessed 2026-06-16. URL: https://triton-lang.org/main/gluon/index.html.

[27] M. B. S. Ahmad, A. J. Root, A. Adams, S. Kamil, and A. Cheung. "Vector Instruction Selection for Digital Signal Processors Using Program Synthesis." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 1004–1016. ISBN: 9781450392051. DOI: 10.1145/3503222.3507714. URL: https://doi.org/10.1145/3503222.3507714.

[28] A. J. Root, M. B. S. Ahmad, D. Sharlet, A. Adams, S. Kamil, and J. Ragan-Kelley. "Fast Instruction Selection for Fast Digital Signal Processing." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 125–137. DOI: 10.1145/3623278.3624768. URL: https://doi.org/10.1145/3623278.3624768.

[29] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley. "Exocompilation for productive programming of hardware accelerators." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718. ISBN: 9781450392655. DOI: 10.1145/3519939.3523446. URL: https://doi.org/10.1145/3519939.3523446.

[30] Y. Ikarashi, K. Qian, S. Droubi, A. Reinking, G. L. Bernstein, and J. Ragan-Kelley. "Exo 2: Growing a Scheduling Language." In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 426–444. ISBN: 9798400706981. DOI: 10.1145/3669940.3707218. URL: https://doi.org/10.1145/3669940.3707218.

[31] Y. Ikarashi, C. Wang, J. Ragan-Kelley, and G. L. Bernstein. "Value-Sensitive Analysis for Looping Array Code." In submission. 2026.

[32] D. Z. Akeley, Y. Ikarashi, and J. Ragan-Kelley. "Exo-GPU: Safe, Imperative, User-schedulable Programming for Tensor Cores." In submission. 2026.

[33] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. The Morgan Kaufmann Series in Computer Architecture and Design. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 978-0-12-811905-1.

[34] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Commun. ACM*, **52**(4), Apr. 2009, pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[35] M. D. Lam, E. E. Rothberg, and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 63–74. DOI: 10.1145/106972.106981. URL: https://doi.org/10.1145/106972.106981.

[36] K. Goto and R. Van De Geijn. "Anatomy of high-performance matrix multiplication." *ACM Trans. Math. Softw.*, **34**(3), May 2008. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: https://doi.org/10.1145/1356052.1356053.

[37] F. G. Van Zee and R. Van De Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality." *ACM Trans. Math. Softw.*, **41**(3), June 2015. ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

[38] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual, Volume 2: Earlier Generations of Intel 64 and IA-32 Processor Architectures, Throughput and Latency*. Version 050; document number 356477-050US. Intel Corporation. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel64-and-ia32-architectures-optimization.html (visited on 07/03/2026).

[39] H. Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration." In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[40] Amazon Web Services. *Trainium/Inferentia2 Architecture Guide for NKI*. AWS Neuron Documentation. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/guides/architecture/trainium_inferentia2_arch.html, accessed 2026-07-03.

[41] P. Feautrier. "Dataflow analysis of array and scalar references." *Int. J. Parallel Program.*, **20**(1), 1991, pp. 23–53. DOI: 10.1007/BF01407931. URL: https://doi.org/10.1007/BF01407931.

[42] H. Xi and F. Pfenning. "Dependent Types in Practical Programming." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 214–227. ISBN: 1581130953. DOI: 10.1145/292540.292560. URL: https://doi.org/10.1145/292540.292560.

[43] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. "Halide: decoupling algorithms from schedules for high-performance image processing." *Commun. ACM*, **61**(1), 2018, pp. 106–115. DOI: 10.1145/3150211. URL: https://doi.org/10.1145/3150211.

[44] T. Chen et al. "TVM: An Automated End-to-end Optimizing Compiler for Deep Learning." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291211.

[45] M. Steuwer, T. Remmelg, and C. Dubach. "LIFT: A functional data-parallel IR for high-performance GPU code generation." In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[46] B. Hagedorn, J. Lenfers, T. Koehler, S. Gorlatch, and M. Steuwer. *A Language for Describing Optimization Strategies*. 2020. arXiv: 2002.02268 [cs.PL].

[47] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." *ACM Trans. Graph.*, **31**(4), 2012, 32:1–32:12. DOI: 10.1145/2185520.2185528. URL: https://doi.org/10.1145/2185520.2185528.

[48] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. "The tensor algebra compiler." *Proceedings of the ACM on Programming Languages*, **1**(OOPSLA), Oct. 2017, pp. 1–29. DOI: 10.1145/3133901. URL: https://doi.org/10.1145/3133901.

[49] Y. Hu, T. Li, L. Anderson, J. Ragan-Kelley, and F. Durand. "Taichi: a language for high-performance computation on spatially sparse data structures." *ACM Trans. Graph.*, **38**(6), 2019, 201:1–201:16. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

[50] A. Reinking, G. Bernstein, and J. Ragan-Kelley. "Formal Semantics for the Halide Language." MA thesis. EECS Department, University of California, Berkeley, May 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-40.html.

[51] G. L. Steele Jr. "Growing a language." In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1998.

[52] A. Kothari, A. R. Noor, M. Xu, H. Uddin, D. Baronia, S. Baziotis, V. Adve, C. Mendis, and S. Sengupta. "Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures." In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 514–529. ISBN: 9798400703850. DOI: 10.1145/3620665.3640385. URL: https://doi.org/10.1145/3620665.3640385.

[53] A. Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs." *IEEE Micro*, **40**(4), 2020, pp. 10–21. DOI: 10.1109/MM.2020.2996616.

[54] S. Karandikar et al. “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud.” In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE, 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.

[55] S. P. Luttik and E. Visser. “Specification of Rewriting Strategies.” In: *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications*. Algebraic'97. Amsterdam, The Netherlands: BCS Learning & Development Ltd., 1997, p. 9. ISBN: 3540762280.

[56] M. G. J. van den Brand, P. Klint, and J. J. Vinju. “Term Rewriting with Traversal Functions.” *ACM Trans. Softw. Eng. Methodol.*, **12**(2), Apr. 2003, pp. 152–190. ISSN: 1049-331X. DOI: 10.1145/941566.941568. URL: https://doi.org/10.1145/941566.941568.

[57] Halide Contributors. *Halide Github Repository*. https://github.com/halide/Halide. Accessed: 2026-07-17. 2013.

[58] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang. “Allo: A Programming Model for Composable Accelerator Design.” *Proc. ACM Program. Lang.*, **8**(PLDI), June 2024. DOI: 10.1145/3656401. URL: https://doi.org/10.1145/3656401.

[59] D. Gopan, T. Reps, and M. Sagiv. “A framework for numeric analysis of array operations.” *SIGPLAN Not.*, **40**(1), Jan. 2005, pp. 338–350. ISSN: 0362-1340. DOI: 10.1145/1047659.1040333. URL: https://doi.org/10.1145/1047659.1040333.

[60] D. Monniaux and F. Alberti. “A Simple Abstraction of Arrays and Maps by Program Translation.” In: *Static Analysis*. Ed. by S. Blazy and T. Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 217–234. ISBN: 978-3-662-48288-9.

[61] P. Cousot, R. Cousot, and F. Logozzo. “A parametric segmentation functor for fully automatic and scalable array content analysis.” In: POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 105–118. ISBN: 9781450304900. DOI: 10.1145/1926385.1926399. URL: https://doi.org/10.1145/1926385.1926399.

[62] I. Dillig, T. Dillig, and A. Aiken. “Fluid updates: beyond strong vs. weak updates.” In: *Proceedings of the 19th European Conference on Programming Languages and Systems*. ESOP'10. Paphos, Cyprus: Springer-Verlag, 2010, pp. 246–266. ISBN: 3642119565. DOI: 10.1007/978-3-642-11957-6_14. URL: https://doi.org/10.1007/978-3-642-11957-6_14.

[63] A. Laird, B. Liu, N. S. Bjørner, and M. M. Dehnavi. “SpEQ: Translation of Sparse Codes using Equivalences.” *Proc. ACM Program. Lang.*, **8**(PLDI), 2024, pp. 1680–1703. DOI: 10.1145/3656445. URL: https://doi.org/10.1145/3656445.

[64] W. S. Moses, I. R. Ivanov, J. Domke, T. Endo, J. Doerfert, and O. Zinenko. “High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs.” In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*. Ed. by M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy. ACM, 2023, pp. 119–134. DOI: 10.1145/3572848.3577475. URL: https://doi.org/10.1145/3572848.3577475.

[65] G. E. Collins. "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition: a synopsis." *SIGSAM Bull.*, **10**(1), Feb. 1976, pp. 10–12. ISSN: 0163-5824. DOI: 10.1145/1093390.1093393. URL: https://doi.org/10.1145/1093390.1093393.

[66] G. E. Collins. "Quantifier Elimination by Cylindrical Algebraic Decomposition — Twenty Years of Progress." In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Ed. by B. F. Caviness and J. R. Johnson. Texts and Monographs in Symbolic Computation. Vienna: Springer, 1998, pp. 8–23. DOI: 10.1007/978-3-7091-9459-1_2.

[67] N. Halbwachs and M. Péron. "Discovering properties about arrays in simple programs." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 339–348. ISBN: 9781595938602. DOI: 10.1145/1375581.1375623. URL: https://doi.org/10.1145/1375581.1375623.

[68] J. Zhao, D. Grubb, M. Rusch, T. Wei, K. Anderson, B. Nikolic, and K. Asanović. *The Saturn Microarchitecture Manual*. Tech. rep. UCB/EECS-2024-215. Dec. 2024. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-215.html.

[69] R. J. Hanson, F. T. Krogh, and C. L. Lawson. *A Proposal for Standard Linear Algebra Subprograms*. Technical Memorandum 33-660. Pasadena, California: Jet Propulsion Laboratory, California Institute of Technology, Nov. 1973. URL: https://ntrs.nasa.gov/citations/19740005175.

[70] J. Zhao, D. Grubb, M. Rusch, T. Wei, K. Anderson, B. Nikolic, and K. Asanović. *The Saturn Vector Unit*. https://github.com/ucb-bar/saturn-vectors/tree/master. Accessed: 2025-11-07. 2024.

[71] NVIDIA. *Parallel Thread Execution ISA Version 9.0*. 2025. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[72] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608 [cs.LG]. URL: https://arxiv.org/abs/2407.08608.

[73] NVIDIA. *CuTe*. 2025. URL: https://docs.nvidia.com/cutlass/media/docs/cpp/cute.

[74] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré. *ThunderKittens: Simple, Fast, and Adorable AI Kernels*. 2024. arXiv: 2410.20399 [cs.LG]. URL: https://arxiv.org/abs/2410.20399.

[75] NVIDIA. *CUTLASS Grouped Kernel Schedules*. 2025. URL: https://docs.nvidia.com/cutlass/media/docs/cpp/grouped_scheduler.html.

[76] B. Lee. *CUDA HGEMV*. 2024. URL: https://github.com/Bruce-Lee-LY/cuda_hgemv.

[77] NVIDIA. *H100 GPU*. 2025. URL: https://www.nvidia.com/en-us/data-center/h100/.

[78] M. Jarzynski and M. Olano. *A PCG3D Family of Hash Functions*. 2020. URL: https://github.com/markjarzynski/PCG3D.

[79] T. Sheard. "Accomplishments and Research Challenges in Meta-Programming." In: *Proceedings of the 2nd International Conference on Semantics, Applications, and Implementation of Program Generation*. SAIG'01. Florence, Italy: Springer-Verlag, 2001, pp. 2–44. ISBN: 3540425586.

[80] W. Taha and T. Sheard. "Multi-stage programming with explicit annotations." In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '97. Amsterdam, The Netherlands: Association for Computing Machinery, 1997, pp. 203–217. ISBN: 0897919173.

[81] W. Taha. "Multi-Stage Programming: Its Theory and Applications." PhD thesis. Oregon Graduate Institute of Science and Technology, 1999.

[82] G. Mainland. "Explicitly heterogeneous metaprogramming with MetaHaskell." In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 311–322. ISBN: 9781450310543. DOI: 10.1145/2364527.2364572. URL: https://doi.org/10.1145/2364527.2364572.

[83] O. Kiselyov. "The Design and Implementation of BER MetaOCaml - System Description." In: *Fuji International Symposium on Functional and Logic Programming*. Cham, Switzerland: Springer International Publishing, 2014, pp. 86–102. URL: https://api.semanticscholar.org/CorpusID:9880167.

[84] J. McCarthy. "LISP: A Programming System for Symbolic Manipulations." In: *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*. ACM '59. Cambridge, Massachusetts: Association for Computing Machinery, 1959, pp. 1–4. ISBN: 9781450373647. DOI: 10.1145/612201.612243. URL: https://doi.org/10.1145/612201.612243.

[85] B. C. Smith. "Reflection and Semantics in LISP." In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 23–35. ISBN: 0897911253. DOI: 10.1145/800017.800513. URL: https://doi.org/10.1145/800017.800513.

[86] B. C. Smith. "Procedural reflection in programming languages." PhD thesis. Massachusetts Institute of Technology, 1982.

[87] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. "The Racket Manifesto." In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113. URL: https://drops.dagstuhl.de/opus/volltexte/2015/5021.

[88] M. Ballantyne, A. King, and M. Felleisen. "Macros for domain-specific languages." *Proc. ACM Program. Lang.*, **4**(OOPSLA), Nov. 2020. DOI: 10.1145/3428297. URL: https://doi.org/10.1145/3428297.

[89] L. Parreaux and A. Shaikhha. "Multi-stage programming in the large with staged classes." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 35–49. ISBN: 9781450381741. DOI: 10.1145/3425898.3426961. URL: https://doi.org/10.1145/3425898.3426961.

[90] A. Brahmakshatriya and S. Amarasinghe. "BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. New York, NY, USA: ACM, 2021, pp. 39–51. DOI: 10.1109/CGO51591.2021.9370333.

[91] N. Xie, L. White, O. Nicole, and J. Yallop. "MacoCaml: Staging Composable and Compilable Macros." *Proc. ACM Program. Lang.*, **7**(ICFP), Aug. 2023. DOI: 10.1145/3607851. URL: https://doi.org/10.1145/3607851.

[92] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. "Terra: A Multi-Stage Language for High-Performance Computing." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 105–116. ISBN: 9781450320146. DOI: 10.1145/2491956.2462166. URL: https://doi.org/10.1145/2491956.2462166.

[93] Z. DeVito and P. Hanrahan. "The Design of Terra: Harnessing the Best Features of High-Level and Low-Level Languages." In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morriset. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 79–89. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.79. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2015.79.

[94] R. Ierusalimschy, L. H. H. de Figueiredo, and W. Celes. "The Implementation of Lua 5.0." *JUCS - Journal of Universal Computer Science*, **11**(7), 2005, pp. 1159–1176. ISSN: 0948-695X. DOI: 10.3217/jucs-011-07-1159. eprint: https://doi.org/10.3217/jucs-011-07-1159. URL: https://doi.org/10.3217/jucs-011-07-1159.

[95] The jQuery Foundation. *jQuery Official Documentation*. https://jquery.com/. Accessed: 2024-06-01. 2006.

[96] J. Clark. *XSL Transformations (XSLT) Version 1.0*. https://www.w3.org/TR/xslt. World Wide Web Consortium (W3C) Recommendation. 1999.

[97] S. S. Huang, D. Zook, and Y. Smaragdakis. "Statically Safe Program Generation with SafeGen." In: *Generative Programming and Component Engineering*. Ed. by R. Glück and M. Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 309–326. ISBN: 978-3-540-31977-1.

[98] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. "Legion: expressing locality and independence with logical regions." In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12*. Salt Lake City, UT, USA: IEEE, Nov. 2012, p. 66. DOI: 10.1109/SC.2012.71. URL: https://doi.org/10.1109/SC.2012.71.

[99] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730 [cs.PL].

[100] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe. "GraphIt: a high-performance graph DSL." *PACMPL*, **2**(OOPSLA), 2018, 121:1–121:30. DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.

[101] A. Venkat, T. Rusira, R. Barik, M. Hall, and L. Truong. "SWIRL: High-performance many-core CPU code generation for deep neural networks." *The International Journal of High Performance Computing Applications*, **33**(6), 2019, pp. 1275–1289. DOI: 10.1177/1094342019866247.

[102] B. Hagedorn, A. S. Elliott, H. Barthels, R. Bodik, and V. Grover. *Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs*. 2020. arXiv: 2003.06324 [cs.PL].

[103] S. Feng et al. "TensorIR: An Abstraction for Automatic Tensorized Program Optimization." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 804–817. DOI: 10.1145/3575693.3576933. URL: https://doi.org/10.1145/3575693.3576933.

[104] R. Yadav, M. Garland, A. Aiken, and M. Bauer. "Task-Based Tensor Computations on Modern GPUs." *Proc. ACM Program. Lang.*, **9**(PLDI), 2025. DOI: 10.1145/3729262. URL: https://doi.org/10.1145/3729262.

[105] S. Donadio, J. C. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. A. Padua, and K. Pingali. "A Language for the Compact Representation of Multiple Program Versions." In: *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 136–151. DOI: 10.1007/978-3-540-69330-7_10. URL: https://doi.org/10.1007/978-3-540-69330-7_10.

[106] K. Fatahalian et al. "Sequoia: Programming the Memory Hierarchy." In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 83–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188543. URL: https://doi.org/10.1145/1188455.1188543.

[107] Q. Yi, K. Seymour, H. You, R. W. Vuduc, and D. J. Quinlan. "POET: Parameterized Optimizations for Empirical Tuning." In: *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. Rome, Italy: IEEE, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370637. URL: https://doi.org/10.1109/IPDPS.2007.370637.

[108] A. Hartono, B. Norris, and P. Sadayappan. "Annotation-based empirical performance tuning using Orio." In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. Rome, Italy: IEEE, 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161004. URL: https://doi.org/10.1109/IPDPS.2009.5161004.

[109] C. Chen, J. Chame, and M. Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Southern California, 2008.

[110] F. Franchetti, T. M. Low, D. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. "SPIRAL: Extreme Performance Portability." *Proceedings of the IEEE*, **106**(11), 2018, pp. 1935–1968. DOI: 10.1109/JPROC.2018.2873289. URL: https://doi.org/10.1109/JPROC.2018.2873289.

[111] S. Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software – ICMS 2010*. Ed. by K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama. Berlin, Heidelberg: Springer, 2010, pp. 299–302. ISBN: 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49.

[112] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. "Schedule Trees." In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Ed. by S. Rajopadhye and S. Verdoolaege. Vienna, Austria: INRIA, Jan. 2014, pp. 1–9.

[113] R. Baghdadi et al. "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming." In: *PACT*. San Francisco, CA, USA: IEEE Computer Society, 2015, pp. 138–149. DOI: 10.1109/PACT.2015.17.

[114] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code." In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019*. Washington, DC, USA: IEEE, 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197. URL: https://doi.org/10.1109/CGO.2019.8661197.

[115] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. "AlphaZ: A System for Design Space Exploration in the Polyhedral Model." In: *Languages and Compilers for Parallel Computing*. Ed. by H. Kasahara and K. Kimura. Berlin, Heidelberg: Springer, 2013, pp. 17–31. ISBN: 978-3-642-37658-0. DOI: 10.1007/978-3-642-37658-0_2.

[116] A. Susungi, N. A. Rink, A. Cohen, J. Castrillon, and C. Tadonki. "Meta-Programming for Cross-Domain Tensor Optimizations." In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 79–92. ISBN: 9781450360456. DOI: 10.1145/3278122.3278131. URL: https://doi.org/10.1145/3278122.3278131.

[117] M. P. Lücke, O. Zinenko, W. S. Moses, M. Steuwer, and A. Cohen. *The MLIR Transform Dialect. Your compiler is more powerful than you think*. 2024. arXiv: 2409.03864 [cs.PL]. URL: https://arxiv.org/abs/2409.03864.

[118] Y. Ikarashi, J. Ragan-Kelley, T. Fukusato, J. Kato, and T. Igarashi. "Guided Optimization for Image Processing Pipelines." In: *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Piscataway, NJ, USA: IEEE, 2021, pp. 1–5. DOI: 10.1109/VL/HCC51201.2021.9576341.

[119] T. Kœhler, A. Goens, S. Bhat, T. Grosser, P. Trinder, and M. Steuwer. "Guided Equality Saturation." *Proc. ACM Program. Lang.*, **8**(POPL), Jan. 2024. DOI: 10.1145/3632900. URL: https://doi.org/10.1145/3632900.

[120] C. B. Jay and M. Sekanina. "Shape Checking of Array Programs." In: *Computing: the Australasian Theory Symposium*. Sydney, Australia, 1997.

[121] C. B. Jay and P. A. Steckler. "The functional imperative: Shape!" In: *Programming Languages and Systems*. Ed. by C. Hankin. Berlin, Heidelberg: Springer, 1998, pp. 139–153. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053568.

[122] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization." *ACM Trans. Program. Lang. Syst.*, **9**(3), July 1987, pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041.

[123] P. Feautrier and C. Lengauer. "The Polyhedron Model." In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer, 2011, pp. 1581–1592. DOI: 10.1007/978-0-387-09766-4_502.

[124] D. K. Gifford and J. M. Lucassen. "Integrating Functional and Imperative Programming." In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 28–38. ISBN: 0897912004. DOI: 10.1145/319838.319848. URL: https://doi.org/10.1145/319838.319848.

[125] A. J. Bernstein. "Analysis of Programs for Parallel Processing." *IEEE Transactions on Electronic Computers*, **EC-15**(5), 1966, pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[126] C. Urban and A. Miné. "A Decision Tree Abstract Domain for Proving Conditional Termination." In: *Static Analysis*. Ed. by M. Müller-Olm and H. Seidl. Cham: Springer International Publishing, 2014, pp. 302–318. ISBN: 978-3-319-10936-7.

[127] G. M. Essertel, G. Wei, and T. Rompf. "Precise reasoning with structured time, structured heaps, and collective operations." *Proc. ACM Program. Lang.*, **3**(OOPSLA), Oct. 2019. DOI: 10.1145/3360583. URL: https://doi.org/10.1145/3360583.

[128] Y. Guo, P. Yao, and C. Zhang. "Precise Compositional Buffer Overflow Detection via Heap Disjointness." In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024, pp. 63–75.

[129] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS 2000)*. San Diego, California, USA: The Internet Society, Feb. 2000, pp. 3–17. ISBN: 1-891562-07-X. URL: https://www.ndss-symposium.org/ndss2000/first-step-towards-automated-detection-buffer-overrun-vulnerabilities-paper/.

[130] I. Dillig, T. Dillig, and A. Aiken. "Precise reasoning for programs using containers." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 187–200. ISBN: 9781450304900. DOI: 10.1145/1926385.1926407. URL: https://doi.org/10.1145/1926385.1926407.

[131] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[132] J. Liu and X. Rival. "Abstraction of Arrays Based on Non Contiguous Partitions." In: *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2015, pp. 282–299.

[133] A. Liu, G. L. Bernstein, A. Chlipala, and J. Ragan-Kelley. "Verified Tensor-Program Optimization Via High-level Scheduling Rewrites." *Proc. ACM Program. Lang.*, **6**(POPL), Jan. 2022. DOI: 10.1145/3498717.

[134] L.-N. Pouchet, E. Tucker, N. Zhang, H. Chen, D. Pal, G. Rodríguez, and Z. Zhang. "Formal Verification of Source-to-Source Transformations for HLS." In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 97–107. ISBN: 9798400704185. DOI: 10.1145/3626202.3637563. URL: https://doi.org/10.1145/3626202.3637563.

[135] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Heidelberg, Germany: Springer Berlin, 1979.

[136] R. Milner. "Elements of Interaction: Turing Award Lecture." *Commun. ACM*, **36**(1), Jan. 1993, pp. 78–89. ISSN: 0001-0782. DOI: 10.1145/151233.151240. URL: https://doi.org/10.1145/151233.151240.

[137] G. Plotkin, C. Stirling, and M. Tofte. *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. One Broadway, Broadway 12th Floor, Cambridge, MA 02142: The MIT Press, 2000.

[138] R. Milner. "A theory of type polymorphism in programming." *Journal of Computer and System Sciences*, **17**(3), 1978, pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[139] J. H. Geuvers. "Proof assistants: History, ideas and future." *Sadhana*, **34**, 2009, pp. 3–25. URL: https://api.semanticscholar.org/CorpusID:14827467.

[140] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, 2006.

[141] P. M. Phothilimthana, A. S. Elliott, A. Wang, A. Jangda, B. Hagedorn, H. Barthels, S. J. Kaufman, V. Grover, E. Torlak, and R. Bodik. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 65–78. ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. URL: https://doi.org/10.1145/3297858.3304059.

[142] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson. "Vectorization for Digital Signal Processors via Equality Saturation." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. URL: https://doi.org/10.1145/3445814.3446707.

[143] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." *ACM Trans. Math. Softw.*, **5**(3), Sept. 1979, pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847.

[144] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. "An extended set of FORTRAN basic linear algebra subprograms." *ACM Trans. Math. Softw.*, **14**(1), Mar. 1988, pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/42288.42291. URL: https://doi.org/10.1145/42288.42291.

[145] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. "A set of level 3 basic linear algebra subprograms." *ACM Trans. Math. Softw.*, **16**(1), Mar. 1990, pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: https://doi.org/10.1145/77626.79170.

[146] Intel. *Accelerate fast math with Intel oneAPI math kernel library*. 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[147] Apple. *Accelerate Framework*. 2024. URL: https://developer.apple.com/documentation/accelerate/blas/.

[148] Nvidia. *CuBLAS*. 2024. URL: https://developer.nvidia.com/cublas.

[149] R. Clint Whaley, A. Petitet, and J. J. Dongarra. "Automated empirical optimizations of software and the ATLAS project." *Parallel Computing*, **27**(1), 2001. New Trends in High Performance Computing, pp. 3–35. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(00)00087-9. URL: https://www.sciencedirect.com/science/article/pii/S0167819100000879.

[150] K. Goto and R. Van De Geijn. "High-performance implementation of the level-3 BLAS." *ACM Trans. Math. Softw.*, **35**(1), July 2008. ISSN: 0098-3500. DOI: 10.1145/1377603.1377607. URL: https://doi.org/10.1145/1377603.1377607.

[151] A. Castelló, J. Bellavita, G. Dinh, Y. Ikarashi, and H. Martínez. "Tackling the Matrix Multiplication Micro-kernel Generation with Exo." In: *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '24. Edinburgh, United Kingdom: IEEE Press, 2024, pp. 182–192. ISBN: 9798350395099. DOI: 10.1109/CGO57630.2024.10444883. URL: https://doi.org/10.1109/CGO57630.2024.10444883.

[152] B. Köpcke, S. Gorlatch, and M. Steuwer. "Descend: A Safe GPU Systems Programming Language." *Proc. ACM Program. Lang.*, **8**(PLDI), 2024. DOI: 10.1145/3656411. URL: https://doi.org/10.1145/3656411.

[153] A. Castelló, H. Martínez, S. Catalán, J. Lei, Y. Ikarashi, G. Dinh, F. D. Igual, and E. S. Quintana-Ortí. "Portable, High Performance Matrix Multiplication Micro-Kernels for RISC-V with ExO." In: *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2025, pp. 25–32.

[154] R. Iwai, J. Domke, E. Vatai, and Y. Sato. "Prototyping an Autotuning Framework for Program Optimization Using Exo Language." In: *Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops*. SCA/HPCAsiaWS '26. Association for Computing Machinery, 2026, pp. 156–164. ISBN: 9798400723285. DOI: 10.1145/3784828.3785400. URL: https://doi.org/10.1145/3784828.3785400.

[155] R. Iwai, E. Vatai, J. Domke, and Y. Sato. "Evaluation of Vectorization Methods on Arm SVE Using the Exo Language." In: *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. 2024, pp. 178–179.

[156] C. Hong, S. Bhatia, A. Cheung, and Y. S. Shao. *Autocomp: LLM-Driven Code Optimization for Tensor Accelerators*. 2025. arXiv: 2505.18574 [cs.PL]. URL: https://arxiv.org/abs/2505.18574.

[157] S. McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition." In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Ed. by B. F. Caviness and J. R. Johnson. Texts and Monographs in Symbolic Computation. Vienna: Springer, 1998, pp. 242–268. DOI: 10.1007/978-3-7091-9459-1_12.

[158] C. W. Brown. "Improved Projection for Cylindrical Algebraic Decomposition." *Journal of Symbolic Computation*, **32**(5), 2001, pp. 447–465. DOI: 10.1006/jsco.2001.0463.

[159] H. Hong. "An improvement of the projection operator in cylindrical algebraic decomposition." In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. ISSAC '90. Tokyo, Japan: Association for Computing Machinery, 1990, pp. 261–264. ISBN: 0201548925. DOI: 10.1145/96877.96943. URL: https://doi.org/10.1145/96877.96943.

[160] D. M. R. Park. "Fixpoint Induction and Proofs of Program Properties." In: *Machine Intelligence Volume 5*. Edinburgh University Press. 1969, pp. 59–78.

[161] H. Bekic. "Definable Operation in General Algebras, and the Theory of Automata and Flowcharts." In: *Programming Languages and Their Definition*. Springer. 1984, pp. 30–55.