Static Stages for Heterogeneous Programming

ADRIAN SAMPSON, Cornell University, USA
KATHRYN S MCKINLEY, Google, USA
TODD MYTKOWICZ, Microsoft Research, USA

Heterogeneous hardware is central to modern advances in performance and efficiency. Mainstream programming models for heterogeneous architectures, however, sacrifice safety and expressiveness in favor of low-level control over performance details. The interfaces between hardware units consist of verbose, unsafe APIs; hardware-specific languages make it difficult to move code between units; and brittle preprocessor macros complicate the task of specializing general code for efficient accelerated execution. We propose a unified low-level programming model for heterogeneous systems that offers control over performance, safe communication constructs, cross-device code portability, and hygienic metaprogramming for specialization. The language extends constructs from *multi-stage programming* to separate code for different hardware units, to communicate between them, and to express compile-time code optimization. We introduce *static staging*, a different take on multi-stage programming that lets the compiler generate all code and communication constructs ahead of time.

To demonstrate our approach, we use static staging to implement BraidGL, a real-time graphics programming language for CPU–GPU systems. Current real-time graphics software in OpenGL uses stringly-typed APIs for communication and unsafe preprocessing to generate specialized GPU code variants. In BraidGL, programmers instead write hybrid CPU–GPU software in a unified language. The compiler statically generates target-specific code and guarantees safe communication between the CPU and the graphics pipeline stages. Example scenes demonstrate the language's productivity advantages: BraidGL eliminates the safety and expressiveness pitfalls of OpenGL and makes common specialization techniques easy to apply. The case study demonstrates how static staging can express core placement and specialization in general heterogeneous programming.

CCS Concepts: • **Computing methodologies → Graphics processors**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: Multi-stage programming, heterogeneous programming, graphics programming, OpenGL

ACM Reference Format:
Adrian Sampson, Kathryn S McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 71 (October 2017), 27 pages.
https://doi.org/10.1145/3133895

Authors' addresses: A. Sampson, Department of Computer Science, Gates Hall, Cornell University, Ithaca, NY 14853, US; K. McKinley, Google, 1600 Amphitheatre Parkway, Mountain View CA 94043, US; T. Mytkowicz, Microsoft Research, 1 Microsoft Way, Redmond, WA 98052, US.

2475-1421/2017/10-ART71
https://doi.org/10.1145/3133895

1 INTRODUCTION

Heterogeneous computer systems are ubiquitous. Smartphone SoCs integrate dozens of disparate units; mainstream laptop and desktop chips include CPU and GPU cores on a single die; and modern datacenters include GPUs, FPGAs, and special-purpose ASICs [Hauswald et al. 2015; Jouppi et al. 2017; Putnam et al. 2014]. Our work seeks to improve *heterogeneous programming*, in which a single software artifact spans multiple computation units and memories with different resources, capabilities, ISAs, and performance characteristics.

Current techniques for heterogeneous programming tend toward two extremes. On one hand, low-level APIs such as CUDA [Nickolls et al. 2008] and OpenGL [Segal and Akeley 2016] offer precise control over when and where code executes, but their close adherence to hardware comes at the cost of safety and expressiveness. On the other hand, high-level abstractions that hide distinctions between hardware units, such as autotuning approaches [Ansel et al. 2009; Luk et al. 2009; Phothilimthana et al. 2013] and domain-specific languages [Chafi et al. 2011; Ragan-Kelley et al. 2013], sacrifice low-level control over cross-unit coordination.

This paper seeks to combine safety and expressiveness with control and performance. We propose a new abstraction for heterogeneous programming that preserves direct control over hardware resources. We identify two fundamental concepts in heterogeneous programming: placement and specialization. *Placement* controls when and where code runs. Different units in heterogeneous systems have different capabilities and performance characteristics, so programmers need to control which code runs on which unit and how the components communicate. *Specialization* refines general code for efficient execution on specific hardware or inputs. Specialized hardware accelerates specific computational patterns, so applications need to specialize code to match the hardware's strengths.

Static staging for heterogeneity. We describe Braid, a language and compiler that provide efficient abstractions for placement and specialization. Braid provides a unified language with explicit *hardware targets* to place code onto each architecture in the system. Braid's type system enforces safe communication and correct code specialization for each target.

The key idea is to generalize work on *multi-stage programming* [Taha and Sheard 1997] to address both placement and specialization. Staging offers a foundation for safe interoperation between program components, but traditional staged languages focus on dynamic code generation and domain-specific optimization [Brown et al. 2011; DeVito et al. 2013; Rompf and Odersky 2010; Rompf et al. 2014]. Our compiler uses a new approach, *static staging*, to emit hardware-specific code and communication constructs ahead of time. Static staging lets Braid exploit classic staging's expressiveness without its run-time cost.

In addition to place-specific stages, programs can use a *compile-time* stage to express specialization and to explore hardware-specific optimization strategies. Because both concepts rest on a consistent language foundation, specialization *composes* with placement in Braid: applications can use metaprogramming to decide where code should run. Our key finding is that multi-stage programming, with new restrictions and extensions, can serve as a unifying foundation for expressive, high-performance heterogeneous programming languages.

Braid's philosophy is a counterpoint to domain-specific languages that target exotic hardware [Chafi et al. 2011; Ragan-Kelley et al. 2013], where the goal is to hide low-level execution details from programmers. Instead, Braid exposes hardware details and makes them safe. Whereas DSLs are appropriate for domain experts, Braid offers *system* experts direct control over heterogeneous hardware that higher-level abstractions lack. Performance is a leaky abstraction, and hardware details often find themselves embedded into algorithm specifications despite efforts to the contrary [Mytkowicz and Schulte 2014]. Google's TPU [Jouppi et al. 2017], Microsoft's Catapult [Putnam et al. 2014], and other bespoke accelerators are ascendant, and system designers

will need to cope with its inevitable impact on software. Languages based on static staging can expose these hardware-level execution details without resorting to brittle, C-based APIs.

Case study in real-time graphics. We demonstrate our approach in the domain of real-time graphics applications, such as video games, which arguably represent today's most widespread form of heterogeneous programming. Graphics software consists of code running on the CPU and in each stage of the GPU pipeline. Current APIs [Microsoft 2008; Segal and Akeley 2016] use separate languages in each context and unsafe interfaces between them. These applications also specialize code extensively for performance: modern games can produce hundreds of thousands of GPU program variants that customize generic visual effects [He et al. 2016, 2015]. To produce these variants, programmers must use rudimentary token-stream manipulation à la C preprocessor.

We implement BraidGL, a real-time graphics programming system for hybrid CPU–GPUs, and show how it addresses these problems. The compiler takes a single source program with staging annotations and emits a mixture of JavaScript for the CPU, GLSL for the GPU, and WebGL "glue code" [Jackson and Gilbert 2017]. Unlike traditional graphics APIs such as OpenGL and Direct3D, BraidGL programs statically guarantee safe interaction: the communicating programs agree on the types of the values they exchange. In contrast to non-graphics GP-GPU languages such as CUDA and OpenACC [Nickolls et al. 2008; OpenACC 2015], BraidGL explicitly exposes the telescoping sequence of stages that make up the 3D rendering pipeline. Unlike any other heterogeneous programming language we are aware of, BraidGL delivers safe compile-time and run-time metaprogramming tools that compose with staged execution. BraidGL's combination of compile-time safety and low-level control fills a void left empty by both high-level tools and C-based APIs of last resort.

We use three case studies to demonstrate that BraidGL combines performance, safety, and expressiveness. We show how to use static staging to explore specialization strategies that improve frame rates without adding complexity. To demonstrate its safety, we define a formal semantics for a core of Braid and state a type preservation theorem. To demonstrate expressiveness, we show how BraidGL eliminates error-prone boilerplate code for placement and eases the expression of algorithmic specialization. We posit that GPU-accelerated graphics programming poses challenges that are similar to those in other kinds of heterogeneous systems, which suggests static staging is general enough to meet programmer needs for safety and performance in an age of heterogeneity.

We built a live-coding environment to encourage experimentation with Braid and BraidGL. The in-browser environment is available online along with the compiler's source code [Sampson 2017].

2 OVERVIEW BY EXAMPLE

To motivate the problem, this section illustrates the challenges of programming heterogeneous systems using WebGL [Jackson and Gilbert 2017], the JavaScript variant of OpenGL. We then contrast current practices with Braid.

2.1 An Example in WebGL

Real-time graphics software relies on high-performance GPU hardware to achieve interactive frame rates. To draw each frame, the CPU loads scene data into the GPU's memory and issues commands to instruct the GPU to render individual objects. The rendering pipeline assembles a mesh of vertices in 3D space that define an object's shape, positions primitive polygons to cover the mesh, and computes a color for every visible point on every triangle. Modern GPUs are massively data-parallel programmable processors, and custom software controls how each step in the pipeline works to produce a visual effect.

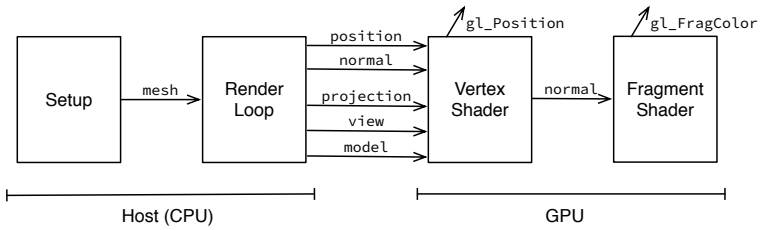


Fig. 1. Stages in a simple graphics program. Each box is a stage; each arrow is a communicated value. The `position` and `normal` vector arrays describe a mesh. `projection`, `view`, and `model` are transformation matrices. The shader stages also send `gl_Position` and `gl_FragColor` vectors to the rendering system.

```
③ var VERTEX_SHADER =
    "attribute vec3 aPosition, aNormal;" +
    "uniform mat4 uProjection, uModel, uView;" +
    "varying vec3 vNormal;" +
    "void main() {" +
      "vNormal = aNormal;" +
      "gl_Position =" +
        "uProjection * uView * uModel *" +
        "aPosition;" +
    "}";

④ var FRAGMENT_SHADER =
    "varying vec3 vNormal;" +
    "void main() {" +
      "gl_FragColor = abs(vNormal);" +
    "}";

  // One-time setup.
① var program =
    compile(gl, VERTEX_SHADER, FRAGMENT_SHADER);
  var loc_aPosition =
    gl.getAttribLocation(program, "aPosition");
  /* ... */
  var mesh = loadModel();

  // Per-frame render loop.
  while (1) {
    // Bind the shader program and its parameters.
②   gl.useProgram(program);
    bindAttr(gl, loc_aPosition, getPositions(mesh));
    /* ... */
    draw(gl, getSize(mesh));
  }
```

(a) Tiny rendering program in JavaScript and GLSL.

```
① var mesh = loadModel();

  # Render loop.
② render js<
    var position = getPositions(mesh);
    # ...

    # Vertex shader.
③   vertex glsl<
      gl_Position = projection * view *
        model * position;

      # Fragment shader.
④     fragment glsl<
        gl_FragColor = abs(normal);
      >;
    >;
    draw(getSize(mesh));
  >;
```

(b) An equivalent program in BraidGL.

Fig. 2. A bare-bones shader in WebGL and in Braid.

Code that runs on the GPU consists of short kernels called *shader programs*, which are written in a GPU-specific programming language. Each programmable stage in the rendering pipeline can be customized with a different kind of shader program. The two most common kinds are *vertex shaders*, which compute the position of each vertex in a 3D mesh, and *fragment shaders*, which compute the color of each pixel on an object's surface. The CPU sends input vertex parameters to

the vertex shader, which executes once per vertex to compute a position vector. Next, the GPU's fixed-function logic uses the computed vertex positions to produce a set of primitive polygons. The fragment shader executes once per pixel in the area of each polygon, interpolating between the vertex positions, and produces a color for each pixel. The pipeline organization means that each stage can produce data for the next stage to consume.

Figure 1 illustrates these execution phases for the minimal example program in Figure 2a using JavaScript, WebGL, and GLSL. In this example:

- The CPU-side *setup* phase ① loads 3D mesh data, compiles the GLSL shader source code, and looks up "location" handles for GPU-side variables. Later code will use these handles to communicate with the GPU.
- The *render loop* ② executes on the CPU, once for each frame. It tells the driver which shader program to use, binds parameters using the location handles, and invokes a draw command to trigger the GPU pipeline. This example uses *transformation matrices* called `projection`, `view`, and `model`, which determine the position, orientation, and scale of the camera and the object. The vector arrays `position` and `normal` describe the object's shape; each consists of one vector per vertex.
- The *vertex shader* ③ is a GLSL program that writes to the magic variable `gl_Position` to determine the coordinates of each vertex. This example uses an input object-relative vertex position, given by `position`, and multiplies it with a series of transformation matrices to compute a view-relative position. It passes the `normal` vector to the fragment shader using the prefixed names `aNormal` and `vNormal` to distinguish the vertex and fragment shaders' views of the same data.
- The *fragment shader* ④ writes to `gl_FragColor` to produce each pixel's color. The GPU gathers the values of `vNormal` computed in neighboring vertex shader executions, interpolates them, and uses the result for the `vNormal` variable in the fragment shader.

This example demonstrates three major pitfalls in OpenGL programming.

Programming in strings. The host program embeds the GPU programs as string literals and passes them to the graphics driver for just-in-time compilation. This stringly-typed API limits static checking and obscures data flow, causing syntax errors and undefined variables to stay silent at compile time and crash the program at run time.

The potential for subtle errors makes it challenging to experiment with different strategies for code placement. For example, a common optimization technique is to move computations from the fragment shader to the vertex shader, where they run at a lower rate, or from the vertex shader to the CPU, where they run only once per frame. The transformation matrix multiplication `uProjection * uView * uModel` in the vertex shader could also occur on the CPU, but moving this computation would require changing the programming language, the API for matrix multiplication, and the variable names.

Unsafe, verbose communication. Inter-stage communication involves three steps. Each shader declares its parameters and outputs: `uniform` denotes a per-draw-call parameter; `attribute` denotes a per-vertex parameter; and `varying` denotes a communication channel between different shaders. During setup, the host code uses each parameter's name to look up its handle. Finally, to draw each frame, the host code uses the handles to assign the parameter values. None of these constructs are type-checked at compile time.

Some general-purpose GPU programming languages, including CUDA and its successors [Howes and Rovatsou 2016; OpenACC 2015; Sander et al. 2015], use a *single-source* model where communication can be checked at compile time. These simpler languages do not target real-time graphics programming, which requires composition of multiple nested stages to model the rendering pipeline.

Unscalable specialization. To extract high performance from the GPU, graphics applications specialize general code to create simpler shader *variants*. Because divergent control is expensive on GPUs, shaders avoid `if` branches. Instead, programmers often use GLSL's C-preprocessor-like `#ifdef` to generate cheaper, straight-line shader programs:

```
#ifdef TINT
gl_FragColor = abs(vNormal) + vec4(0.1, 0.1, 0.5, 1.0);
#else
gl_FragColor = abs(vNormal);
#endif
```

The C preprocessor approach suffices for a handful of build flags but does not scale well to the massive specialization required in modern games, which can generate hundreds of thousands of shader variants [He et al. 2016, 2015]. Traditional `#ifdef` code manipulation makes it difficult to ensure that each variant will pass type checking or even parse. CUDA and other general-purpose GPU languages have the same limitation.

2.2 Real-time Graphics Programming with Braid

Braid uses static staging to address these problems. Figure 2b shows the BraidGL version of the WebGL program in Figure 2a. The successive pipeline stages in Figure 1 map to a nested series of stages. The programmer *quotes* code with angle brackets <*e*> to control code generation and placement: `js`<*e*> delimits JavaScript code executing on the CPU and `glsl`<*e*> indicates GLSL code executing on the GPU. The example in Figure 2b uses three pairs of angle brackets to delimit four stages, from the outermost setup stage to the innermost fragment-shader stage. Data flows from earlier to later stages using variable references that cross stage boundaries. The BraidGL compiler provides `render`, `vertex`, and `fragment` intrinsics that generate code for the appropriate compute unit. This three-level nesting reflects the essential complexity of the real-time graphics domain. A single level of staging would suffice to program other emerging forms of hardware heterogeneity, such as fixed-function accelerators for machine learning or encryption.

This section uses our example to give an overview of Braid. Section 3 describes it in detail.

Unified, type-safe programming. Braid does not use strings for code. The code for the setup, render, vertex, and fragment stages all coexist in a single, type-safe program. This uniformity makes the flow of data clear, detects errors statically, and simplifies the job of moving computations between stages. For example, since the transformation matrices are constant in this program, we could move their multiplication to the CPU:

```
var pvm = projection * view * model;
vertex glsl<
  gl_Position = pvm * position;
  # ...
```

This change reduces CPU–GPU communication but increases the CPU's workload, so the best placement depends on the platform. A uniform programming model simplifies rapid exploration of placement strategies.

Safe communication. Braid enforces a consistent interface between heterogeneous components. The vertex shader references the `position` variable, which is defined in the render-loop stage.

Cross-stage references are a special case of a more general concept in Braid called *materialization* that abstracts communication channels in heterogeneous hardware.

Metaprogramming for specialization. Programmers can define *staging-based macros* that transform code for efficiency. A programmer might write a compile-time conditional macro, `@static_if`, to replace the `#ifdef`-based conditional tinting from above:

```
gl_FragColor = @static_if tint
  (abs(normal) + vec4(0.1, 0.1, 0.5, 1.0))
  abs(normal)
```

Because it is based on stages, Braid's macro system guarantees that *all generated programs will be well-typed.* Unlike with `#ifdef` metaprogramming, programmers can trust that any possible combination of compile-time options will lead to meaningful shader code. The key idea is to treat the compilation phase as another stage that precedes runtime setup. Stages thus offer a common basis for compile-time specialization and run-time coordination.

2.3 Dynamic vs. Static Stages

Braid's static staging approach generalizes *multi-stage programming* [Taha and Sheard 1997]. Traditional staging focuses on constructing compilers and domain-specific languages [Brown et al. 2011; DeVito et al. 2013; Rompf and Odersky 2010; Rompf et al. 2014]. In this *dynamic staging* approach, a stage-0 program generates a stage-1 program, which may generate a stage-2 program, and so on. If the stages need to communicate, an earlier stage bakes values into the later-stage program as constants [Kiselyov 2014; Rompf and Odersky 2010; Taha and Sheard 1997].

Traditional multi-stage programming's flexible semantics leads to costly implementations. Current implementations, such as Scala LMS [Rompf and Odersky 2010], Terra [DeVito et al. 2013], and MetaOCaml [Calcagno et al. 2003b], build up abstract syntax trees and compile them on the fly. AST manipulation enables arbitrary code transformation, but it requires that staged programs invoke a full compiler to run any later-stage code. The cost is acceptable for offline code generation but inappropriate for application code.

Our work extends and restricts staging to make it practical to use in heterogeneous programs. The key goal is to generate code ahead of time for all stages. Static staging introduces *materialization*, a new construct that denotes inter-stage communication without AST manipulation. Materialization coexists with traditional *splicing*, which retains prior work's code generation semantics. The static programming model means that staged Braid programs do not need to bundle a Braid compiler to execute. The next section details the distinction between materialization and traditional splicing and its consequences for the language implementation.

3 STATIC STAGING IN BRAID

Braid's goals are to express placement and specialization in a heterogeneous system while statically compiling efficient code. Its unique features in service of these goals are:

- A new kind of escape expression, called *materialization*, for inter-stage communication.
- Multi-level escape expressions, which compose staging constructs for specialization together with staging constructs for placement.
- An optimization, *pre-splicing*, that avoids the cost of runtime metaprogramming while preserving its semantics.
- A hygienic macro system that uses staging to encapsulate reusable specialization strategies.

Table 1 enumerates Braid's staging constructs.

This section describes the Braid language and its prototype compiler. For simplicity, the basic Braid system has only one target, JavaScript, and uses unannotated quotes <*e*>. The compiler emits

Syntax	Name	Section	Purpose
`target<e>`	quote	3.1	Produce a code value for `target` that will run `e`.
`!e`	run		Execute a code value.
`[e]`	splice		Inline a code value from the parent scope.
`%[e]`	materialize	3.2	Communicate a value from the parent scope.
`n[e]`	multi-level escape	3.3	Evaluate `e` in the context `n` levels away and splice.
`$[e]` & `$<e>`	open code	3.4	Splice while preserving the current quote's scope.
`@name`	macro invocation	3.5	Call a function defined at any earlier stage.

Table 1. Language constructs in Braid.

string-wrapped JavaScript code for each <*e*> and runs the code using `eval`. Section 5 shows how we add compiler backends for hardware targets in real-time graphics.

3.1 Traditional Staging: Quote, Run, and Splice

Braid borrows three central constructs from traditional multi-stage programming: the quote, run, and splice expressions. *Quotation* (also known as *bracketing* in MetaML [Taha and Sheard 1997] and *quasiquotation* in Lisp [Bawden 1999]) denotes a deferred computation and produces a *code value*. In Braid, quotation is written using angle brackets:

```
var program = <
  var highlight = fun color:Vec3 ->
    min(color * 1.5, vec3(1.0, 1.0, 1.0));
  highlight(vec3(0.2, 0.8, 0.1))
>
```

Here, `program` is a code value that, when executed, defines and invokes a function. Its type is written `<Vec3>`, which indicates that the delayed code will produce a value of type `Vec3`. A *run* operator, `!`*e*, executes code values. Here, `!program` evaluates to the vector $(0.3, 1.0, 0.15)$.

Quotations use a *splice* expression (also called *escape* or *unquote*), to compose code values. In Braid, square brackets denote splicing. The splice expression `[`*e*`]` must appear inside a quote. It evaluates *e* in the quote's *parent* context to produce a code value to insert into quote. For example, splicing can "bake in" a parameter as a constant:

```
var makeHighlight = fun amount:<Float> ->
  < fun color:Vec3 ->
      min(color * [amount], vec3(1.0, 1.0, 1.0)) >
```

The `[amount]` splice looks up a code value from the environment *outside* the quote and splices it into the body of the quoted highlighting function. The `makeHighlight` function takes a parameter of type `<Float>`, which may be any quoted code that produces a `Float`. For example, invoking the function and then running the result with the expression `!(makeHighlight(<2.0>))` produces a function with the literal `2.0` inlined into its body. The result is equivalent to:

```
fun color:Vec3 ->
  min(color * 2.0, vec3(1.0, 1.0, 1.0))
```

In this example, splicing is akin to partial evaluation.

Compiling stages statically. The baseline Braid compiler emits JavaScript code for all stages. It wraps quoted code in string literals and compiles the run operator `!` to JavaScript's `eval`. Splicing uses string replacement with a magic token. Our `makeHighlight` function compiles to this JavaScript code:

```
var QUOTE_1 = "function (color) {" +
"  return min(color * __SPLICE_1__, vec3(1.0, 1.0, 1.0))" +
"}";
var makeHighlight = function (amount) {
  return QUOTE_1.replace("__SPLICE_1__", amount);
}
```

Our above invocation compiles to JavaScript using `eval`:

```
var QUOTE_2 = "2.0";
var highlight2 = eval(makeHighlight(QUOTE_2));
```

This example shows that the Braid compiler emits quoted and non-quoted code at the same level of abstraction, as JavaScript statements. Traditional staged compilers represent quoted code as an AST that looks very different from the executable code emitted for non-quoted expressions. In Braid, there is no runtime AST data structure, and running quoted code does not require an additional translation step. This ahead-of-time code generation is key to performance in heterogeneous programming. BraidGL, for example, emits complete GLSL programs from shader quotes at compile time so it can match the performance of manually-written OpenGL code.

This trio of staging concepts—quote, run, and splice—make up Braid's tools for traditional metaprogramming. Next, we describe Braid's extensions to staging for heterogeneous programming.

3.2 Expressing Communication: Materialization

Braid adds an abstraction for efficient inter-stage communication. In traditional dynamic staging, the only way to communicate with a later stage is to *lift* a raw value to a code value and splice it into the later stage's code [Taha and Sheard 1997]. Traditional splicing does not suffice for heterogeneous programming, where it is both inefficient and inflexible. Instead, Braid introduces a new communication construct, called *materialization*, that makes a value defined in one stage available in another, later stage.

For example, this GPU-side function adjusts a color by a CPU-specified amount:

```
var amount = 0.8;
< fun color:Vec3 ->
    min(color * %[amount], vec3(1.0, 1.0, 1.0)) >
```

The expression `%[amount]` is a materialization escape. Like a splice escape, `%[`*e*`]` evaluates *e* in the current quote's parent context. But unlike with splicing, *e* need not be a code value. It may have an ordinary type, such as `Float`, because materialization does not manipulate any code—it communicates *e* through shared memory or dedicated communication channels, depending on the hardware target. In BraidGL, materialization abstracts the OpenGL APIs for binding shader parameters. Materialization distinguishes Braid from prior multi-stage languages, which view splicing as the only method for cross-stage persistence [Hanada and Igarashi 2014; Kiselyov 2014]. The explicit distinction between splicing and materialization is the core concept behind static staging that differentiates it from prior multi-stage programming work.

Implementing materialization. To compile quotes containing materialization expressions, the Braid implementation uses a strategy similar to lambda lifting for closures. The hardware target determines how to communicate materialized data. For CPU targets, a code value consists of a code pointer and an environment containing materialized values. Our highlighter example compiles to:

```
var QUOTE_1 = "... color * m1 ...";
{ code: QUOTE_1, env: { m1: amount } }
```

The code value is a JavaScript object. Its `env` member maps the opaque materialization name `m1` to a concrete value. To execute code values with materialized data, programs bind the environment's names and then invoke `eval`.

Section 5 describes how BraidGL implements materialization for CPU–GPU communication.

Cross-stage references. While materialization expressions can be useful on their own, Braid programmers typically use materialization *implicitly* via references to variables defined at earlier stages. These cross-stage references use materialization under the hood without requiring bracket syntax. For example, this version of the highlighter program uses cross-stage references to the variables `amount` and `color`:

```
fun color:Vec3 amount:Float ->
  < min(color * amount, vec3(1.0, 1.0, 1.0)) >
```

This code has identical semantics to a quote that uses explicit materialization escapes:

```
< min(%[color] * %[amount], vec3(1.0, 1.0, 1.0)) >
```

The type checker keeps track of the stage where each variable is defined, and when a reference crosses a stage boundary to reach its definition, the compiler uses materialization to communicate the value.

Cross-stage references are *nearly* syntactic sugar for materialization escapes. A reference to an earlier-stage variable such as `color` has the same semantics as a materialized reference `%[color]`, but it differs in performance when a stage uses the same variable multiple times. Consider the highlighter with a parameterized channel limit:

```
< min(color * amount, vec3(limit, limit, limit)) >
```

This code should not waste bandwidth to communicate three copies of the `limit` value to the GPU. The Braid compiler ensures that cross-stage references communicate their values only once.

3.3 Multi-Level Escapes

Staging in Braid plays two distinct roles: specialization and placement. In programs that compose both roles, it must be possible to specialize code that uses placement to coordinate multiple hardware units. For example, this program creates code to run on the GPU:

```
var amount = ...;
var shader = fun color:Vec3 -> <
  gl_FragColor = [
    if (amount == 1.0)
      < color >
      < min(color * amount, vec3(1.0, 1.0, 1.0)) >
  ]
>
```

It uses a splice escape to decide whether to emit code to highlight the color parameter. When `amount` is 1.0, the shader can use simpler code and avoid wasting bandwidth to communicate the value. However, this program uses *runtime* metaprogramming. Every invocation of the `shader` function will splice together a fresh shader code string. This dynamic code generation is both inefficient and unnecessary: `amount` cannot change between calls, so the generated code is the same every time. To fix this problem, Braid programs can opt into *compile-time* metaprogramming using *multi-level escape* expressions.

Multi-level escapes generalize the traditional escape construct to enable splicing and materialization between non-adjacent stages. By annotating an escape expression $n[e]$ with a number n, programs can evaluate e and splice it at the nth prior stage *without passing through the intermediate stages.* While a single-stage splice escape produces metaprogramming code at the immediately preceding stage, a multi-stage splice lets the programmer move the cost of metaprogramming earlier—even to compilation time.

In our example above, metaprogramming occurs at the CPU execution stage, but we would like to perform it at compile time instead to eliminate its run-time overhead. We can introduce a compile-time stage by wrapping our program in a new, top-level quote:

```
var amount = ...;
!<
  var shader = fun color:Vec3 -> <
    gl_FragColor = 2[
      if (amount == 1.0)
        < color >
        < min(color * amount, vec3(1.0, 1.0, 1.0)) >
    ]
  >
>
```

By writing the splice escape as `2[...]`, this version performs splicing at the *outer* specialization stage. Executing this program performs splicing, but it produces a second-stage program that does not do any splicing itself. The resulting shader code uses one of the two alternative expressions and omits the other entirely. A plain, single-level splice expression, in contrast, would instead materialize `amount` at run time and use it to splice the shader dynamically, as in the previous version of the example.

The numeric stage indexing in multi-level escapes is powerful but not always ergonomic. Rather than as a user-facing feature, multi-level escapes are important as an underpinning for cross-stage references (Section 3.2) and macros (Section 3.5). Our formalism for Braid (Section 4) uses them to define the semantics for both.

Implementing generalized splicing. The Braid compiler recursively emits nested quotes using nested JavaScript string literals. Stage splicing works by replacing a token within an inner string:

```
var amount = ...;
var QUOTE_1 = "var shader = function (color) {" +
  "var QUOTE_2 =" +
  "  \"gl_FragColor = __SPLICE_1__\";" +
  "return QUOTE_2;" +
"}";
QUOTE_1.replace("__SPLICE_1__", ...);
```

The two-level splice `2[e]` is *not* equivalent to a nested splice `[[e]]`. The latter splices code twice, once in each stage:

```
var amount = ...;
var QUOTE_1 = "var shader = function (color) {" +
  "var QUOTE_2 =" +
  "  \"gl_FragColor = __SPLICE_1__\";" +
  "return QUOTE_2.replace(\"__SPLICE_1__\", __SPLICE_2__);"  ①
 + "}";
QUOTE_1.replace("__SPLICE_2__", ...);  ②
```

The outer splice ② runs first and inserts code into the outer quote, which splices again ① to transfer the value into the inner quote. Multi-level splicing is required to generate staged code that does not itself generate any code at run time.

3.4 Expressive Metaprogramming with Open Code

As described so far, quotes must contain self-contained, executable programs without unbound variable references. This strategy rules out important specialization strategies. For example, a graphics shader might need to conditionally darken or lighten a computed color:

```
var night = ...;
var shader = <
  var color = get_color();  ①
  gl_FragColor = [
    if night
      < color * 0.7 >  ②
      < min(color * 1.2, vec3(1.0, 1.0, 1.0)) >  ③
  ]
>
```

Both quoted `color` references ②③ are illegal because the variable is not defined in any earlier stage. A *sibling* quote ① declares the variable, but cross-stage references may only safely refer to enclosing ancestor stages.

In the multi-stage programming literature, these self-contained quotes with no out-of-scope references are called *closed code*. Some languages allow *open code*, where quotes can refer to variables that are locally undefined [Benaissa et al. 1999; Moggi et al. 1999; Nanevski and Pfenning 2005; Taha and Nielsen 2003]. Unconstrained open code prohibits static code generation because variable references can be undefined until run time. Braid introduces a limited form of open-code quotation that allows static compilation.

In Braid, programs opt into open code by annotating escapes and quotes with a `$` prefix. For instance, this version of the quoted shader code is legal:

```
var color = get_color();
gl_FragColor = $[
  if (night)
    $< color * 0.7 >
    $< min(color * 1.2, vec3(1.0, 1.0, 1.0)) >
]
```

Open quotes `$<...>` share the environment of their nearest corresponding `$`-prefixed escape expression, so the references to `color` in this version of the code are legal. Whereas ordinary quotes might be spliced anywhere, including into contexts where `color` is not defined, these open quotes may *only* be spliced into their nearest `$[...]` escape, where `color` is guaranteed to be defined. The type system enforces this requirement to prohibit splicing anywhere else. For example, this Braid program produces a type error:

```
var sneaky;
var shader1 = <
  var color = ...;
  $[ sneaky = $<color> ];
>;
var shader2 = < $[sneaky] >;
```

This contrived example would smuggle a reference to the variable `sneaky` from a context where it is defined (`shader1`) into a context where it is undefined (`shader2`). The type system prohibits the assignment to `sneaky` to prevent the ill-formed splice.

This simple type constraint, where every quote has exactly one splice point, suffices, but future work may explore rules that add flexibility while maintaining static safety. For example, the compiler could allow multiple potential splice points, as long as they are statically enumerable, or programs could transform open quotes with additional context before splicing them at their destinations.

3.4.1 Pre-Splicing to Avoid Code Generation. Braid's restrictions on open code provide an opportunity to avoid the cost of runtime metaprogramming where it seems unavoidable. Consider a version of our highlighting shader where `amount` is a runtime parameter:

```
var shader = fun color:Vec3 amount:Float -> <
  gl_FragColor = $[
    if (amount == 1.0)
      $< color >
      $< min(color * amount, vec3(1.0, 1.0, 1.0)) >
  ]
>
```

Because `amount` is unknown at compile time, the multi-level escape approach from Section 3.3 will not work. Instead, this program splices GLSL code just before executing it to avoid a branch on the GPU. But there are only two possible final versions of the complete shader program, so it is wasteful to re-splice the code on every `shader` invocation.

The Braid compiler introduces an optimization to avoid wasteful code generation. *Pre-splicing* leverages the type system's open-code restrictions, which ensure that the set of resolutions for each `$[...]` escape is statically enumerable. The optimization iterates through each possible resolution of each such escape and produces a *variant* of the quote that inlines the chosen quotes. Then, it transforms the code from each splice escape to look up the correct variant in a table. In our example, pre-splicing produces optimized code equivalent to this escape-free program:

```
var shader = fun color:Vec3 amount:Float ->
  if (amount == 1.0)
    < gl_FragColor = color >
    < gl_FragColor = min(color * amount, vec3(1.0, 1.0, 1.0)) >
```

The JavaScript backend emits a switch on the IDs that identify each resolution:

```
var QUOTE_1_1 = "...";  // The pre-spliced variants.
var QUOTE_1_2 = "...";
var id = (amount === 1.0) ? 1 : 2;
switch (id) {  // Variant lookup.
case 1:
  return QUOTE_1_1;
case 2:
  return QUOTE_1_2;
}
```

Pre-splicing trades reduced runtime cost for increased code size by generating a combinatorial space of specialized programs. For example, if a quote has two pre-spliced escapes, and each contains 3 possible subquotes, then the optimization will produce 3×3 complete variants. The Braid compiler provides a flag to disable the pre-splicing optimization if exponential code bloat becomes a problem.

3.5 Reusable Specialization with Macros

To make specialization strategies reusable, Braid adds a staging-based macro system. Library writers express common patterns for optimizing general code, and application writers reuse them without too much careful reasoning about stages. For example, some examples above use an `if` at an earlier stage to statically choose between two expressions. Using Braid's macro system, we can define a "specialized `if`" construct, `@spif`, which substitutes for `if` in quoted code:

```
var night = ...;
< var value = @spif night 0.3 0.8;
  gl_FragColor = vec3(value, value, value) >
```

In Braid, macro invocations are syntactic sugar for explicit staging constructs. A macro invocation, written `@name`, invokes the function `name` at the stage where `name` is defined and splices the result into the current stage. There is no special syntax to define a macro; any function that accepts code values as arguments can be a macro. To define `@spif`, for example, a library author writes an ordinary function that takes code values as arguments:

$$e ::= c \mid x \mid \textbf{var}\ x = e \mid e\ ;\ e \mid \langle e \rangle \mid !e \mid {}_n[e] \mid \%[e] \mid {}^{\$}_{n}[e] \mid \$\langle e \rangle$$
$$x \in \text{variables},\ c \in \text{constants},\ n \in 1, 2, \ldots$$

Fig. 3. Syntax for BraidCore.

```
var spif = fun cond:<Bool> t:<Float> f:<Float> ->
  if !cond t f;
```

Recall that `!` executes code values, so `!cond` in this example evaluates the condition expression.

The Braid compiler desugars macro invocations to multi-level splice escapes (Section 3.3) that skip to the stage where the function's name is bound. The macro's arguments become quote expressions. In our example, the macro is only one quotation level away, so the `@spif` invocation desugars to a single-level escape:

```
1[ spif <night> <0.3> <0.8> ]
```

The macro syntax offers a convenient way for Braid programmers to harness the power of multi-level splicing for specialization without directly reasoning about stages.

Macro functions can also take open-code quotes as arguments. Our `@spif` macro, for example, can let its `t` and `f` arguments refer to variables in the calling context, as in:

```
< var color = ...;
  var value = @spif night (0.3 * color) color; >
```

To resolve the references to `color`, the `spif` definition uses open-code arguments marked with `$`:

```
var spif = fun cond:<Bool> t:$<Float> f:$<Float> ->
  if !cond t f
```

The `$<Float>` parameter type indicates that `t` and `f` should be passed as open quotes.

Braid's type system for staging ensures that code remains well-typed in all stages, and the syntactic sugar for macro invocations inherits the same guarantee. In other words, Braid's macro system is *hygienic:* macro writers need not worry about aliasing names in specialized code [Kohlbecker et al. 1986; Lee et al. 2012].

4 TYPE SYSTEM AND SEMANTICS

We formalize a semantics for BraidCore, a minimal version of Braid, to rigorously define its staging constructs and safety guarantees. This section summarizes the formalism; the accompanying supplemental material gives the full static and dynamic semantics and sketches the proof of a safety theorem. Figure 3 lists the abstract syntax for BraidCore.

The static staging semantics for BraidCore are simpler than most traditional staging semantics because of the limitations on open code. Approaches such as MetaML [Taha and Sheard 1997] use a tagging strategy. Each value carries an integer tag that indicates its relative stage number (negative for earlier stages, positive for later stages). In the rules for quote and escape expressions, the semantics must shift every value's tag upward or downward. BraidCore instead organizes values into per-stage environments on a stack where variables always resolve in the topmost environment.

Type system. Types are either primitive or code types:

$$\tau ::= t \mid \langle \tau \rangle \qquad\qquad t ::= \texttt{Int} \mid \texttt{Float} \mid \cdots$$

A type context Γ consists of a stack of per-stage contexts γ that map variable names to types:

$$\Gamma ::= \cdot \mid \gamma, \Gamma \qquad\qquad \gamma ::= \cdot \mid x : \tau, \gamma$$

The typing judgment $\Gamma_1 \vdash s : \tau; \Gamma_2$ builds up a context. The rule for variable lookup retrieves a value from the γ on the top of the stack, and assignment adds a new mapping:

$$\textsc{type-lookup}\quad \frac{}{\gamma, \Gamma \vdash x : \gamma(x); \gamma, \Gamma} \qquad \textsc{type-var}\quad \frac{\Gamma_1 \vdash e : \tau; \gamma, \Gamma_2}{\Gamma_1 \vdash \mathbf{var}\, x = e : \tau; (x : \tau, \gamma), \Gamma_2}$$

Cross-stage references are omitted from BraidCore because they can be modeled as syntactic sugar for materialization. A splice expression ${}_n[e]$ checks the expression n levels up:

$$\textsc{type-splice}\quad \frac{\Gamma_1 \vdash e : \langle \tau \rangle; \Gamma_2 \qquad \mathrm{len}(\overline{\gamma}) = n}{\overline{\gamma}, \Gamma_1 \vdash {}_n[e] : \tau; \overline{\gamma}, \Gamma_2}$$

Here, $\overline{\gamma}, \Gamma$ denotes a prefix $\overline{\gamma}$ and a tail context Γ, and $\mathrm{len}(\overline{\gamma})$ determines the prefix's size. The result of e must be a code type. Materialization is similar, but the escape only moves a single stage and e need not result in a code type.

Dynamic semantics. We define a big-step operational semantics. Following the type system, a heap H consists of a stack of per-stage environments h. Values are either constants c or code values $\langle\!\langle e, h \rangle\!\rangle$, which consist of an expression and an associated environment that contains the results of materialization expressions.

The main big-step judgment $H; e \Downarrow H'; v$ evaluates an expression to a value and updates the heap. As with the type system, we use $h(x)$ to denote variable lookup. The rules for assignment and variable lookup work with the top per-stage environment in the heap:

$$\textsc{lookup}\quad \frac{}{h, H; x \Downarrow h, H; h(x)} \qquad \textsc{assign}\quad \frac{H; e \Downarrow h, H'; v}{H; \mathbf{var}\, x = e \Downarrow (x \mapsto v, h), H'; v}$$

To evaluate a quotation, the semantics "switches" from the main big-step judgment to a quoted judgment written $H; h; e \Downarrow_i H'; h'; e'$ where i is the current quotation level. The latter judgment scans over the quoted expression to find escapes that need to be evaluated eagerly with the main semantics. It threads through a heap H, which may be updated inside escaped expressions, and a materialization environment h, which holds the results of top-level materialization escapes. The rules leave most expressions, such as assignments, intact:

$$\textsc{quoted-assign}\quad \frac{H; h; e \Downarrow_i H'; h'; e'}{H; h; \mathbf{var}\, x = e \Downarrow_i H'; h'; \mathbf{var}\, x = e'}$$

The judgment switches back to ordinary $\Downarrow$ interpretation when an escape goes beyond the current quotation level. For example, a splice ${}_n[e]$ where $n = i$ returns to the top level. It evaluates e to a code value and includes the resulting expression:

$$\textsc{quoted-splice-resume}\quad \frac{H; e \Downarrow H'; \langle\!\langle e_q, h_q \rangle\!\rangle \qquad n = i \qquad \mathrm{merge}(h, h_q) = h'}{H; h; {}_n[e] \Downarrow_i H'; h'; e_q}$$

A $\mathrm{merge}(h, h')$ helper judgment combines the variable mappings from two environments.

5 STATIC STAGING FOR REAL-TIME GRAPHICS

This section instantiates our language design and implementation for BraidGL, a prototype language for real-time 3D rendering on CPU–GPU systems. The compiler emits a combination of host code

in JavaScript, shader code in GLSL [Segal and Akeley 2016], and interface code using the WebGL API [Jackson and Gilbert 2017].

5.1 Targets and Annotations

In Braid, programmers choose a *target* for each quote that controls how it is compiled. Annotations on quotes, written *t*<*e*>, select a target *t*. Targets specify the code-generation language and hardware. Targets are defined in compiler extensions with three components: a language variant that adds platform-specific intrinsics and removes unsupported features; a code-generation backend; and communication strategies that specify how to materialize values from other targets.

The BraidGL compiler extension defines a shader target, written `glsl`<*e*>. The compiler emits shader quotes as GLSL source code in string literals. The `!` operator cannot run shaders directly; instead, BraidGL defines special intrinsic operations that execute the code in the graphics pipeline.

BraidGL also includes a `js`<*e*> annotation, which directs the compiler to emit JavaScript code as a function declaration and to execute it on the CPU. The per-frame render stage uses this code type in place of plain <*e*> to avoid the cost of using JavaScript's `eval` on each frame.

5.2 Binding Stages with Intrinsics

BraidGL provides three intrinsics that execute a graphics program's stages. The `render` intrinsic registers code to run on the CPU to draw each frame; `vertex` binds a vertex shader; and `fragment` associates a fragment shader with a vertex shader. We leave less common stages, such as tessellation, geometry, and compute shaders [Segal and Akeley 2016], for future work.

The `render` intrinsic registers code that draws each new frame. The `vertex` intrinsic compiles to a call to WebGL's `gl.useProgram()`, which instructs the `gl` context to use a given compiled shader program for the next draw call. Finally, `fragment` instructs the compiler to emit setup code that compiles the fragment shader's code together with its containing vertex shader to create an executable program. Together, this nesting of intrinsics and quotes in BraidGL:

```
render js<
  # (render loop code)
  vertex glsl<
    # (vertex shader code)
    fragment glsl<
      # (fragment shader code)
    > > >
```

compiles to this JavaScript code:

```
var QUOTE_1 = "(vertex shader code)";
var QUOTE_2 = "(fragment shader code)";
var shader_1 = compile_shader(gl, QUOTE_1, QUOTE_2);
add_renderer(function () {
  // (render loop code)
  gl.useProgram(shader_1);
});
```

The `compile_shader` runtime function wraps WebGL's GLSL compilation calls, and `add_renderer` registers a callback to draw each frame [Web Hypertext Application Technology Working Group 2017]. The end result is a JavaScript program that resembles hand-written WebGL code.

5.3 Binding Shader Parameters

Materialization expressions and cross-stage references in BraidGL denote communication constructs in WebGL. This example communicates a vector value from the CPU to both shader stages:

Program	BraidGL LoC	JS+GLSL LoC
Phong	61	119
head	59	141
couch	144	218

Table 2. The lines of code in each case study program and its equivalent JavaScript and GLSL code.

```
var color = vec3(0.2, 0.5, 0.4);
vertex glsl<
  ... color ...
  fragment glsl< ... color ... >
>
```

The compiler generates three ingredients for each materialization. First, it emits a `uniform`, `varying`, or `attribute` declaration in the GLSL code for each shader to indicate its source:

```
var QUOTE_1 =
  "uniform vec3 param_1;" +
  "void main() { ... param_1 ... }";
```

Next, the compiler emits a location handle lookup in the host code:

```
var shader_1 = compile_shader(gl, QUOTE_1, QUOTE_2);
var shader_1_loc_1 = gl.getUniformLocation(shader_1, "param_1");
```

Finally, when compiling the `vertex` intrinsic, the compiler emits code to bind materialized values:

```
gl.uniform3fv(shader_1_loc_1, vec3(0.2, 0.5, 0.4));
```

The materialization's type dictates which OpenGL API call to use in the generated code. For example, the `gl.uniformMatrix4fv` function communicates a 4×4 matrix value.

Vertex attributes. The above example shows a *uniform* shader parameter, which is constant across all vertex shader invocations. BraidGL also supports *vertex attribute* parameters for per-vertex data. For example, programs may communicate a model's mesh coordinates using a vertex attribute.

In BraidGL, a `T Array` represents a dynamically-sized buffer of `T` values. When a shader's materialization expression has a `T Array` type, the compiler binds it as an attribute instead of as a uniform. The result in the shader quote has type `T` and refers to the *current* value of the attribute. For example, in this materialization from our earlier example:

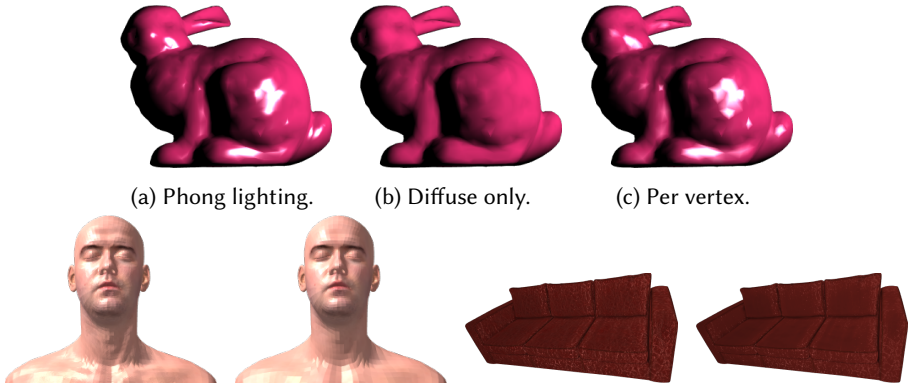
```
var position = getPositions(mesh);
vertex glsl<
  ... %[position] ...
>
```

the `position` variable has type `Vec3 Array`, but the materialization inside the vertex shader produces a single `Vec3`. The BraidGL compiler uses WebGL's `gl.vertexAttribPointer` API call to bind attribute buffers.

OpenGL does not support direct communication of attributes from the CPU to the fragment shader. Attributes can only communicate to the vertex shader. To implement array references in fragment shaders, BraidGL emits code to copy the attribute's value through the vertex shader via `varying`-qualified parameters.

6 EVALUATION

We evaluate Braid's ergonomics to show how static staging helps graphics programmers explore placement and specialization choices. We develop three case studies and describe how to accomplish a variety of well-known visual effects and optimization techniques.



(a) Phong lighting. (b) Diffuse only. (c) Per vertex.

(d) Head. (e) No bump map. (f) Couch. (g) Medium detail.

Fig. 4. Outputs from the case study programs.

We implement a compiler for Braid and BraidGL in 5554 lines of TypeScript and a grammar for the PEG.js [Majda 2010] parser generator. The compiler uses a generalization of lambda lifting, which we call *scope lifting*, to compile both functions and quotes into closure-like structures. BraidGL emits JavaScript code that uses the WebGL 1.0 API, a Web standard supported by all major browsers [Jackson and Gilbert 2017]. We have developed a live-coding environment for BraidGL that runs the compiler and the generated code in the browser. The open-source implementation and the interactive environment are available online [Sampson 2017].

The implementation includes a complete interpreter for Braid. Unlike the compiler, the interpreter implements quotes using runtime ASTs, so it can pretty-print staged code. Pretty-printing is useful for debugging Braid programs, but we use the compiler for all performance experiments.

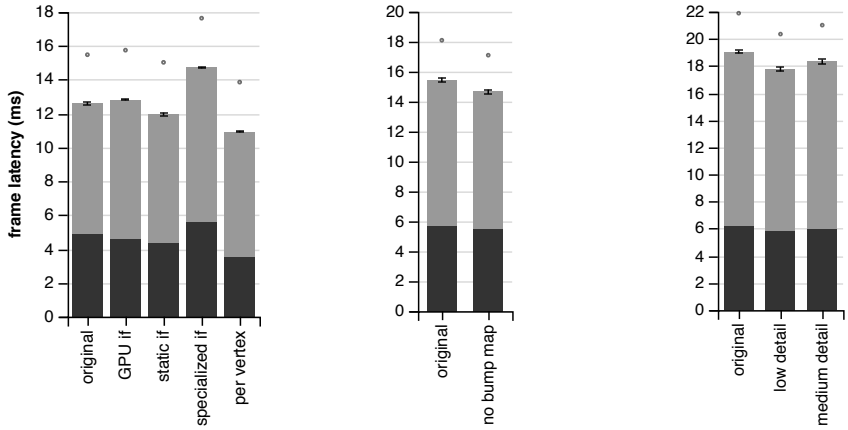
Table 2 shows the lines of BraidGL code in each program compared to the equivalent JavaScript and GLSL, which can be twice as verbose. By design, the BraidGL compiler emits code that matches handwritten WebGL code, up to cosmetic differences such as variable naming. No abstraction in BraidGL prevents the compiler from delivering the same performance as an equivalent hand-written version of the program.

To evaluate performance, we measure the time it takes to draw each frame. Each program renders an 8×8×8 grid of objects. A test harness loads the scene in a browser and measures the execution time of the compiled JavaScript using the browser's high-resolution timing API, `Performance.now` [Grigorik et al. 2016]. We also measure *draw time*, the total time per frame spent in OpenGL's `glDrawArrays` and `glDrawElements` calls, which invoke the graphics pipeline to draw pixels. Draw time measurements help separate the time spent on CPU and communication from the core GPU rendering time. We collect per-frame latencies over 8 seconds and report the mean and 95th percentile times.

The experiments used an Apple MacBook Pro with a quad-core 2.3 GHz Intel 3615QM CPU and an Nvidia GeForce GT 650M GPU running the Safari 10.0 browser with WebKit 12603.1.1 on macOS 10.12 (build 16A270f).

6.1 Phong Lighting Model

The Phong reflection model [Phong 1975] is a ubiquitous algorithm for approximating lighting effects. Figure 4a shows the Phong program stylizing the "Stanford bunny" mesh [Stanford 2003].



(a) Phong lighting. (b) Head. (c) Couch.

Fig. 5. The average time to draw a frame for three case studies. The bar height depicts the total time. The black portion is the time spent in WebGL draw calls. Error bars show the standard error of the mean. Points show the 95th percentile latency.

Programmers often render 3D objects with different components of a general shader program. Phong lighting, for example, consists of two components: *diffuse* lighting, which resembles light cast on a matte surface, and a *specular* component, which approximates reflections on a glossy material. Figure 4b shows an object rendered with only diffuse lighting, simulating a matte material. We explore three strategies for customizing the Phong shader to draw matte and glossy objects: a straightforward dynamic conditional and two specialization techniques. We then demonstrate a common optimization that trades off accuracy for efficiency.

Dynamic conditional. To choose an object's appearance, the programmer can choose to introduce a dynamic condition into the GPU-side shader code to choose an object's appearance. The CPU chooses whether to draw a given object as matte or glossy, and then the GPU includes one or both lighting components:

```
render js<
  var matte = get_appearance();
  vertex glsl<  # ...
    fragment glsl<  # ...
      var color = if matte diffuse (diffuse + ...);
```

The final `...` placeholder represents the code that computes the specular Phong component. To communicate the `matte` condition variable, the Braid compiler generates code to bind a uniform parameter to the Phong shader. The Braid code is free of boilerplate and makes it clear that the parameter choice is located on the CPU.

Figure 5a shows the dynamic conditional's small performance cost. The *original* bar is the unconditional shader and *GPU if* denotes the version with a dynamic condition. The latter takes 12.8 ms on average to render each frame, compared to 12.6 ms for the original.

Static conditional. Programmers often need to convert dynamic conditions into compile-time decisions for efficiency. When all objects in a scene use the same parameters—when they are all matte, for example—a dynamic conditional is wasteful. A compile-time *static if* construct uses a macro that executes at compile time:

```
var matte = 1;  # Compile-time parameter.
var static_if = fun c:Int t:$<Float3> f:$<Float3> ->
  if c t f;
!<
  render js<
    # ... Phong code ...
  >
>
```

Wrapping the entire program in `!<...>` introduces a compile-time stage. To convert the dynamic condition to a static one, we replace `if` with `@static_if` in the shader code:

```
var color = @static_if matte diffuse (diffuse + ...);
```

No other changes are necessary. The generated WebGL code does *not* use a uniform parameter to communicate `matte`, and the compiler removes the code for specular reflection. In Figure 5a, the *static if* bar shows this version of the Phong program with `matte` turned on. The frame latency is 12.0 ms, which is slightly faster than the original program. Braid's stages provide zero-cost compile-time specialization.

Specialized conditional. Applications can gain performance by generating simpler versions of a shader and choosing between them on the CPU. This transformation reduces the control burden on the GPU, so modern games can generate hundreds of thousands of specialized shader variants [He et al. 2015]. However, rapidly switching between shaders can incur overhead, so the advantage of specializing depends on the hardware and workload. It is important to experiment with both.

Braid makes it simple to switch between the two strategies. In the Phong program, we add a *specializing if* macro, `@spif`, that resembles `@static_if` but is declared in the CPU stage instead of a compile-time stage. We then replace the `if` in the original code with `@spif` to get identical behavior but different performance:

```
var color = @spif matte diffuse (diffuse + ...);
```

The `@spif` macro uses pre-splicing (Section 3.4.1) to produce two condition-free versions of the GLSL shader. The compiled CPU code chooses which to bind based on the `matte` variable. This flexibility highlights how static staging unifies *run-time* decisions for placement and *compile-time* decisions for specialization.

In Figure 5a, the *GPU if* and *specialized if* bars compare between a GPU-side dynamic condition and the specialized equivalent. In this case, the overhead of specialization is a disadvantage: its frame latency is 14.7 ms, compared to 12.8 ms for a standard `if`. The difference emphasizes the need for a language that lets programmers rapidly experiment with both versions to make judgments based on empirical measurements.

Per-vertex promotion. Shader developers move code between the vertex and fragment shaders to trade off between performance and visual quality. The fragment stage runs many times between each vertex, so while code in the fragment shader is more costly than per-vertex code, it computes more detailed visual effects. The graphics pipeline interpolates results from the vertex stage for the fragment stage. Programmers *promote* fragment computations to the vertex stage to produce cheaper, lower-quality results.

Traditionally, experimenting with promotion requires large code changes. In Braid, materialization makes it trivial. We promote the lighting computation to the vertex stage by wrapping it in materialization brackets:

```
vertex glsl<  # ...
  fragment glsl<
    var color = %[
      # ... lighting code ...
    ] > >
```

Figure 4c shows this version's lower-quality output. In Figure 5a, the *per vertex* bar shows the performance advantage: the optimized version's latency is 10.9 ms, compared to 12.6 ms for the original program.

6.2 3D-Scanned Head Rendering

To show how developers experiment with textures, we use the *head* 3D scanning data from McGuire [2011]. It comprises a mesh, a surface texture, and a bump map. We implement a renderer using all three. Figure 4d shows the output. We demonstrate how programmers can use BraidGL to avoid the boilerplate typically associated with textures and to eliminate their cost with a simple specialization strategy.

Texture mapping. Many shaders use GPUs' special support for *texture mapping*, which "paints" a 2D image onto the surface of a 3D object. The object's data includes a mapping between mesh vertices and coordinates in the image. In WebGL, loading images into the GPU's texture memory requires even more boilerplate code than binding ordinary shader parameters. The programmer chooses a numbered *texture unit*, activates it, and binds a texture object before rendering the object:

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, preloaded_texture);
gl.uniform1i(sampler_loc, 0);  // unit "0" corresponds to gl.TEXTURE0 above
```

Then, in GLSL, the texture appears as a uniform value. The shader passes the value into a texture lookup function, `texture2D`:

```
uniform sampler2D sampler;
varying vec2 texcoord;
void main() {
  vec4 color = texture2D(sampler, texcoord);
}
```

The BraidGL compiler implements a materialization strategy that eliminates the boilerplate. Materializing a `Texture` value from the CPU stage to a GPU stage automatically generates binding code for a GPU texture unit. In the *head* program, the GPU materializes a CPU-defined variable `tex`:

```
var texcoord = mesh_texture_coordinates(head);
var tex = texture(load_image("head.jpg"));
render js< # ...
  vertex glsl< # ...
    fragment glsl<
      var color = texture2D(tex, texcoord);
    > > >
```

The compiler chooses units for the two textures in *head* and generates the code to bind them.

Optional bump mapping. Programmers use *bump mapping* to add detail to surfaces by changing the way they reflect light. A bump map texture dictates how much to deflect the normal vector at each point. In the *head* program, the bump map makes the object look more realistic, but it may be unnecessary in some contexts, such as when the object is in the distance. Figure 4e shows the output without bump mapping.

We use a Braid macro to make bump mapping optional in the *head* program. The macro, `@bump?`, takes the bump-map lookup code as an argument and, when a compile-time `use_bump_map` flag is cleared, returns a default zero-deflection vector instead. When bump mapping is disabled, the

corresponding texture materialization is eliminated, and the BraidGL compiler does not generate the code that loads binds the bump texture. The frame latency (Figure 5b) improves from 15.4 ms to 14.7 ms with this change.

6.3 Procedural Couch Model

We use a *couch* scene from He et al. [2016] that exemplifies graphics applications with many specializable features and complex trade-offs between efficiency and functionality. It uses a repeating leather texture, an ambient occlusion map to simulate shadows, a mask texture that controls darkening due to wear, bump mapping, and specular mapping to modulate reflections. Figure 4f shows the output.

Texture averaging. To reduce detail levels for efficiency, graphics programmers need to selectively disable texture layers. We introduce a CPU-side `average` function that computes the mean color in an image. Shaders can replace a texture lookup call, such as `texture2D(leather, texcoord)`, with code that instead *materializes* the CPU-side expression `average(leather)`. Instead of communicating an entire texture, the specialized program sends a single vector to the GPU. Static staging enables local code changes that affect global interactions between heterogeneous components.

Rapid trade-off exploration. When developing complex effects, graphics programmers need to rapidly explore a space of configurations and check their output visually. BraidGL makes this exploration simple: we can toggle contributions to the surface colors in *couch* by specializing away code in the fragment shader stage and replacing texture lookups with `average`. Exploring these changes with traditional APIs would require carefully coordinating simultaneous changes to GPU and CPU code. In BraidGL, eliminating code that uses parameters from the CPU automatically removes the corresponding host-side code that sets up and communicates that data.

Toggling specialization options yielded a range of visual effects, including a low-detail, flat version and a medium-detail version (Figure 4g) that maintains most of the leatherness while preserving some performance benefit. Figure 5c shows the three versions' frame latencies, which range from 17.7 ms to 19.0 ms.

7 RELATED WORK

Static staging builds on prior work in multi-stage programming, language abstractions for placement, and GPU languages.

Multi-stage programming. Research on type systems for staged programming originated with MetaML [Taha 2003; Taha and Sheard 1997]. Our primary contribution is the distinction between splicing and materialization. Whereas traditional splicing inlines values into generated code as literals, our materialization performs inter-stage communication *without code generation.* Existing staged languages assume that the final "target program" will consist of a single stage; materialization, in contrast, lets application code efficiently coordinate multiple stages without invoking a compiler. Prior languages conflate the two communication modes [Hanada and Igarashi 2014; Kiselyov 2017].

Braid is most closely related to "heterogeneous" staged languages, where the meta-level language and quoted language differ, such as Terra [DeVito et al. 2013], variants of MetaOCaml [Eckhardt et al. 2007; Takashima et al. 2015], pluggable object languages in MetaHaskell [Mainland 2012], and Scala's lightweight modular staging [Rompf and Odersky 2010]. Quoted domain-specific languages also separate execution contexts: for example, database queries from application code [Cheney et al. 2013; Najd et al. 2016]. This prior work prioritizes code generation, so it lacks Braid's materialization concept for runtime communication between stages. Jeannie [Hirzel and Grimm 2007] is more

similar: it uses stages to represent glue between Java and C. We extend this basic idea for placement in heterogeneous hardware.

Several type systems address open code in dynamic multi-stage programming [Benaissa et al. 1999; Calcagno et al. 2003a, 2004; Chen and Xi 2003; Davies and Pfenning 1996; Kim et al. 2006; Moggi et al. 1999; Nanevski and Pfenning 2005; Taha and Nielsen 2003]. Braid restricts open code so the compiler may generate all code ahead of time. Future work could explore relaxing these restrictions while preserving AOT compilation.

Braid's staging-based macros resemble hygienic macro expansion systems [Flatt 2002, 2016; Kohlbecker et al. 1986; Lee et al. 2012]. Like MacroML [Ganz et al. 2001], Braid uses staging to define macros, but unlike MacroML, it defines them using syntactic sugar for function calls.

Because its annotations control the order of evaluation, staging works as a form of programmer-directed partial evaluation [Taha and Sheard 1997]. Our work focuses on domain- and hardware-specific optimizations over transparent specialization as in partial evaluation. It does not share the classic termination problems of partial evaluation because compilation runs in a single pass.

Placement abstractions. Language abstractions for code placement also appear in SPMD languages such as X10 [Charles et al. 2005] and Chapel [Chamberlain et al. 2007]. Their goal is to expose locality in distributed systems composed of nodes with roughly equal capabilities. The model is insufficient for single-node heterogeneous systems, such as CPU–GPU hybrids, where one unit is subordinate and units have wildly different capabilities and ISAs.

In ML5 [Murphy et al. 2007], a single distributed program expresses computations that execute on many different machines, and a type system with modal logic rules out unsafe sharing between places. Marking each data element with a place is a data-centric dual to Braid's code-centric placement annotations.

We give programmers explicit control over staged execution, whereas some systems *automatically* decompose monolithic programs into phases. In partial evaluation, *binding-time analysis* finds the parts of a program that can be computed eagerly when a subset of the inputs are available. Jørring and Scherlis [1986] define staging transformations as compiler optimizations, and $\lambda^{\mathbb{12}}$ hoists computation in a higher-order, two-stage language [Feltman et al. 2016].

Programming GPUs. The most common general-purpose programming environments for GPUs are CUDA [Nickolls et al. 2008] and OpenCL [Stone et al. 2010]. OpenCL uses a string-based API similar to OpenGL, while CUDA and other *single-source* GPU languages [Gregory and Miller 2012; Howes and Rovatsou 2016; OpenACC 2015; Sander et al. 2015] avoid the hazards of stringly-typed interfaces. Like OpenGL, however, these simpler non-graphics languages still lack safe metaprogramming tools for compile-time and run-time specialization. To help address this shortcoming in CUDA and similar languages, we plan to port Braid to a GPGPU backend.

Mainstream GPU shader languages directly reflect the structure of current real-time graphics hardware pipelines: programmers write separate kernels for each programmable stage [Advanced Micro Devices 2015; Apple 2016; Kessenich 2015; Microsoft 2008; Segal and Akeley 2016]. To communicate, each kernel declares unsafe interfaces that must align across all kernels. Even in modern APIs that use bytecode representations for shaders, such as Vulkan [Khronos 2017] and Metal [Apple 2016], host–shader interfaces cannot be statically checked.

In *rate-based* languages, type qualifiers describe different rates of computation in a single program that spans all stages of the programs' graphics pipeline [Foley and Hanrahan 2011; He et al. 2016; Proudfoot et al. 2001]. Rates play a similar role to stages in Braid: a compiler uses them to split the program into per-stage kernels. Similarly, *import operators* in Spire [He et al. 2016] are analogous to materialization expressions in Braid: they move data between rates. While Braid and Spire have similar goals, their approaches differ in important ways. First, while Braid's stages are always

explicit, Spire seeks to automatically infer its data-centric rate annotations. Hiding these details can make performance less predictable. Spire's `@Vertex` and `@Fragment` annotations can appear on declarations that are far from the computations involving the annotated variables, which can hinder reasoning about when and where a computation will run. Second, Braid's quotes are a closer match for the conceptual model in current graphics code, where shader stages are clearly delimited from each other and from host code. Porting legacy OpenGL code to BraidGL does not require radical reorganization. Finally, Braid focuses on defining general language constructs and semantics for heterogeneous placement and specialization. Spire focuses specifically on the GPU domain and does not include formal semantics.

Several experimental languages for writing shaders [Austin and Reiners 2005; Baggers 2017; Bexelius 2016; Boguta 2016; Elliott 2004; LambdaCube 2016; McCool et al. 2004, 2002; McDirmid 2009; Scheidegger 2011], like others for GPGPU programming [Klöckner 2014; Klöckner et al. 2012], tend to focus on dynamic code generation, whereas our work focuses on ahead-of-time optimization. Futhark [Henriksen et al. 2017] uses a pure-functional approach to describe data parallelism in GPU-GPUs. Lime [Auerbach et al. 2010] transparently targets both GPUs and FPGAs by adding constructs for data, pipeline, and task parallelism to Java.

8 CONCLUSION

We hope this paper leaves the reader with three main ideas:

- Heterogeneous programming languages need abstractions for *placement* and *specialization*.
- With extensions for static code generation, *multi-stage programming* can offer a foundation for both concepts.
- Current tools for *real-time graphics* are especially unsafe, verbose, and brittle. Our community has an opportunity to make graphics development less bad.

We see static staging both as a practical solution to the immediate problems with GPU programming and as a sound semantic foundation for emerging heterogeneous systems.

9 ACKNOWLEDGMENTS

This work owes a great debt to Yong He, Kayvon Fatahalian, and Tim Foley, who introduced the authors to the problems in graphics programming and discussed early ideas. Ömer Sinan Ağacan's blog post on the purpose of staging [Ağacan 2015] helped clarify the MSP landscape; Ömer also provided feedback on an early draft. Thanks to the entire RiSE group at Microsoft Research, where the project started. Thanks to Richie Henwood, Eric Lin, and Yiteng Guo, who have contributed recent improvements to the Braid compiler.

REFERENCES

Advanced Micro Devices. Mantle Programming Guide and API Reference 1.0. https://www.amd.com/Documents/Mantle-Programming-Guide-and-API-Reference.pdf.

Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

Apple. Metal Shading Language Specification, Version 2.0. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.

Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Chad Austin and Dirk Reiners. 2005. Renaissance: A functional shading language. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*.

Ömer Sinan Ağacan. Staging is not just code generation. http://osa1.net/posts/2015-05-17-staging-is-not-just-codegen.html.

Baggers. Varjo: Lisp to GLSL Language Translator. https://github.com/cbaggers/varjo.

Alan Bawden. 1999. Quasiquotation in Lisp. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*.

Zine-El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. 1999. Logical Modalities and Multi-Stage Programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*.

Tobias Bexelius. GPipe. http://hackage.haskell.org/package/GPipe.

Kovas Boguta. Gamma. https://github.com/kovasb/gamma.

Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

C. Calcagno, E. Moggi, and T. Sheard. 2003a. Closed Types for a Safe Imperative MetaML. *Journal of Functional Programming* 13, 3 (May 2003), 545–571.

Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *European Symposium on Programming (ESOP)*.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003b. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *International Conference on Generative Programming and Component Engineering (GPCE)*.

Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-specific Approach to Heterogeneous Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Chiyan Chen and Hongwei Xi. 2003. Meta-programming Through Typeful Code Representation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-stage Language for High-performance Computing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar Swadi, and Walid Taha. 2007. Implicitly Heterogeneous Multi-stage Programming. *New Generation Computing* 25, 3 (Jan. 2007), 305–336.

Conal Elliott. 2004. Programming Graphics Processors Functionally. In *Haskell Workshop*.

Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. 2016. Automatically Splitting a Two-Stage Lambda Calculus. In *European Symposium on Programming (ESOP)*.

Matthew Flatt. 2002. Composable and Compilable Macros: You Want It When?. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Matthew Flatt. 2016. Binding As Sets of Scopes. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*.

Tim Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. In *SIGGRAPH*.

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros As Multi-stage Computations: Type-safe, Generative, Binding Macros in MacroML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Kate Gregory and Ade Miller. 2012. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. O'Reilly. http://www.gregcons.com/cppamp/

Ilya Grigorik, James Simonsen, and Jatinder Mann. High Resolution Time Level 2: W3C Working Draft. https://www.w3.org/TR/hr-time/.

Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *International Symposium on Functional and Logic Programming (FLOPS)*.

Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers. In *International Symposium on Computer Architecture (ISCA)*.

Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. In *SIGGRAPH*.

Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. 2015. A System for Rapid, Automatic Shader Level-of-detail. In *SIGGRAPH Asia*.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

Martin Hirzel and Robert Grimm. 2007. Jeannie: Granting Java Native Interface Developers Their Wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Lee Howes and Maria Rovatsou. SYCL Specification. https://www.khronos.org/registry/sycl/.

Dean Jackson and Jeff Gilbert. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/.

Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*.

John Kessenich. An Introduction to SPIR-V: A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels. https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf.

Khronos. Vulkan 1.0.48: A Specification. https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf.

Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. 2006. A Polymorphic Modal Type System for Lisp-like Multi-staged Languages. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming (FLOPS)*.

Oleg Kiselyov. MetaOCaml – an OCaml dialect for multi-stage programming. http://okmij.org/ftp/ML/MetaOCaml.html.

Andreas Klöckner. 2014. Loo.py: Transformation-based Code Generation for GPUs and CPUs. In *International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY)*.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (March 2012), 157–174.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *ACM Conference on LISP and Functional Programming*.

LambdaCube. LambdaCube 3D. http://lambdacube3d.com.

Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. 2012. Marco: Safe, Expressive Macros for Any Language. In *European conference on Object-Oriented Programming (ECOOP)*.

Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Geoffrey Mainland. 2012. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

David Majda. PEG.js: Parser Generator for JavaScript. http://pegjs.org.

Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. 2004. Shader Algebra. In *SIGGRAPH*.

Michael McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*.

Sean McDirmid. Two Lightweight DSLs for Rich UI Programming. http://research.microsoft.com/pubs/191794/ldsl09.pdf.

Morgan McGuire. Computer Graphics Archive. http://graphics.cs.williams.edu/data.

Microsoft. Direct3D. https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx.

Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. 1999. An Idealized MetaML: Simpler, and More Expressive. In *European Symposium on Programming (ESOP)*.

Tom Murphy, VII, Karl Crary, and Robert Harper. 2007. Type-safe Distributed Programming with ML5. In *Conference on Trustworthy Global Computing (TGC)*.

Todd Mytkowicz and Wolfram Schulte. 2014. Waiting for Godot? The Right Language Abstractions for Parallel Programming Should Be Here Soon: The Multicore Transformation. *Ubiquity* (June 2014), 4:1–4:12.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-specific Languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*.

Aleksandar Nanevski and Frank Pfenning. 2005. Staged Computation with Names and Necessity. *Journal of Functional Programming (JFP)* 15 (Nov. 2005), 893–939. Issue 6.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53.

OpenACC. The OpenACC Application Programming Interface. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

Bui Tuong Phong. 1975. Illumination for Computer Generated Pictures. *Commun. ACM* 18, 6 (June 1975), 311–317.

Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2013. Portable Performance on Heterogeneous Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. A Real-time Procedural Shading System for Programmable Graphics Hardware. In *SIGGRAPH.*

Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Y. Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA).*

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM Conference on Programming Language Design and Implementation (PLDI).*

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE).*

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical Precision JIT Compilers. In *ACM Conference on Programming Language Design and Implementation (PLDI).*

Adrian Sampson. Braid source code, documentation, and interactive compiler. https://capra.cs.cornell.edu/braid/.

Ben Sander, Greg Stoner, Siu-Chi Chan, Wen-Heng Chung, and Robin Maffeo. HCC: A C++ Compiler For Heterogeneous Computing. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0069r0.pdf.

Carlos Scheidegger. Lux: the DSEL for WebGL graphics. http://cscheid.github.io/lux/.

Mark Segal and Kurt Akeley. The OpenGL 4.5 Graphics System: A Specification. https://www.opengl.org/registry/doc/glspec45.core.pdf.

Stanford. The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/.

John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Design & Test* 12, 3 (May 2010), 66–73.

Walid Taha. 2003. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23–28, 2003. Revised Papers.* Chapter A Gentle Introduction to Multi-stage Programming, 30–50.

Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL).*

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM).*

Naoki Takashima, Hiroki Sakamoto, and Yukiyoshi Kameyama. 2015. Generate and Offshore: Type-safe and Modular Code Generation for Low-level Optimization. In *Workshop on Functional High-Performance Computing (FHPC).*

Web Hypertext Application Technology Working Group. HTML Living Standard. Section 8.9: Animation Frames. https://html.spec.whatwg.org/multipage/webappapis.html.