Software-Defined Vector Processing on Manycore Fabrics

Philip Bedoukian Neil Adit Edwin Peguero Adrian Sampson
Cornell University
Ithaca, NY, USA

ABSTRACT

We describe a tiled architecture that can fluidly transition between manycore (MIMD) and vector (SIMD) execution. The hardware provides a *software-defined vector* programming model that lets applications aggregate groups of manycore tiles into logical vector engines. In manycore mode, the machine behaves as a standard parallel processor. In vector mode, groups of tiles repurpose their functional units as vector execution lanes and scratchpads as vector memory banks. The key mechanism is an *instruction forwarding network:* a single tile fetches instructions and sends them to other trailing cores. Most cores disable their frontends and instruction caches, so vector groups amortize the intrinsic hardware costs of von Neumann control. Vector groups also use a decoupled access/execute scheme to centralize their memory requests and issue coalesced, wide loads.

We augment an existing RISC-V manycore design with a minimal hardware extension to implement software-defined vectors. Cycle-level simulation results show that software-defined vectors improve performance by an average of 1.7× over standard MIMD execution while saving 22% of the energy. Compared to a similarly configured GPU, the architecture improves performance by 1.9×.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *Reconfigurable computing*; *Multiple instruction, multiple data*; *Single instruction, multiple data.*

KEYWORDS

Manycore, reconfigurable hardware, vector machines, SIMD

ACM Reference Format:
Philip Bedoukian, Neil Adit, Edwin Peguero, and Adrian Sampson. 2021. Software-Defined Vector Processing on Manycore Fabrics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480099

1 INTRODUCTION

Tiled manycore processors [8, 21, 29, 31] and vector machines [18, 19, 24] represent contrasting approaches to scalable parallel computation. The difference is between MIMD and SIMD parallelism: manycores offer flexible programmability, while vector machines exploit control and data regularity to amortize control overheads.

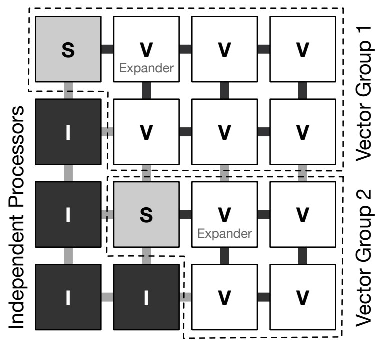


Figure 1: A manycore fabric with two software-defined vector groups. Cores can be in one of three modes: *scalar* cores lead vector groups, *vector* cores follow scalar cores, and *independent* cores operate in a plain manycore style. Cores can change which mode they are in during run time.

In practice, workloads are neither perfectly regular nor entirely irregular—computations exist on a spectrum of regularity. Successful parallel machines therefore tend to combine aspects of both SIMD and MIMD parallelism. GPGPUs augment vector engines with sophisticated runtime monitoring to cope with irregularity, and Larrabee and its descendants [25, 33] augment manycore tiles with small SIMD functional units. These standard approaches, however, "bake in" a specific point in the trade-off space between regular and irregular parallelism and suffer low utilization for other points in the spectrum.

We propose a manycore architecture that can flexibly adopt the efficiency advantages of a vector machine. The architecture supports run-time configuration to group together small, independent computation tiles to form large, SIMD-optimized vector engines. We introduce *software-defined vectors:* an architectural abstraction that lets a single manycore fabric operate as completely independent processors, large aggregate vector engines, and as flexible combinations of these two extremes.

Figure 1 gives the software view of a 4×4 tiled machine with software-defined vectors. The software can coalesce tiles into *vector groups* that execute in lockstep and disable the processors' frontends, including their instruction caches. A vector group consists of one *scalar core*, which runs scalar computations and issues memory accesses for the group, and several *vector cores* that share a single SIMD instruction stream. A vector group emulates a classic vector-thread architecture [16]: the scalar core controls execution by launching data-parallel microthreads that run on the vector cores. Tiles that are not part of a vector group execute in an *independent* mode where they continue to behave as in a typical manycore.

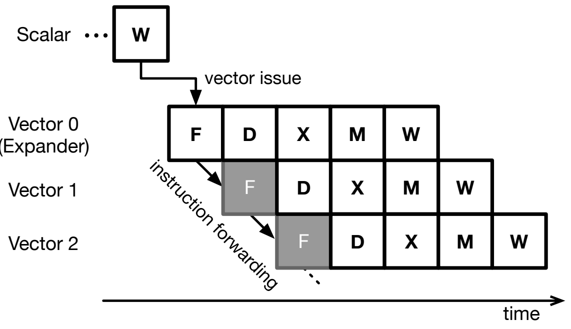


Figure 2: Execution in a Rockcress vector group using a classic RISC pipeline. The scalar core starts a microthread with a *vector issue* instruction, which tells the first vector core (the *expander*) to begin fetching instructions. The expander core fetches an instruction from its instruction cache and sends it to the second vector core over the *instruction forwarding network* (inet). Subsequent vector cores disable their fetch logic and accept instructions from the inet instead.

Vector machines offer three main efficiency advantages: compute density, coalesced wide memory accesses, and control cost amortization. Compared to standard per-core hardware SIMD units, software-defined vector groups sacrifice sheer compute density but more flexibly amortize memory and control overheads. The key research challenge in this paper is realizing these benefits while reusing the distributed computational resources in the manycore fabric. The control coupling within a vector group allows the scalar core to centralize regular, wide memory requests on the vector cores' behalf. Software-defined vectors also naturally support a configurable vector length: applications can choose an ideal hardware vector length according to their regularity. They naturally compose with per-core SIMD by grouping small, fixed-size vector units into larger logical vector engines.

This paper instantiates the software-defined vector abstraction in Rockcress, a processor consisting of minimal modifications on top of an existing RISC-V manycore processor. The key addition in Rockcress is an *instruction forwarding network* (inet): vector cores pass each instruction they execute directly to their neighbors in the vector group. One core fetches an instruction and sends it to its immediate neighbors, those cores then begin executing the instruction and forward it to their neighbors, and so on. Figure 2 shows how Rockcress uses the inet to forward instructions from the first vector core in a group, called the *expander* core, to the other vector cores. Only the expander needs a full fetch stage: the remaining cores disable their instruction cache and fetch logic, instead accepting instructions from the inet.

To keep vector groups busy, Rockcress relies on the independently controlled scalar core to perform shared computations and issue memory requests on behalf of the group. We augment the manycore's scratchpads with cheap bookkeeping to support a decoupled access/execute [26] arrangement.

Contributions. The contributions in this paper encompass the design of a hardware–software abstraction for software-defined vectors (Section 2) and a hardware implementation that augments a simple manycore processor with instruction forwarding to support the abstraction (Section 3). We demonstrate a C-based programming model and compiler that targets the augmented manycore machine to support both MIMD and SIMD parallelism (Section 4).

Relative to manycore execution, Rockcress vector groups offer these efficiency advantages: (1) Performance and energy savings by aggregating word-sized loads into wide vector loads. (2) Energy savings from disabling vector cores' frontends and instruction caches. (3) Performance improvement from overlapping scalar computation and data accesses with vector computation in a decoupled access/execute arrangement. The scheme's overheads consist of the cost to set up vector groups and invoke microthreads. Our evaluation (Section 6) uses cycle-level modeling to investigate whether the benefits consistently outweigh the costs.

On the 15 benchmarks in the PolyBench/GPU suite [10], we find that software-defined vectors improve performance by an average of 1.7× over optimized manycore baselines that exploit memory-level parallelism (MLP). The scheme also reduces the total dynamic energy by 22%. We also compare against a GPU model configured to match the memory and execution resources of the manycore. Rockcress outperforms the GPU by an average of 1.9×.

2 SOFTWARE-DEFINED VECTORS

We describe the software-defined vector ISA from the programmer's perspective. The goal is to let software dynamically form flexibly-sized vector units by reserving portions of a standard manycore machine. The ISA abstracts the concrete hardware implementation in Section 3. There are three main components: forming vector groups (Section 2.1), using them to run microthreads of vectorized computation (2.2), and feeding them with vectorized memory accesses (2.3). We describe the ISA as an extension to RISC-V [34], but the principle applies to any RISC-like machine.

2.1 Vector Groups

The first component of the software-defined vector ISA is the instructions for creating *vector groups*. A vector group is a contiguous region of tiles in the manycore fabric that are logically coupled so that they can run SIMD computations.

Forming a vector group. The programmer can create a vector group from a rectangular region of tiles by instructing each constituent core to enter vector mode. Cores compute a bitmask that describes the vector group configuration, the instruction forwarding path, frontend configuration (e.g., whether the I-cache is enabled), and the group coordinate and size. The scalar core and the first vector core in the group, called the *expander* core, leave their frontends enabled; the other vector cores disable their I-cache. To enter vector mode, cores write the bitmask to a special control/status register (CSR), `vconfig`.

When adjacent cores write to the `vconfig` CSR, a vector group is formed. Each core determines a thread id to distinguish itself from the other cores in the group. The cores then wait for a signal on the inet indicating that their neighboring cores have also entered vector mode and then begin forwarding instructions. The latency to form a group is similar to that of a software barrier; all of the cores in the future group must reach the same configuration instruction.

Disbanding a vector group. Cores need to exit vector mode to return to MIMD execution or to participate in global synchronization. Since cores in vector mode do not maintain a program counter, they need to receive an up-to-date PC and control signal to resume normal execution. The scalar core indicates that the vector group should disband by sending a special `devec` instruction along with a valid PC. When a core receives this signal, the core forwards it to adjacent cores, resets the `vconfig` CSR, and begins normal execution from the given PC.

2.2 Vectorized Computation

Vector groups support a vector-thread execution model [16]. The scalar core uses the `vissue` instruction to launch microthreads on the vector cores:

```
vissue imm
```

The `vissue` instruction includes an instruction pointer indicating the beginning of the microthread. On commit, the scalar core sends the starting PC to the vector group via the inet. Vector threads are *asynchronous:* the `vissue` instruction retires immediately, and the scalar core executes concurrently while the microthread runs.

The expander core is responsible for fetching the instructions in the microthread. It fetches and executes instructions itself, but it also forwards them via the inet to the other cores in the vector group. During microthread execution, all cores in the vector group share a single instruction stream, so control divergence is not possible. Only jumps to consistent addresses (i.e., function calls) are allowed within microthreads. If the application needs divergent control flow, it must disband the vector group and return to MIMD execution.

The microthread ends when the expander core encounters a thread termination instruction `vend`.

Instruction forwarding not only amortizes frontend energy, but also enables efficient synchronization across cores in a vector group (Section 4.2).

2.3 Vectorized Memory Access

A key advantage of vector architectures is their ability to exploit regular memory access patterns. Vector groups need a way to aggregate many small memory demands from all the cores and emit a single, wide request. Our ISA relies on the scalar core to centralize vector memory requests, which entails two main mechanisms: (1) a decoupled access/execute (DAE) scheme to let the scalar core run ahead and feed data to the vector cores, and (2) support for wide load instructions that run on the scalar core.

2.3.1 *Decoupled Access/Execute for Vector Groups.*

Vector groups centralize their regular memory requests on the scalar core to amortize their overhead. To make this work, we adopt a lightweight decoupled access/execute (DAE) [26] scheme that lets the scalar core run ahead and load data to be delivered to the vector cores.

Our scheme reuses the vector cores' local scratchpads as a queue for the incoming data. To simplify the queue's bookkeeping, the architecture organizes the loaded data into chunks of memory called *frames*. Each frame contains the data that a single microthread needs to consume.

Figure 3 shows a logical view of the frame queue. The scalar core issues loads to fill frames, and vector cores consume frames in creation order—but the order that data arrives *within* a given frame

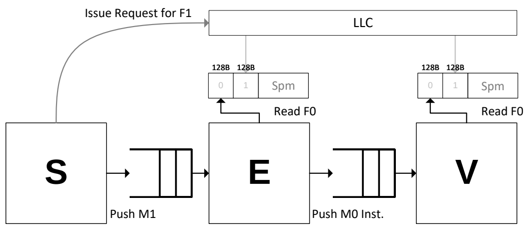


Figure 3: A logical view of frames and microthreads. The vector cores read frame 0 (F0) to execute microthread 0 (M0). Concurrently, the scalar core issues memory requests to fill the next frame (F1) and initiates the corresponding microthread (M1) that will consume this frame.

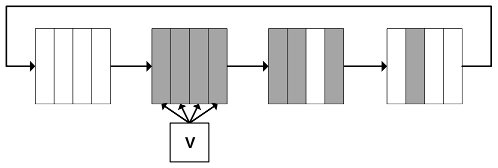


Figure 4: An example state of the frame queue with four frames each containing four words. Older frames are farther left. The first frame is freed and awaiting memory. The vector core is currently accessing the second, full frame. The third and fourth frames are partially full (data is still arriving) and the vector core cannot read them yet.

does not matter. Vector cores maintain a circular buffer of frame-sized regions in their scratchpads. Figure 4 illustrates an example state of this circular buffer: the core can read from the head of the queue while the memory system concurrently fills future frames.

Before forming a vector group, all cores configure their frame size and total number of frames by writing to a CSR. This event allocates a fixed amount of space on the scratchpad to be used as a logical frame queue. The rest of the scratchpad is free to be used for programmer-allocated data and the stack.

Within microthread code, the vector cores consume frames from their scratchpads. The instruction `frame_start` stalls the current microthread until the frame at the head of the queue is ready, i.e., all its data has arrived. Each vector core executes `frame_start` to receive a base offset to the current frame on writeback. The vector core can then freely read and write the frame's region of memory in the scratchpad. When it finishes using the memory, the vector core uses another instruction, `remem`, to free the current frame.

At the C level, using frames on vector cores looks like this:

```c
int frame_ptr = FRAME_START();
int a = spad[frame_ptr + 0];
int b = spad[frame_ptr + 1];
int c = a+b;
REMEM();
```

Our architecture's DAE mechanism is only for the load path—stores do not use it. The primary reason is that loads are on the critical path for vector computations, but stores typically are not, so it is possible to use simple non-blocking store operations that hide memory latency without a DAE setup. Standard non-blocking stores

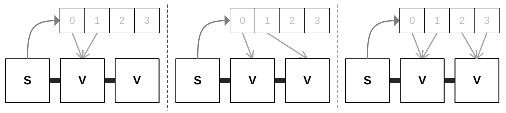


Figure 5: A scalar core sends a request to a cache line and generates multiple responses to the vector cores. (Left) Single load of fetch width 2. (Middle) Group load of fetch width 1. (Right) Group load of fetch width 2.

also avoid the cost for execution units to synchronize and forward output data for coalescing on a central access unit.

2.3.2 Wide Vector Loads. The reason for centralizing a vector group's loads is to coalesce them into *wide accesses* for contiguous chunks of data. We augment our architecture with a simple way to issue wide loads that reuses the existing memory network on the manycore. The key component is a new *vector load* instruction that the scalar core can run on behalf of its vector group. A vector load requests an entire cache line from the LLC and distributes the results to the vector cores in the group, one chunk at a time. We limit the length of a vector load to the cache line size: aligned loads (beginning at offset 0 in the line) request data from a single cache line and unaligned loads request data from *at most* two cache lines.

We add a new instruction for non-blocking vector accesses:

```
vload sp, addr, off, width, var
```

The operands are: (1) the offset in the receiving scratchpads, (2) the source memory address, (3) the core offset in the vector group to receive the first response, (4) the access width per vector core, and (5) the variant (see below). Operands (1), (2), and (3) allow data to be loaded from main memory directly into a scratchpad memory. Operands (4) and (5) configure the style of vector access. We pack these operands into two registers and an immediate in order to fit within a standard RISC-V instruction format.

While `vload` by default only supports aligned cache lines, we also support unaligned loads using pairs of instructions. Two additional variants load a *suffix* of one line and a *prefix* of a second line that combine to form a full line-sized block. The program issues both instructions with the same arguments to achieve an unaligned load.

Vector loads support three variants which instruct the LLC where to send each part of the accessed cache line:

- Single: Sends part of the line to a single vector core.
- Group: Sends consecutive parts of the line to each core in the vector group.
- Self: Sends all data back to the core that sent the request.

Figure 5 illustrates the variants' request and response paths.

We find that the supported variants can map effectively to many patterns, and code that does not fit them can fall back to single-word loads. The work division strategy determines what type of load can be used. Consider this sum example:

```
for (int i = tid; i < N; i += VLEN)
    for (int j = 0; j < M; j += FLEN)
        c[i] += a[i*M+j];
```

Each vector core performs a separate sum across the `j` dimension. The scalar core must issue a separate vector load of size `FLEN` for each vector core.

```
for (int core = 0; core < VLEN; core++)
    vload sp, a[(i+core)*N+j], core, FLEN, SINGLE;
```

However, if the work division strategy was changed, then a more efficient variant could be employed.

```
for (int i = 0; i < N; i++)
    for (int j = tid*FLEN; j < M; j += FLEN*VLEN)
        c_partial[i] += a[i*M+j];
```

Here, the vector cores work on the same sum across `j`. The scalar core can fetch consecutive chunks of `j` to each core to generate a partial sum. In this case, a group load can be used to fetch `FLEN*VLEN` worth of data:

```
vload sp, a[i*N+j], 0, FLEN, GROUP;
```

The added cost is the need for a reduction of the partial sums, but this can be done somewhat cheaply.

In some cases such as a single-level loop nest, any variant can suffice—the choice comes down to performance rather than correctness. Groups loads can be more efficient because they require fewer scalar instructions and memory requests.

2.4 Vector Odds and Ends

Vector groups emulate classical vector machines and can support traditional vector operations, including prediction, gather/scatter, and shuffles.

Predication lets vector machines implement conditional execution without resorting to divergent control flow. Our microthreads support predication using a single mask that consists of a 1-bit flag on each vector core. Two predication instructions set the per-core flag based on a comparison between two registers:

```
pred_eq rs1, rs2
pred_neq rs1, rs2
```

When the flag is 0, a vector core executes all instructions as nops until a subsequent predication instruction sets the flag back to 1. At the C level, a simple condition:

```
if (cond == 1) { c = a + b; }
```

Can be implemented this way:

```
PRED_EQ(cond, 1);
c = a + b;
PRED_EQ(0, 0);
```

By using a single, implicit mask, our ISA's predication feature avoids the need for special predicated instructions, as in other vector ISAs, that would make the core more complex even in MIMD mode. The scheme adds more instruction overhead than a typical predication scheme, but the worst cases are for nested conditionals where the kernel is better suited to standard manycore mode anyway.

Vector groups perform scatters/gathers by performing word-sized memory operations on the vector cores (not the scalar core). The accesses are non-blocking to avoid stalling the entire vector group to wait for each component memory access.

To perform a shuffle, vector cores execute remote stores to other cores' scratchpads (a common feature in manycore architectures). To ensure consistent access to shuffled data, the vector group must synchronize (Section 4.2).

3 THE ROCKCRESS ARCHITECTURE

This section describes a design implementing the software-defined vector ISA from Section 2. The key architectural mechanism is *instruction forwarding:* vector cores share a single instruction stream by passing each instruction directly to one another, hop by hop.

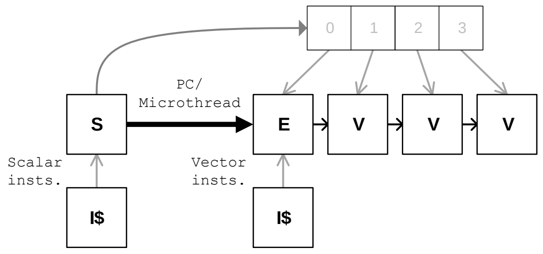


Figure 6: The scalar (S), expander (E), and vector (V) roles in a vector group of length four. The expander is a vector core but also fetches instructions.

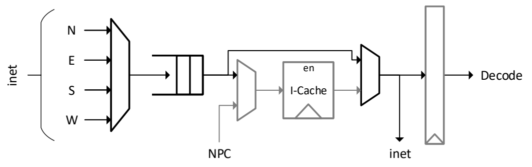


Figure 7: Modifications to the fetch stage to interface with the inet and form vector groups. The black components are additions; gray indicates parts of an ordinary fetch stage.

3.1 A Manycore Baseline

Our extensions start with a minimal set of assumptions about a standard manycore processor. We base our assumptions on a mature open-source design [5], which consists of a tiled grid of simple in-order RISC-V CPUs. Each CPU has a local instruction cache but an explicitly managed scratchpad for data—we do not assume cache coherence. We assume a NoC connecting the cores and the global memory system. The array shares a set of LLCs in front of off-chip DRAM; these caches partition the global address space by striping.

3.2 Instruction Forwarding

We add a systolic instruction forwarding network (inet) that is separate from the manycore's existing data network. Whereas the data network may be a dynamic, packet-switched network, the inet is a simple static network of direct 1-cycle connections between neighboring tiles. Forwarding an instruction consists of a 32-bit register read and write, which consumes significantly less energy than an I-cache hit.

A vector group consists of one scalar core, one expander core, and many vector cores. Figure 6 illustrates an example vector group of length four. Only the scalar core and the expander core need active processor frontends and I-caches; the remaining non-expander vector cores receive instructions from the inet. Energy efficiency scales with vector length as more cores become vector cores.

Figure 7 shows how we modify cores' fetch stage to take advantage of the inet. A vector core receives instructions in its fetch stage via a single inet queue. The queue is driven by a multiplexer that selects between the output of each of the four adjacent cores. A vector core bypasses the I-cache directly to the decode stage, while the expander core uses PCs received from the scalar core to fetch instructions from the I-cache. The instructions sent to the decode stage are also output on the inet to be used by adjacent cores.

The rest of this section describes how instruction forwarding works from the perspective of each of the three roles.

Scalar core. Scalar cores act independently and fetch from the scalar instruction stream. Its instructions include both normal scalar computations and `vissue` instructions, which launch microthreads on the vector cores (Section 2.2). It is not possible to cancel a speculatively launched microthread, so the scalar core sends the launch message after the `vissue` commits. The scalar core can execute all RV-G instructions; it is typically responsible for integer address calculations and memory requests.

Expander core. The expander core can run most RV-G instructions, including direct jumps and function calls. It may also execute conditional branches, but the program must ensure that the branch outcome is the same for the entire vector group. The expander core pauses instruction fetch when it encounters a branch to avoid sending the wrong instructions to other vector cores and resumes fetching when the branch path is resolved later in the pipeline. The expander core does not forward control flow instructions because other vector cores cannot diverge. The microthread finishes when the expander core encounters a `vend` instruction.

The main purpose of the expander core is to enable asynchronous computation. The scalar core need not facilitate instruction fetch for the vector cores, so it can freely run ahead.

Vector core. Vector cores act as vector computation lanes. Their task is to do arithmetic work with low control overhead. They automatically forward every instruction they receive to their downstream vector cores—they never squash instructions. Arithmetic, memory, and predication instructions are allowed in a vector core's instruction stream: control flow is not allowed.

3.3 Scratchpad-Based Decoupled Access

We augment the cores' scratchpads to support the logical DAE queue described in Section 2.3. The key hardware support is metadata that tracks when data is available and ready for consumption. A straightforward but costly approach would be to add per-word full/empty bits to the entire scratchpad. Rockcress opts for a lower-complexity solution that tracks readiness at a frame granularity.

The hardware maintains a set of counters that track the number of words that have arrived in a given frame. Whenever a word arrives to the scratchpad from the data network, the core increments the counter for the frame containing the destination address. When the counter's value equals the configured frame size, the entire frame is ready and the core can begin using the data. When the core frees the frame, the values of each counter are shifted to the left by one and the rightmost count is set to zero. This counter-based mechanism allows data to arrive out of order within a frame while still enforcing in-order consumption of frames themselves.

The number of frame counters in the hardware limits the number of frames that can be *open* (i.e., receiving data) simultaneously. More counters let the DAE scheme run farther ahead and tolerate more variability in memory latency. Our Rockcress implementation has five 10-bit frame counters.

```
for (int i = 0; i < 128; ++i) {
  process(i, a[i]);
}
```

(a) Original DOALL loop.

```
VECTORIZE(get_vector_config(...));
VECTOR_ISSUE({
  int vec_i = 0;
  float a_spad;
});
for (int i = 0; i < 128; i += VECTOR_LENGTH) {
  VECTOR_LOAD(a_spad, &(a[i]), GROUP);
  VECTOR_ISSUE({
    process(vec_i + thread_id, a_spad);
    vec_i += VECTOR_LENGTH;
  });
}
DEVECTORIZE();
```

(b) Simplified Rockcress vectorized code.

Figure 8: Example code for a simple DOALL loop.

3.4 Architecture Support for Wide Accesses

Our baseline processor has a collection of shared LLCs that sit between the manycore array and main memory. We modify these caches to support the wide access instructions described in Section 2.3. The caches need to send multiple chunked responses for a single line-sized request.

We add a counter to each cache, which it uses to serially generate responses for consecutive words in a line. When a wide access hits in the cache, it initializes this counter. Each cycle, the cache generates a response based on the base address plus the current count and then increments the counter. Accesses generate serial responses to a given base core and base scratchpad offset (Core, Offset) as:

$$(\text{Addr} + \text{Cnt}) \rightarrow (\text{BC} + \text{Cnt}/\text{RPC}, \text{BO} + \text{Cnt}\%\text{RPC})$$

where Addr is the memory address, Cnt is the current response count, BC is the base core to respond to, RPC is the responses per core, and BO is the base scratchpad offset.

Caches can generate responses to any rectangular vector group layout, but this layout must be provided by a wide access packet. The scalar core's memory unit generates a wide access packet using the vector group's configuration (`vconfig`) and the in-flight `vload` instruction. `vconfig` provides the base core of the vector group, which is the top-left vector core, and the group's dimensions. This is combined with the number of responses and the offset (from the base core) specified in the `vload`. The unit also determines the proper access widths and offset in the case of unaligned loads.

4 PROGRAMMING WITH SOFTWARE-DEFINED VECTORS

This section covers two programming concerns: compilation and synchronization within vector groups.

4.1 Compiling Vector Microthreads

We use C preprocessor macros and a custom assembly-manipulation pass to provide a programming model for implementing code for Rockcress. The workflow first compiles application C code to RISC-V assembly using stock GCC, and then our custom pass runs on the assembly to produce executable Rockcress code.

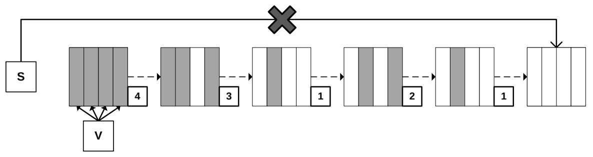


Figure 9: The scalar core cannot fetch words for frames that are ahead of the active frame by more than the number of frame counters.

Figure 8 shows a simple DOALL loop in sequential C and its vectorization for Rockcress. The vectorized code strip-mines the loop to move by `VECTOR_LENGTH` steps. C macros `VECTORIZE` and `DEVECTORIZE` wrap inline assembly to manipulate the `vconfig` CSR (Section 2.1) to create and destroy a vector group. We provide a `VECTOR_LOAD` macro that emits a wide load instruction to copy data from main memory to the vector cores' scratchpads.

Next, the code uses `VECTOR_ISSUE` to delimit microthread code. The compiler splits the program into code that runs on the scalar core and the microthread bodies that run on the vector cores. Critically, the two different contexts maintain separate state. For example, the loop induction variable, `i`, resides on the scalar core, so the vector code maintains its own copy, declared as `vec_i`. The code uses one microthread to initialize the vector state and a second microthread for the loop body, including the increment to `vec_i`.

The Rockcress compiler must generate assembly for individual microthreads while preserving their semantics in the context of the entire program. For example, to generate correct and efficient assembly for the statement `vec_i += VSIZE`, the compiler needs to allocate a register for `vec_i` that persists across invocations of the microthread. It does not suffice to compile each microthread in isolation, such as in a separate function—the compiler must be aware that the body microthread is invoked in a loop, and that the loop follows the setup microthread. Instead, the Rockcress compiler preprocesses the program to extract the microthread bodies into a separate C source file, compiles it separately, and then uses an assembly post-processing pass to merge the microthreads back into the main scalar code.

4.2 Intra-Group Implicit Synchronization

The cores within a vector group occasionally need to synchronize. Most prominently, scalar cores performing decoupled accesses (Section 2.3.1) need to ensure that they do not run too far ahead of vector core execution; as Figure 9 illustrates, the scalar core could overrun the number of hardware counters allocated to the frames. Shuffles must also synchronize to avoid freeing frames that other cores need.

We implement a compiler-driven scheme that *implicitly* synchronizes vector groups by waiting for instructions to propagate through the inet. The key insight is that the inet forms a bounded queue: it is impossible to send instructions into it indefinitely, so slow cores later in the group will cause earlier cores to block. Therefore, a core can only stay a bounded number of instructions "behind" any other core in the vector group. To synchronize, each core can wait until it has seen enough instructions execute such that all cores in the group must have passed a given barrier point.

To perform implicit synchronization, we first need a bound on this *instruction delay:* a value n such that any two instructions in the pipelines of any cores in the vector group may be separated by at most n dynamic instructions. To find this bound, we identify the longest path from any core to any other core in a $m \times m$ vector group and include every source of queueing along the path, including the inet queue itself and the stages of the CPUs' pipelines. We derive this bound:

$$n = (2m-2) \cdot \mathrm{q}_{\mathrm{inet}} + \sum_{i}^{\mathrm{stages}} (\mathrm{buf}_i) + \mathrm{ROB}$$

Here, $2m-2$ is the longest instruction forwarding path in the vector group, $\mathrm{q}_{\mathrm{inet}}$ is the size of the inet queues, buf_i is the length of pipeline buffers in the decode, rename, issue and commit stages, and ROB is the number of entries in each core's reorder buffer. This n bounds the number of instructions that can be accommodated in the buffers before stalling the inet.

Our compiler-driven implicit synchronization scheme implements a vector-group barrier by ensuring that code before and after the barrier is separated by at least n microthread instructions. To prevent excessive scalar-core runahead, for example, we need to ensure that the scalar core does not request too many data frames without issuing microthreads to consume them. We first compute the maximum number of in-flight frames:

$$\mathrm{num\_active\_frames} = \left\lceil \frac{n}{\mathrm{instructions\_per\_frame}} \right\rceil$$

where instructions_per_frame is the length of a `vissue` microthread. Using this value, the compiler can determine how many frames the scalar core can safely run ahead:

$$\mathrm{ahead\_offset} = \mathrm{max\_frames} - (\mathrm{num\_active\_frames} + \mathrm{q}_{\mathrm{inet}})$$

where max_frames is the number of frame counters and $\mathrm{q}_{\mathrm{inet}}$ accounts for the maximum number of queued microthreads between the scalar and expander cores. The compiler uses this bound to ensure that the scalar core does not exhaust the frame counters in the vector cores.

5 EXPERIMENTAL SETUP

We model a baseline manycore machine, the software-defined vector extensions, and a competitive GPU using the gem5 cycle-level simulation infrastructure [6].

5.1 Manycore & Rockcress

For the cycle-level model of Rockcress, we start with a baseline manycore that reflects the assumptions in Section 3.1: the Celerity open-source RISC-V manycore [5, 8]. Our gem5 model uses the Ruby memory system and the Garnet2.0 mesh NoC. Each tile has an I-cache, a scratchpad, and an 8-stage CPU with in-order issue, out-of-order writeback, and in-order commit. At the top and bottom of each mesh column, there is a shared LLC. DRAM is connected to each LLC slice and uses a fixed-latency, fixed-bandwidth model. Table 1a lists the microarchitectural parameters. The SRAM latencies are estimated using CACTI 6.5 [20] assuming a 32 nm process at 1 GHz. We augment the baseline model to support software-defined vectors by modifying the CPUs, scratchpads, and LLCs as described in Section 3.

Table 1: Microarchitectural parameters for the models.

(a) Manycore.

Component	Setting
Cores	64
ALU Latency	1
Multiply Latency	2
Divide Latency	20
FP ALU Latency	3
FP MUL Latency	3
SIMD Width	4 words
SIMD ALU Latency	3
Load Queue Entries	2
inet Queue Entries	2
Cache line Size	64 bytes
I-Cache Capacity	4kB
I-Cache Hit Latency	1 Cycle
I-Cache Ways	2
Spm Capacity	4kB
Spm Hit Latency	2 Cycles
Router Hop Latency	1
On-Chip Net Width	4 words
LLC Capacity	256kB
LLC Banks	16
LLC Hit Latency	1 Cycle
LLC Ways	4
DRAM Latency	60ns
DRAM Bandwidth	16GB/s

(b) APU.

Component	Setting
Compute Units (CUs)	4
Lanes per vALU	16
vALUs per CU	4
vALU Latency	4
Wavefront Size	64
Wavefronts per CU	4
Registers per CU	8192
Cacheline Size	64 bytes
TCP Capacity	16kB
TCP Hit Latency	1 Cycle
TCP Ways	16
TCC Capacity	256kB
TCC Hit Latency	2 Cycles
LLC Capacity	4MB
LLC Hit Latency	2 Cycles
LLC Ways	16
CPU L2 Capacity	512kB
CPU L2 Hit Latency	2 Cycles
CPU L2 Ways	8
CPU L1D Capacity	64kB
CPU L1I Capacity	32kB
DRAM Latency	60ns
DRAM Bandwidth	16GB/s

The LLCs represent disjoint address spaces and thus do not require cache coherence. They are write-back, pseudo-LRU replacement caches with 64-byte lines, which limits wide accesses to 16 words. We also experiment with longer lines to increase the amount of coalescing that can occur.

A vector request can only generate one word response per cycle per port (CPU-side or memory-side). We experiment with various on-chip network widths to increase the number of words that can be sent per cycle to a single core.

We also consider manycore configurations with standard fixed-length SIMD units in each core using the RISC-V vector extension [1]. These configurations, unlike Rockcress, optimize for compute density. Rockcress's extensions can also apply to this baseline, aggregating short per-core SIMD units into wider vector groups.

5.2 Energy Model

We develop a first-order energy model to complement our cycle-level simulation. The model assigns energy costs to simulation statistics, such as memory accesses and instruction executions, and computes a total dynamic energy for a benchmark execution. When a core is in vector mode, it omits the energy costs for fetch and I-cache accesses.

We use CACTI 6.5 [20] to model the access costs of the I-caches, scratchpads, and LLCs. A 4-wide vector load consumes as much energy in the LLC as 4 scalar loads in our model. We model a single I-cache fetch per instruction.

For CPU energy, we use a published breakdown for Ariane [35]. Ariane is a single-issue RISC-V core with in-order issue, out-of-order writeback, and in-order commit, like our gem5 model. It has been used in a manycore [2]. Zaruba and Benini [35] break down Ariane's energy per component, per instruction, per cycle. We use the energy costs as follows:

- The I-cache and D-cache costs are substituted for our modeled I-cache and scratchpad respectively.
- We omit virtual memory (VM) and performance counter (CTS) costs because our architecture lacks them.
- Four different instructions costs (integer ALU, integer MUL, integer DIV, and load/store) are mapped to corresponding statistics from the gem5 simulation.
- Floating-point operations map to costs for integer operations (including FMA, which counts as multiply).
- For MUL and DIV, we scale the multiplier energy cost to the maximum number of cycles the operation takes (2 cycles for multiply and up to 64 cycles for divide).
- Vector instruction costs are estimated by multiplying the functional unit and writeback costs by the vector length. The rest of the instruction cost is left unchanged.

5.3 GPU

We use an existing gem5 APU model [12] to model a GPU. The APU consists of CPUs coupled to a GPU via a shared LLC. The GPU is divided into compute units (CUs) with four vector ALUs (vALUs) each. Each vALU has 16 lanes and executes a 64-thread wavefront every four cycles.

We tune the microarchitectural parameters to make a rough comparison with the manycore. Table 1b lists the model's parameters. The DRAM model is identical to the one in the manycore model. The L2s (called LLC in the manycore and TCC in the GPU) have the same capacity. The L1s (scratchpad in manycore, TCP in GPU) have different sizes but remain that way due to architectural differences. The GPU also has an additional L3 (GPU LLC) that is shared with two CPUs in the system. We model hit latencies using CACTI, assuming a 1 GHz frequency and 32 nm process.

The number of CUs in the GPU is determined by arithmetic intensity per area, which will be higher than a manycore. In Ariane, only 15% of the core area is devoted to the integer arithmetic unit. We roughly estimate that there should be 4× more ALU lanes in the GPU configuration than there are ALUs in the manycore.

Each GPU CU has four wavefronts of 64 threads each. Larger GPU designs typically have more wavefronts to hide memory latency, however, these designs require significant resources and would be unfairly provisioned compared to the manycore.

6 EVALUATION

This section uses our cycle-level models to compare manycore, GPU, and software-defined vector execution efficiency.

6.1 Benchmarks

We evaluate Rockcress on all 15 benchmarks in the PolyBench/GPU suite [10] (Table 2). We compile the C code using GCC 10.1.0 with `-O3` optimization, targeting the uncompressed RV-G ISA and vector extensions. We optimize the benchmarks for each target by unrolling loops using the canonical GCC pragma. We translate the GPU benchmarks from CUDA to HIP and compile using HIPCC.

On the software-defined vector architecture, we identify kernels, form vector groups at the beginning of each kernel, and disband them at the end. The cores use a global barrier between kernels. The typical mapping strategy is to partition a kernel's outer loops among vector groups and inner loops among the cores within the groups. In general, microthreads consist of multiple inner loop iterations to reduce the communication cost between vector and scalar cores. We compare 4- and 16-wide vector groups and create the maximum number of vector groups that fit within 64 cores. We strip-mine the loops in the kernels according to the configured vector length.

We check correctness using a serial version of each kernel. Floating point errors never exceed the thresholds specified in the PolyBench/GPU implementations.

6.2 Configurations

We compare multiple versions of the benchmarks running on the manycore and a separate GPU version (Section 5.3). Table 3 enumerates the naming convention and corresponding features. We consider a basic MIMD baseline (NV), a competitive baseline optimized with non-blocking wide accesses for MLP (NV_PF), and a baseline with narrow per-core SIMD units (PCV_PF). The MLP optimized baselines use the *vload* instruction to fetch full cache lines into their private scratchpads. NV_PF approximates the Celerity manycore [5, 8] which supports non-blocking scalar loads. Vector group lengths can be configured at compile time, so we compare the baselines with the fastest Rockcress configuration (BEST_V). BEST_V includes both V4 and V16 and the ability to choose a larger cache line size. All MLP optimized benchmarks were able to use SIMD extensions with the exception of gramschm. We choose the closest valid configuration in place of them for completeness (PCV without MLP for PCV_PF, V4 for V4_PCV, and V16 for V16_PCV).

In the vector configurations, the vector groups leave some unused cores. In V16, for example, we can create 3 groups of 17 tiles, so we use only 80% of the tiles in total. V4 uses 94% while NV and NV_PF use 100%. While it is possible to use the remaining cores in independent mode or a smaller vector group, our evaluation leaves them idle for simplicity.

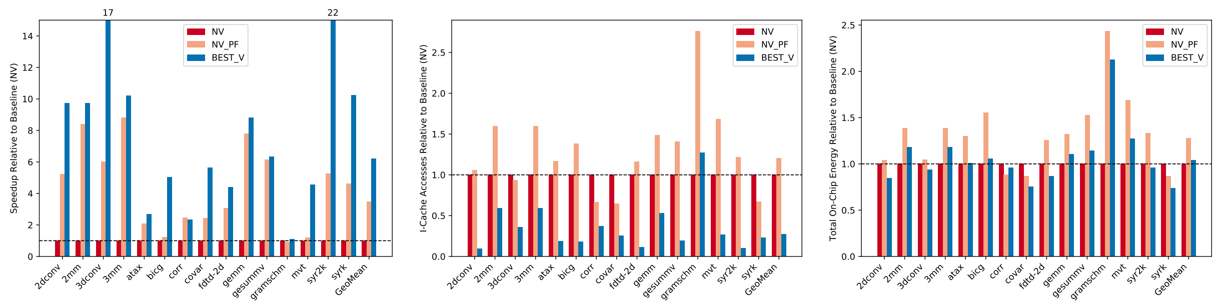
6.3 Performance

Figure 10a shows the speedup over the basic manycore baseline. Both NV_PF and the vector configurations outperform the NV baseline by exploiting MLP. However, the vector configurations are the fastest and outperform the manycore baseline optimized with non-blocking wide loads (NV_PF) by 1.7× on average.

The benefit of software-defined vectors varies by application. For example, 3dconv using 16-wide vector groups outperforms the NV_PF baseline by 2.0×, while 2mm only approximately matches the performance of NV_PF. Along with 3dconv, bicg and mvt perform exceptionally well in a vector configuration, with 4.1× and 3.8× speedup over NV_PF respectively. The only benchmark that did not improve due to decoupled access was gramschm. This benchmark is not able to take advantage of vector loads due to its access pattern and must resort to scalar loads.

Table 2: PolyBench/GPU applications used in the evaluation.

Name	Input	Description	Algorithm opt.	Mem opt.	Kernels
2dconv	2048×2048 image	3x3 filter applied to an image			1
2mm	256×256 matrix	Two matrix multiplies	Tiled Outer Product.	Transpose	2
3dconv	256×256×256 volume	3x3 filter applied to a volume			1
3mm	256×256 matrix	Three matrix multiplies	Tiled Outer product	Transpose	3
atax	2048×2048 mat, 2048 vec	Mat-transpose vec ($y = A^T Ax$)	Loop reordering		2
bicg	2048×2048 mat, 2048 vec	Biconjugate Gradient Method			2
corr	512×512 matrix	Matrix correlation	Kernel fusion	Transpose	2
covar	512×512 matrix	Matrix covariance	Kernel fusion	Transpose	2
fdtd-2d	512×512 mat, 30 tmax	Finite-difference Time-domain			3
gemm	256×256 matrix	Matrix mul. ($C = \alpha AB + \beta C$)	Tiled Outer product	Transpose	1
gesummv	4096×4096 mat, 4096 vec	Matrix vector ($y = \alpha Ax + \beta Bx$)			1
gramschm	320 vectors of length 320	Gram–Schmidt decomposition			3
mvt	4096×4096 mat, 4096 vec	Mat-vec(Ax_1), transpose($A^T x_2$)			1
syr2k	256×256 matrix	Symmetric Rank-2K Update			1
syrk	256×256 matrix	Symmetric Rank-K Update			1



(a) Speedup relative to NV baseline. **(b) I-cache accesses.** **(c) Total on-chip energy.**

Figure 10: Performance, I-cache statistics, and energy results for our benchmarks.

Table 3: Benchmark configurations evaluated.

Config. Name	Group Size	SIMD Words	Wide Access	DAE	Long Lines
NV	1	1			
NV_PF	1	1	×		
PCV_PF	1	4	×		
V4	4	1	×	×	
V16	16	1	×	×	
V4_PCV	4	4	×	×	
V16_PCV	16	4	×	×	
V4_LL_PCV	4	4	×	×	×
V16_LL	16	1	×	×	×
V16_LL_PCV	16	4	×	×	×
BEST_V	4 or 16	1	×	×	?
BEST_V_PCV	4 or 16	4	×	×	?
GPU	1	16			

6.4 Energy

Rockcress saves energy by reducing the total cost of fetching instructions. Figure 10b shows the number of I-cache accesses in each configuration. V4 and V16 reduce I-cache accesses compared to NV by 2.2× and 4.7× respectively. NV_PF increases I-cache accesses over the baseline because it adds instructions to stage data to the scratchpad before moving it to a register. The vector configurations incur the same cost but amortize it over fewer frontends. V4 and V16 reduce I-cache accesses over NV_PF by 2.7× and 5.6×.

Figure 10c shows the resulting changes in total energy according to our energy model. The reductions in fetch costs translate to a 22% reduction in energy versus the NV_PF baseline, approximately matching the NV baseline. Vector configurations are not more energy efficient than NV because the energy the inet saves is offset by the additional energy required by the scratchpad to exploit MLP.

6.5 Scalability

This section evaluates the viability of the NV_PF baseline and highlights its main bottlenecks.

Baseline Scalability. We evaluate the scalability of our baseline manycore system (NV_PF). Figure 11 shows the speedup for an increasing number of cores over a single core (with the same memory system capacity and bandwidth). The scability of 2mm, 3mm, and gemm increases linearly, while the majority of benchmarks are sub-linear after 16 cores.

Figure 12 details the core performance for different manycore sizes using a CPI stack.[1] Benchmarks that do not scale well see a significant increase in stalls due to memory past 16 cores. At larger core counts, the majority of stalls are waiting on loads (labeled *frame stall* in the figure).

[1]Each stacked bar in the CPI stack shows the relative number of cycles where an event occurs in a core's issue stage (normalized to cycles where an instruction was issued). Each total stacked bar height indicates the actual CPI of the core.

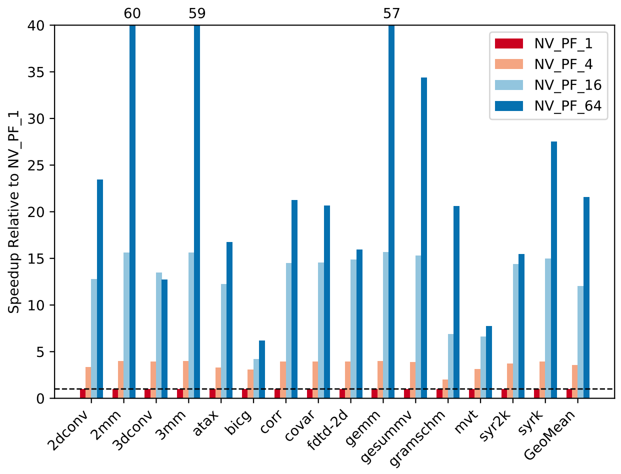


Figure 11: The relative speedup for an increasing number of cores compared to a single core processor.

DRAM bandwidth. DRAM bandwidth is the main bottleneck for larger core counts. The effective DRAM bandwidth per core decreases from 16 GB/s/core at one core to 0.25 GB/s/core at 64 cores. While this is enough bandwidth for benchmarks with high reuse, such as 2mm, 3mm, and gemm, it bottlenecks others.

A recent iteration of Celerity [5] dedicates two HBM2 channels per 128 cores which, yields 0.5 GB/s/core (double our system's bandwidth). We experiment with increasing the memory bandwidth for the 64-core baseline and compare it to using a vector configuration. Figure 13 shows the effects on memory performance. The additional memory bandwidth provides a linear performance improvement for most benchmarks, indicating that they are bottlenecked by DRAM bandwidth. However, we find that using a vector configuration can improve memory performance with no additional DRAM bandwidth. Benchmarks such as 2dconv, 3dconv, covar, syr2k, and syrk see better overall performance with a vector configuration than with additional DRAM bandwidth. Other benchmarks such as atax, corr, fdtd-2d, and gesummv see a reduction in memory stalls, but the overhead from using the inet overshadows this improvement overall. Section 6.6 shows how our DAE system and group wide accesses improve memory performance.

Software-defined vectors provide a way for DRAM-bottlenecked manycore architectures to make better use of memory bandwidth. For applications with regular parallelism, then, the technique facilitates scalability to larger core counts.

6.6 Characterization

Wide access performance. Wide accesses can improve the cache hit rate and better utilize DRAM bandwidth. Figure 17a shows the cache miss rate for various configurations. Vector groups have a better hit rate than NV_PF for two reasons. First, loads can be coalesced across a vector group. Three benchmarks, atax, bicg, and mvt, use group loads where NV_PF cannot. The second reason is that fewer requests are sent per cycle in V4 because there are fewer scalar cores than independent cores in NV_PF.

Both bicg and mvt see the largest reductions in LLC misses. These benchmarks access the critical matrix column-wise which results in poor cache line utilization. Grouped loads are able to extract spatial locality across cores for these types of loads.

DAE performance. DAE reduces the effective memory latency for vector cores, but we find that it does not completely eliminate memory stalls. Figure 15c shows the number of stalls in vector cores due to waiting for a frame to fill for V4 and NV_PF. Ideally, every vector core would never stall as is the case for 2mm, 3mm, and gemm. V4 reduces frame stalls by about 2× over NV_PF. This plot does not show the absolute stall reduction, but instead is normalized to each configuration's run time. For example, bicg V4 is 2.3× faster than NV_PF and has about the same ratio in the plot. However, V4 actually has about 2.7× fewer absolute stalls.

Per-core hardware prefetchers could achieve similar speedups to our DAE techniques. However, software DAE offers flexibility: fixed hardware may be over- or under-utilized for a given kernel. Our DAE system's overhead is also amortized by having only one scalar core (access core) for multiple vector cores (execute cores).

Hardware vector units and GPU. We compare Rockcress to traditional, fixed-length SIMD units in each core. Figure 14a shows the performance impact of adding SIMD units to both the baseline (PCV_PF) and Rockcress (BEST_V_PCV), along with the GPU.

Versus a GPU, Rockcress achieves 1.9× speedup on average. Benchmarks with high arithmetic intensity perform well on the GPU, as expected. These include 2mm, 3mm, gemm, 2dconv, and 3dconv. However, most benchmarks are memory-bound and are slower on the GPU. GPUs rely on massive multithreading and register files to hide memory latency; a larger GPU design would perform better on memory-bound benchmarks. We compare to an under-provisioned GPU design to highlight the area efficiency of Rockcress's DAE mechanism. Software-based DAE is more efficient at hiding memory latency because it only requires one additional run-ahead thread.

Narrow SIMD units do not improve performance in most cases. The problem is that manycores already have high compute density, and the bottleneck is memory bandwidth—so adding SIMD units does not help. As with the GPU, compute-bound kernels such as 2mm, 3mm, and gemm see some benefit. However, most kernels are memory bound and SIMD units exacerbate the issue.

We also experiment with adding SIMD to Rockcress vector groups, so that RISC-V vector instructions are forwarded on the inet. This design point adjusts the vector length by a coarser amount per added core. SIMD units composed in vector groups also have a negligible impact on performance.

Figures 14b and 14c compare the I-cache accesses and energy consumption of SIMD units and vector groups. All configurations significantly reduce the number of I-cache accesses, but the energy reduction per vector length is superior for SIMD units. Vector groups only amortize fetch energy whereas SIMD instructions amortize all of the pipeline energy besides the functional units and register file. SIMD units composed within vector groups do not significantly impact the total energy consumption because it is already well amortized.

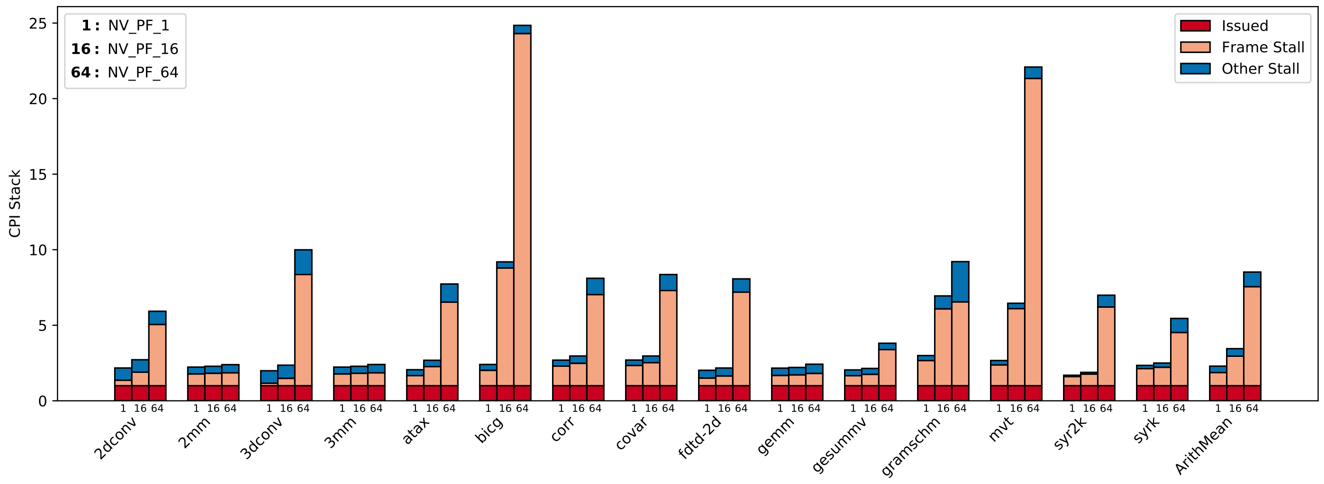


Figure 12: A CPI stack showing the core pipeline stalls for various manycore sizes and benchmarks. As the number of cores increases, most benchmarks are dominated by frame/memory stalls.

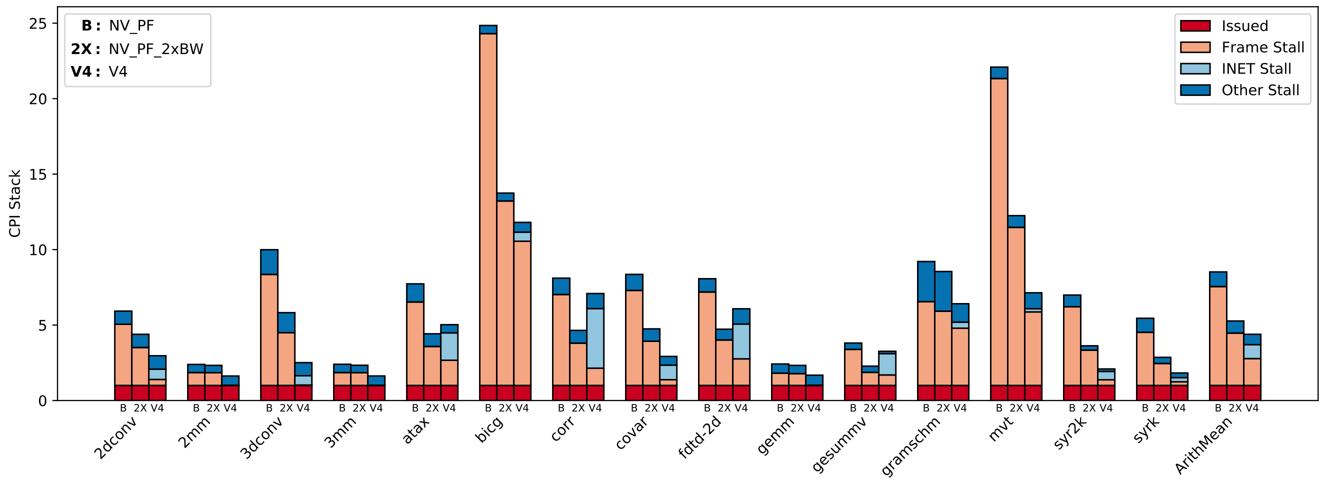
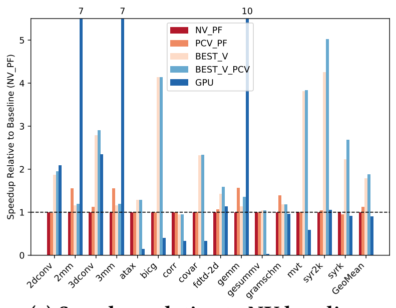
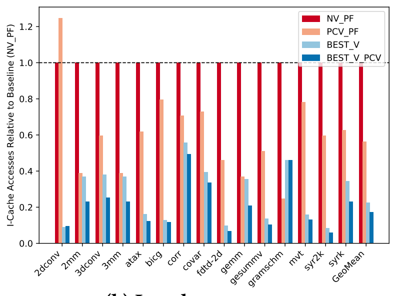


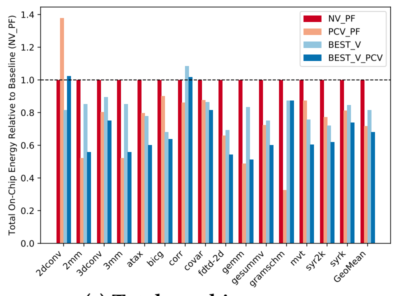
Figure 13: A CPI stack showing the core pipeline stalls for the baseline (NV_PF), the baseline with 2× the DRAM bandwidth (NV_PF_2xBW), and vector groups of size four (V4). V4 with 16GB/s of DRAM bandwidth outperforms the baseline with 32GB/s of DRAM bandwidth due to better utilization of the existing memory bandwidth. The vector configurations only average the events from the expander cores because the root cause of a stall is not apparent in a non-expander vector core.



(a) Speedup relative to NV baseline.

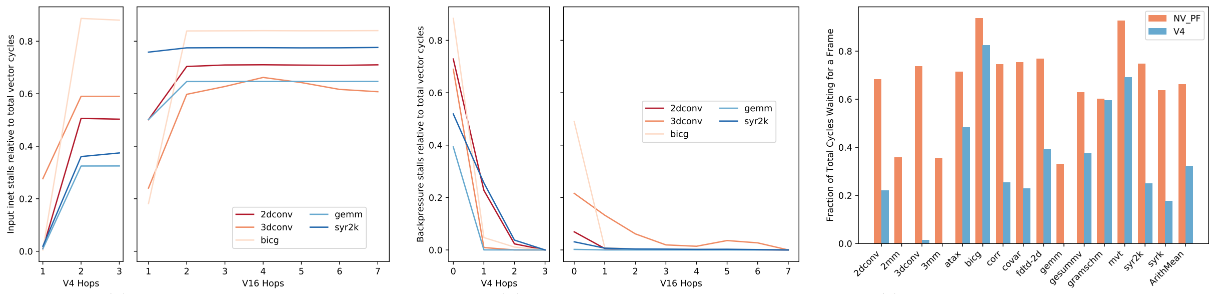


(b) I-cache accesses.



(c) Total on-chip energy.

Figure 14: Performance, I-cache statistics, and energy results for our benchmarks with SIMD units.



(a) Input inet stalls. **(b) Backpressure stalls.** **(c) Cycles waiting for frames.**

Figure 15: Characterization of vector groups. Hops are the traveled inet distance from the scalar core (hop 0 is the scalar core).

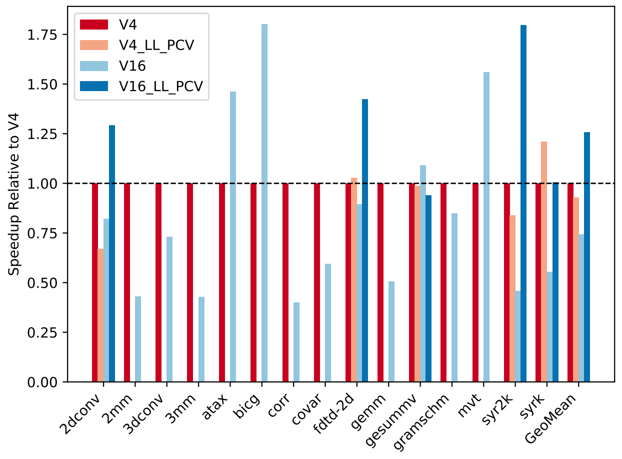


Figure 16: Speedup for various vector group configurations.

Vector length flexibility. Rockcress can adapt to different optimal vector widths for different applications. The average speedup is 1.1× for V16 alone and 1.5× for V4, but by allowing benchmarks to choose the best among the two options, the mean rises to 1.7×.

Increasing the vector length can help amortize frontend costs (cf Figure 10b), but it can also make performance worse. There are three main reasons: (1) fewer cores are active in our V16 configuration (see Section 6.1), (2) the scalar core falls behind and becomes a bottleneck, (3) there is a longer forwarding network, which could create more opportunities for inet stalling and backpressure. To investigate these possibilities, Figure 15a compares the number of cycles the inet input buffer is empty between V4 and V16. The scalar bottleneck is worse in V16, as indicated by the number of inet stalls at hop 1 (the expander core). However, the trend plateaus after two hops, which suggests that all of the inet stalls are generated by the expander core pipeline and these persist through the entire forwarding network. Thus, the scalar bottleneck is the problem rather than the longer forwarding network: The scalar core has to fetch the data to more cores and cannot keep up with the vector core demand.

Figure 15b compares the number of cycles forwarding cores are stalled due to backpressure on the inet. V4 configurations have more backpressure than V16 because microthreads are launched at a slower rate in V16 (scalar bottleneck) and V16 has more buffer space within the group (Section 4.2).

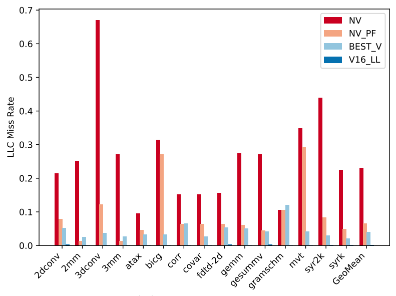
The best vector configuration is V16 for atax, bicg, and mvt; it outperforms V4 by 1.5×, 1.8×, and 1.6× respectively. They perform better at V16 because they use group loads. The number of group loads does not increase as vector groups grow (unlike single core loads), so amortization is free.

Long cache lines. We experiment with increasing the cache line size to allow for longer vector loads and more request amortization. Without any changes to the existing algorithms, the performance would decrease due to cache thrashing. We modify five benchmarks: 2dconv, fdtd-2d, gesummv, syr2k, and syrk with wider loads to take advantage of the longer lines. Group vector loads must be used because larger lines will not fit into a single core's scratchpad. Thus, longer lines are exclusive to vector groups and not as practical for independent cores. We envision a system with reconfigurable cache line sizes [32] to realize long lines for amenable kernels.

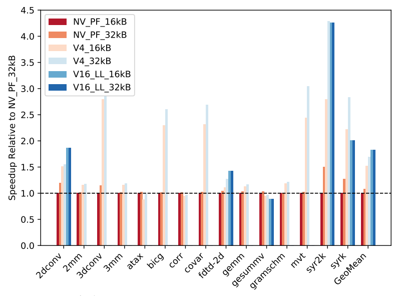
Figure 16 shows the performance of long lines using a cache line size of 1024 bytes. Long lines can minimize the scalar bottleneck because fewer load instructions are needed per microthread. Long lines also improves the cache hit rate as shown in Figure 17a. However, the hit rate was already high with the baseline configurations, so the benefits may be limited in this setup. The benchmarks see modest overall improvement over the non-long-line configurations.

Memory system. We evaluate our system's sensitivity to the on-chip network width and LLC capacity. Figures 17b and 17c shows the impact of varying the cache size and network width for various software configurations. Certain benchmarks like syr2k and syrk are very sensitive to the cache size and network width, while most others show little performance improvement. Increasing the cache size for syr2k has a similar effect to using long lines due to the reduction in miss rate. In general, the on-chip network width is not critical to the performance of vector loads and Rockcress could feasibly be designed with a single word-wide network.

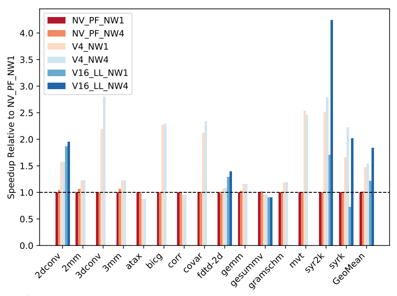
Irregular algorithms. As an example of an irregular application that would waste a standard vector machine, we measure bfs: a breadth-first graph search. A pure manycore (NV) version of bfs is 2.9× faster than either vector version (V4, V16). In Rockcress, a single machine can efficiently execute both regular, vector workloads and irregular, graph-like workloads such as `bfs` via run-time configuration.



(a) LLC stalls.



(b) Speedup: LLC capacity.



(c) Speedup: On-chip network width.

Figure 17: Memory performance.

7 RELATED WORK

Two recent vector ISAs, the RISC-V vector extension [1] and ARM's scalable vector extension (SVE) [27], let programs stay agnostic to the hardware's vector length. However, this flexibility is only in the abstraction: the hardware does not change the compute resources it dedicates to a vector engine. We see it as future work to support such a traditional vector ISA using Rockcress's instruction forwarding approach.

Other architectures can dynamically compose multiple small compute engines into one larger machine. Several efforts reconfigure a multicore processor to trade-off single-thread performance with thread-level parallelism (TLP) [11, 13, 15, 30, 36]. These designs do not target SIMD parallelism, and they require tight coupling between the coalesced components. Software-defined vectors need less invasive modifications to cores. The closest work to ours tightly couples a small group of cores to allow a master core to multicast instructions and scalar work [3]. They do no exploit MLP like in our vector DAE scheme and the tight coupling limits the scalability of large vector lengths. The Cray X1 [9] consists of two-wide vector units that can be optionally coalesced in groups of four to form a single eight-wide vector engine. Rockcress achieves a similar effect but permits a more flexible range of vector widths while also offering a completely independent manycore mode. Libra [22] reconfigures a group of PEs to execute SIMD or VLIW instructions but does not support TLP. Flexible vector lengths have also been proposed for GPUs [14, 17, 23]. The amount of flexibility between MIMD and SIMD is limited due to the centralized controllers and data paths inherent in GPU architectures.

Vector-thread architectures [16, 18] enable a somewhat flexible vector length where each thread can operate independently in MIMD mode or in lockstep SIMD mode to amortize control overhead. Rockcress realizes classic vector-thread architectures as a flexible overlay over a standard manycore machine and provides additional vector length flexibility.

Rockcress's memory system adapts ideas from decoupled access-execute architectures [4, 26] and runahead schemes that use regular cores to implement DAE [7, 28] and applies them to the software-defined vector setting.

8 CONCLUSION

Software-defined vector architectures aim to compete with GPUs for general-purpose parallel programming. By combining a simple MIMD mode and a flexible SIMD mode on the same silicon substrate, architectures can avoid complex hardware thread schedulers and rely on software to choose its own parallelism strategy.

ACKNOWLEDGMENTS

We thank Khalid Al-Hawaj for the RISC-V vector extension implementation and Tuan Ta for his CPU and scratchpad models, which are an integral part of the Rockcress gem5 simulator. We also thank Christopher Batten, Khalid Al-Hawaj, and Zhiru Zhang for their critical feedback on the project in earlier stages. We thank the entire University of Washington/Cornell SDH team for feedback during the project's development.

Support for this research included NSF awards #1845952 and #1723715. This work is also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This work was also supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact describes the environment and experiments required to reproduce our published results.

We include the following materials:

- The architectural model.
- The benchmarks.
- Simulation and data extraction scripts.
- Compilation stack.
- Instructions for how to reproduce our simulation environment in a Docker container or natively.

Using the artifact, you can compile RISC-V benchmarks, run them on our simulator, and analyze the resulting data. We have provided a script to streamline the generation of Figure 10, the key results plot in the main paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Cycle-level simulator.
- **Program:** gem5 [6].
- **Data set:** Polybench/GPU [10].
- **Run-time environment:** Docker or Unix-based system.
- **Hardware:** Recommend at least 4 cores.
- **Metrics:** Simulator event counts (cycles, cache accesses, etc.).
- **Output:** Plots showing the main results of the paper.
- **Experiments:** A subset or all of the data required for the headline result (Figure 10).
- **How much disk space required (approximately)?:** 15GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 50–300 CPU hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD.
- **Workflow framework used?:** Python.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5149289

A.3 Description

This artifact contains the cycle-level architecture model, benchmarks, and scripts used to evaluate Rockcress.

A.3.1 How to access. Our artifact is available as an open-source code repository hosted on GitHub:

https://github.com/cucapra/gem5-mesh

An external RISC-V cross-compiler is needed to compile the provided benchmarks. The compiler can be obtained via an open source repository:

https://github.com/riscv/riscv-gnu-toolchain

A.3.2 Hardware dependencies. No special hardware is required to run the simulator. However, low core-count CPUs will not be able to run all simulations in a reasonable amount of time. A system with at least 4 cores is required, but higher core-count systems are recommended.

A.4 Installation

The installation entails building the gem5 simulator and RISC-V cross-compiler. The setup instructions are outlined in the top-level README. We also provide a Docker container with all the needed packages and pre-built RISC-V compiler.

A.5 Experiment workflow

This artifact reproduces the key results (Figure 10) from the paper. Each data point consists of a benchmark, software setting (i.e., compilation flags like vector length), and hardware setting. The following steps are executed per data point:

- Compile benchmark with software configuration.
- Simulate the binary using gem5 with hardware configuration.
- Extract simulation statistics.

We provide Python scripts to automate the data generation process. Each data point can be simulated in parallel; however, binaries are first compiled serially to avoid conflicts between different software settings. It takes approximately 300 CPU hours (hours on a single CPU) to generate the key results.

For smaller systems, we provide data generation for subsets of the configurations and benchmarks used in the key results. The top-level artifact evaluation script offers the following simulation options:

- small: 50 CPU hours (recommended for 4-core system).
- medium: 150 CPU hours (recommended for 16-core system).
- large: 300 CPU hours (recommended for 32-core system).

A.6 Evaluation and expected results

The artifact evaluation script will produce a speedup, icache, and energy comparison plot. Each experiment size will generate the plots with different data. The large evaluation size will completely reproduce Figure 10 in the paper. The smaller experiments (small and medium) will compare the baseline to one of the optimized configurations. Note that one series in Figure 10 (BEST_V) chooses the best optimized configuration, but the smaller experiments do not simulate all of these potential configurations.

A.7 Experiment customization

We provide a JSON interface to describe experiments. A user can choose software, hardware, and benchmarks to test. Examples are shown in:

https://github.com/cucapra/gem5-mesh/blob/scalar/scripts-phil/eval/experiments/full.json

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Hanna Kruppe, Yunsup Lee, Guy Lemieux, Filip Moc, Rich

Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, and Jim Wilson. 2020. RISC-V “V” Vector Extension, version 0.9. https://github.com/riscv/riscv-v-spec.

[2] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. 2020. BYOC: A “Bring Your Own Core” Framework for Heterogeneous-ISA Research. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[3] D. Bates, A. Bradbury, A. Koltes, and R. Mullins. 2015. Exploiting tightly-coupled cores. In *Journal of Signal Processing Systems*, Vol. 80. 103–120.

[4] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanović. 2004. Cache Refill/Access Decoupling for Vector Machines. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[5] Bespoke Silicon Group. [n.d.]. HammerBlade. https://github.com/bespoke-silicon-group/bsg_bladerunner.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (May 2011).

[7] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. 2001. Speculative precomputation on chip multiprocessors. In *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*.

[8] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael Bedford Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41.

[9] Thomas H. Dunigan, Jeffrey S. Vetter, James B. White, and Patrick H. Worley. 2005. Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro* 25, 1 (2005), 30–40.

[10] Scott Grauer-Gray and Louis-Noël Pouchet. 2012. PolyBench/GPU: Implementation of PolyBench codes for GPU processing. URL: http://www.cs.ucla.edu/pouchet/software/polybench.

[11] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. 2010. Erasing Core Boundaries for Robust and Configurable Performance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[12] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 608–619.

[13] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*.

[14] Xingxing Jin, Brian Daku, and Seok-Bum Ko. 2014. Improved GPU SIMD control flow efficiency via hybrid warp size mechanism. *Microprocessors and Microsystems* 38 (2014), 717–729.

[15] Changkyu Kim, Simha Sethumadhavan, Madhu Saravana Sibi Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. 2007. Composable Lightweight Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[16] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. 2004. The vector-thread architecture. In *International Symposium on Computer Architecture (ISCA)*.

[17] Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. 2012. Dynamic warp resizing: Analysis and benefits in high-performance SIMT. In *IEEE International Conference on Computer Design (ICCD)*.

[18] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *International Symposium on Computer Architecture (ISCA)*.

[19] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. 2015. *The Hwacha vector-fetch architecture manual version 3.8.1*. Technical Report UCB/EECS-2015-262. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.html

[20] Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. [n.d.]. CACTI 6.5. https://github.com/HewlettPackard/cacti.

[21] R Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and S. W. Keckler. 2001. A design space evaluation of grid processor architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[22] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. 2012. Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 84–95.

[23] Timothy G. Rogers, Daniel R. Johnson, Mike O’Connor, and Stephen W. Keckler. 2015. A Variable Warp Size Architecture. In *International Symposium on Computer Architecture (ISCA)*.

[24] Richard M. Russell. 1978. The CRAY-1 Computer System. *Commun. ACM* 21, 1 (Jan. 1978), 63–72.

[25] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-Core X86 Architecture for Visual Computing. *ACM Transactions on Graphics* 27, 3 (Aug. 2008), 1–15.

[26] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *International Symposium on Computer Architecture (ISCA)*.

[27] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (March 2017), 26–39.

[28] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices* (2000).

[29] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. 2018. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. In *International Symposium on Computer Architecture (ISCA)*.

[30] David Tarjan, Michael Boyer, and Kevin Skadron. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Design Automation Conference (DAC)*.

[31] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae Won Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22 (2002), 25–35.

[32] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. 1999. Adapting Cache Line Size to Application Behavior. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) *(ICS ’99)*. Association for Computing Machinery, New York, NY, USA, 145–154. https://doi.org/10.1145/305138.305188

[33] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. *High-Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures*. Springer.

[34] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2011. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Technical Report UCB/EECS-2011-62. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[35] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640.

[36] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. 2007. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *International Symposium on High-Performance Computer Architecture (HPCA)*.