An OpenGL Backend for Halide

by

Nicholas J. Chornay

S.B., Physics, Computer Science and Engineering, MIT, 2012

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

September 2013

Author: .....
Department of Electrical Engineering and Computer Science
September 6, 2013

Certified by:.....
Frédo Durand
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by:.....
Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

An OpenGL Backend for Halide

by

Nicholas J. Chornay

Submitted to the
Department of Electrical Engineering and Computer Science
on September 6, 2013, in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

High performance image processing requires not only an efficient underlying algorithm but also an implementation tailored to maximally exploit the available hardware resources. In practice, this requires low-level optimization, platform-specific instructions, and, when available, the use of special purpose hardware such as GPU. Halide is a domain-specific programming language targeted at image processing applications. Its programming model decouples an algorithm from the details of its execution, vastly simplifying development and optimization. We present an OpenGL backend for the Halide compiler, which enables Halide programs to run GPU computation on devices that support the OpenGL API. In particular, this paves the way for GPU computation on mobile devices using OpenGL ES. In doing so, we demonstrate how a general image processing framework can be built upon functionality designed for 3D graphics applications.

Thesis Supervisor: Frédo Durand
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank Jonathan Ragan-Kelley for his invaluable guidance and for many enlightening discussions over the course of this work. Thanks also go to Andrew Adams for his heroic work on the Halide compiler. I am thankful to them both for welcoming me into this project and into the group. I am amazed by what they have accomplished and excited to see where Halide goes in the future.

A huge thank you is also due to my advisor Frédo Durand for his advice and feedback over the course of research and writing. Frédo's fantastic courses on graphics and computational photography were some of my favorite classes at MIT and were what drew me into this research area; for that I am forever grateful.

I am deeply indebted to my parents; their support throughout my life has been a blessing. Nothing can adequately express my appreciation for effort they have put into my education and growth.

There are many people at MIT who have supported me over the 5 years I spent here. I thank them for making it a place I could feel at home. At this moment I am especially grateful to Stephanie Chang, who is turning this thesis in for me in my absence.

Contents

1 Introduction 9

2 The Graphics Pipeline 11

2.1 The Framebuffer 12

2.2 Vertex Operations 12

2.3 Rasterization 13

2.4 Per Fragment Operations 14

3 GPU Computation 15

3.1 Early Work 15

3.2 Recent Work 16

4 Background on Halide 17

4.1 Motivation 17

4.2 The Halide Language 18

4.2.1 The Algorithm 19

4.2.2 The Schedule 19

4.3 The Halide Compiler 20

4.3.1 Compilation Stages 20

4.3.2 GPU Compilation 22

5 The Halide OpenGL Backend **23**

5.1 A Motivating Example 24

5.2 The Halide OpenGL Runtime 25

5.2.1 Textures 25

5.2.2 Rendering 27

5.3 Code Generation 29

5.3.1 Function Definition 29

5.3.2 Loop Nests 30

5.3.3 Store Operations 31

5.3.4 Load Operations and Index Transformations 31

5.4 Texture Representation 32

5.4.1 Vectorization 33

6 Reflections **35**

Chapter 1

Introduction

Developing high performance image processing algorithms is challenging. Image processing pipelines are often deep and complex, and the optimal execution strategy is non-obvious and even unpredictable, requiring careful trade-offs to balance the competing needs for producer-consumer locality, parallelism, and minimizing redundant work. Exploring this large space of performance tradeoffs is time consuming, especially as the optimizations are tightly intertwined with the structure of the program. On modern systems, optimal execution must also take full advantage of hardware-specific features, ranging from vector operations on the CPU to highly parallelized GPU computation. Using these features, which vary from system to system, often adds an even greater cost to developer time, as a new API and possible even a new programming model must be learned.

The Halide programming language is designed to address some of these problems. The key idea of Halide is the separation of the intrinsic *algorithm*, the mathematical description of the function that is being computed, from the *schedule*, the manner in which the computation is executed on hardware. This detangling allows performance optimizations to be explored at a much higher level of abstraction and makes the process far less time consuming. Halide also supports many different compile targets, providing options for not only *how* an algorithm can be run, but for *where*. Porting an algorithm to another platform only involves modification of the schedule, and largely shields the developer from the semantics and syntax of the underlying libraries.

Of particular interest for image processing is mobile devices, which have evolved into a platform not only for image capture, but for image editing and manipulation as well. While the raw computing power on mobile devices has increased dramatically, both computation and battery resources are at a premium, making the case for efficient use of those resources even stronger.

One opportunity to better make use of available resources is to offload computation onto the device's GPU. GPUs are designed for computations that offer a great degree of parallelism, which we will see makes them suitable image processing algorithms. Prior to this thesis work, Halide supported GPU computation targeting the OpenCL and CUDA frameworks. Unfortunately these are not supported by the GPUs on mobile devices - Android and iOS devices only support OpenGL ES (OpenGL for Embedded Systems), a special subset of the widely used OpenGL API. In order to harness the power of mobile GPUs, Halide needs to be able to target OpenGL.

Unlike OpenCL and CUDA, which were designed with general-purpose GPU computation in mind, OpenGL is squarely[1] targeted at 3D graphics applications. It can still be harnessed for image-processing, but doing so is much more involved, as the algorithm must be translated into the language of textures and shaders the OpenGL is designed to support. Adding an OpenGL backend to Halide is valuable because it abstracts away these details, presenting a unified programming model across platforms.

Contributions

We make the following contributions:

- We present a **conceptual mapping** between the language of 3D computer graphics and those of Halide and image processing
- We detail the **representation choices** within this mapping
- We describe a **new Halide backend** that implements this mapping with the OpenGL API

[1] triangularly?

Chapter 2

The Graphics Pipeline

This chapter describes the graphics pipeline, the sequence of steps involved in generating a 2-dimensional image from a 3-dimensional description of a scene. The steps in the pipeline are not specific to OpenGL; rather the OpenGL API is designed to support them. Many of these steps are skipped over or vastly underutilized when OpenGL is used to do image processing, as described in Chapter 5, but a high level overview is valuable because it sets up the motivations for the sorts of operations that OpenGL supports, and introduces the vocabulary of computer graphics in a less fragmented manner.

The nature of the graphics pipeline also lends insight into the design of graphics hardware. We will see that almost all of these operations are relatively simple on their own, having little arithmetic and a small memory footprint, but also that they afford tremendous opportunity for parallelism, as there is no dependency between operations in a given stage. Graphics hardware is thus designed to support highly parallel computations, and is often capable of exceeding the performance of the CPU in such applications by a great deal. It has long been recognized that this hardware could be used for more than just setting pixel values in the framebuffer; some examples appear in the previous work discussed in Chapter 3.

The specifics of OpenGL are saved for the implementation details in Chapter 5. Interest in general-purpose computing on the GPU has motivated the development of GPU APIs less specifically targeted at the graphics pipeline. OpenGL is still probably the most widely supported

standard, though its deeper intertwinement with the model of the graphics pipeline make it somewhat more difficult for general computation.

2.1 The Framebuffer

The concern of the graphics pipeline is ultimately writing and reading values from the framebuffer, a memory buffer containing pixel data that will be shown on a display.

2.2 Vertex Operations

A model of a 3D object is built out of geometric primitives. Usually these are triangles, but they may also be points or lines. Each of these primitive objects is defined by one or more *vertices*. These vertices live in 3 dimensions, but to begin with they are defined in the coordinates of the larger object that they model.

To produce an image of the scene, we first wish to transform these vertices into the coordinate system centered on the virtual "camera" representing the view we wish to have on a scene. Usually this is accomplished through applying a series of matrix transformations to translate from object coordinates to world coordinates, from then from world coordinates into camera coordinates. The vertices may undergo a perspective transformation as well, which will make objects further away appear smaller, and depends on the field of view of the virtual camera. The other common option is an orthogonal projection, where a vertex's position in the camera's reference frame is exactly its 2-dimensional position, simply ignoring the depth.

Each of these vertices may also be then "lit". A scene without lighting will look flat. The effect of lighting is that it introduces a variation in color that depends on which direction the surface of a triangle is facing (the direction of its normal vector).

Besides lighting and transformations, other information may be calculated on a per-vertex basis. All of this is accomplished by a program called a vertex shader. Note that the per-vertex calculations are all independent of each other, which allows them to be processed in parallel.

A traditional graphics pipeline will also attempt to avoid wasted work by eliminating, or "culling" vertices that fall outside the view of the camera, either because they lie outside of the viewing frustum (the region of space visible to the camera) or because they lie on the back side

of an object (this can be determined from the direction of the surface normal, assuming that the object has a well-defined inside and outside). An object that has only some of its vertices living outside the field will see those vertices "clipped", meaning that they will be moved back to the edge of the field of view.

2.3 Rasterization

Rasterization is the mapping of primitive objects into fragments, or locations and color values on the 2 dimensional grid representing the output window. This involves two steps. For each primitive object, a set of corresponding fragments is determined. Then, for each of these fragments, a color and depth value is calculated. The color value is calculated by a program called a fragment shader.

Recall from the previous section that each vertex may have a number of parameters associated with it. In the fragment shader, each fragment sees values of these parameters interpolated from their values at those vertices. These values are used to calculate the color of the fragment. Like the vertices in the previous stage, the fragment colors may be computed entirely independently of each other.

To add more detail, the calculation of the color value may also involve looking up color values from *textures*, which are themselves 1, 2, or 3-dimensional bitmaps. These may be provided to the program, or the pipeline itself may be used to generate textures by rendering or copying into texture memory. A pipeline may be then run all the way through to render to a texture, and then use that rendered texture. This is way of introducing across-pixel dependencies and, among many other things, enables global effects to be applied to the scene before it is displayed, such as bloom effects for simulated HDR or edge detection to produce cartoon-like images.

We will see in Chapter 5 that the fragment shader is the part of the graphics pipeline that is most relevant to our work.

2.4 Per Fragment Operations

A pixel in the output window may be touched by multiple objects in the scene, and thus have multiple fragments that share its location. In these cases, the final color of the pixel in the framebuffer may be computed in a variety of manners, depending on the properties of the objects in the scene and any other desired effects.

The naive approach is to set the pixel value to the color of the last fragment that was processed and associated with that pixel. This will cause problems with overlapping objects, as they will be compete with each other and the fragments from the object in front will not necessarily win. In computer graphics, this problem of determining what is visible in a scene from the perspective of the camera is known as the visibility problem.

One way of addressing this is to draw the objects in a scene from back to front, overwriting old values. This is known as the painter's algorithm. It is rather inefficient, since work is required fragment values are computed for objects that are later hidden, and can have problems with correctness as well, since cyclic overlaps can prevent objects from having a well-defined depth ordering. The problem of additional work can be mitigated by using the reverse painter's algorithm, which sorts the objects in a scene from front to back, but many of the same issues remain.

The most popular solution is called depth buffering, or z-buffering. Each fragment has a depth value associated with it, and the framebuffer has extra bits at each location to store depth information in addition to a color value. A fragment only gets written into the frame buffer if its depth value is smaller than the depth value of the fragment already at that location. The depth can be compared before the possibly expensive color calculation, saving work by discarding the fragment early.

Similar algorithms are used to handle effects such as transparency and reflections; the framebuffer may contain additional bits per pixel specifically for masking and blending values.

Note that these per-fragment operations are separate from the fragment shader and are generally provided as *fixed-function* operations - they can only be configured, not programmed.

Chapter 3

GPU Computation

Computer scientists have long recognized the possibility of using graphics hardware for more than just storing and displaying pixels.

3.1 Early Work

In 1974 Wolfgang Strasser (and Edwin Catmull, independently months later) described the z-buffer algorithm for solving the visibility problem in computer graphics. Sutherland et. al. initially dismissed this approach (described in the last section of the previous chapter) as "ridiculously expensive", but the simplicity of the algorithm and ease with which it could be computed on hardware led to it becoming very popular.

In 1988, Fournier and Fussel took it upon themselves to formally explore the power of the framebuffer as a stream computation engine, particularly suited to algorithms with a large memory footprint that also exhibited parallelism and locality. They described the framebuffer in the language of automata theory - as a set of independent automata with limited memory processing read-only input tapes - and performed a disciplined analysis of its performance across various graphics algorithms.

3.2 Recent Work

Recent developments have addressed practical issues of GPU computation.

Adobe's Pixel Bender is a programming language for image processing algorithms with a syntax based on the OpenGL Shading Language (GLSL). It performs automatic runtime optimization across the CPU and GPU, and is intended for making image filters for Adobe products.

Apple's Core Image occupies a similar vein, supporting image processing on heterogeneous hardware, but targeted at OS X and iOS. It allows provided filters to be chained together, but does not optimize across filter boundaries. It also provides support for custom filters, though not on mobile devices.

Numerous tutorials for GPU image filtering exist on the internet. Frank Jargstorff of NVIDIA describes how to build a framework for image processing using NVIDIA's Cg programming language. He notes that modern GPUs can often surpass the power of CPUs in signal processing applications, and also mentions the awkwardness of dealing with an API originally designed for 3D applications as a motivation for creating the framework.

In 2012, Ragan-Kelley et. al. introduced Halide, an open-source, domain specific language and compiler targeted at image processing. The initial compiler supported GPU computation using NVIDIA's CUDA platform. Later versions added support for OpenCL, another framework targeting execution on heterogeneous platforms.

Chapter 4

Background on Halide

This thesis presents a new backend for the Halide compiler. The backend itself is described in the following chapter, but in order to provide context for that discussion, this chapter details Halide's programming model and the steps involved in compiling a Halide program. The 2 papers (listed in the references) that have been published on Halide provide a deeper look at it; this chapter is intended only to be an overview.

4.1 Motivations

Image processing algorithms tend to be structured in many-stage pipelines, with individual stages that repeat the same, relatively simple computation across a large number of pixels. This structure provides a vast domain of execution options. The complex interactions and dependencies between different stages of the pipeline mean that the most straightforward approach is unlikely to be anywhere near optimal, even if composed in a low level language such as C.

Optimized programs end up being both difficult to write and horrendously difficult to debug, as the original algorithm is quickly obfuscated by restructuring done in the name of optimization. Furthermore, as the optimal optimizations will likely change if an algorithm is to be used on a different platform, porting practically requires a complete rewrite. The creators of Halide were motivated by the belief that the underlying issue for both of these problems is that

the definition of the function being computed is inherently wrapped up in the details of its computation.

Halide is a domain-specific language embedded in C++ and targeted at high-performance image processing applications. Halide aims to address the issues mentioned above by decoupling the intrinsic *algorithm*, or *what* of the computation, from the *schedule* or the *how* of that computation. The programmer describes the algorithm functionally, and separately describes how the computation is to be carried out, in terms of when and how values should be computed, where they are stored, and whether they are reused or recomputed. The Halide compiler uses this schedule to translate the functional description of the algorithm into imperative machine code. Exploring the space of optimizations leaves the functional description of the algorithm intact, and only affects the schedule. Halide also supports the same algorithm being compiled for different platforms.

The authors of the Halide papers demonstrated that they could express complex algorithms in Halide with clean code that matched or exceeded state-of-the-art performance, and do so with much less effort on the part of the developer.

The remainder of this chapter discusses Halide in more detail. It draws a great deal from the two papers that have been published on Halide, but focusing on two specific goals. The first goal is to provide a high-level understanding of Halide's programming model; the second goal is to provide detail about the Halide compiler's implementation that is relevant to the discussion of the OpenGL backend in the following chapter.

Towards these goals, the discussion of Halide is split into two parts - the first covering how the Halide programming language defines an algorithm and its schedule, the second delving into how that definition is translated into machine code by the Halide compiler.

4.2 The Halide Language

The Halide programming language splits the definition of a function into two separate parts - the *algorithm*, and the *schedule*.

4.2.1 The Algorithm

Algorithms in Halide are defined functionally, in terms of arithmetic and logical operations as well as other functions. Images are treated as functions mapping from integer coordinates into scalar values. The domain of all of these functions is considered to be infinite, and a pipeline is usually described by a chain of functions.

The algorithm defines a value for each point in its (infinite) output domain. How the value is computed is defined separately by the program's schedule.

4.2.2 The Schedule

Image processing pipelines are characterized by multiple stages with complex dependencies. This leaves many choices for where and when a given function can be computed. There are inherent performance tradeoffs in this decision, which the Halide literature characterizes as being between 3 often-competing goals:

- **Locality** - how soon a value is used after it is computed. This affects memory bandwidth, as less recently written or used values fall further out into the cache or into main memory, becoming increasingly more expensive to access.
- **Parallelism** - how much the computation of different regions of a function can be separated.
- **Recomputation** - how much extra work is done, in the sense of function values being computed multiple times

One extreme in the scheduling space is to compute a function's value over its entire output domain before any of the results are used. This option, which Halide calls "root", avoids recomputation entirely, as the value at each location in a function's domain is calculated exactly once. The locality of this schedule choice is the least ideal, however, as each stage will tend to have a large memory footprint, and computed values will fall out of the cache before they are used. Even though each stage of the pipeline may be independently optimized, the performance

is constrained by memory bandwidth. This is the equivalent of chaining library functions together in a language such as MATLAB.

The other extreme of the scheduling spectrum is *inline*, in which function values are computed immediately before they are used, and afterwards discarded. This has fantastic locality, since values are consumed immediately after they are produced, and offers unparalleled parallelism, as there are no dependencies between the computations of values at different points in the output domain. However, with great parallelism and great locality often comes a great amount of recomputation, especially in pipelines where pixel values in one stage depend on many pixel values from previous stages.

In practice, the best approach usually lies somewhere in between - computing the entire pipeline in stages, but in smaller tiles, doing some extra work only at the boundaries and maintaining good locality and parallelism across separate tiles.

Schedules also provide options for the order in which values are computed over a given domain - dimensions may be traversed sequentially, in parallel, or unrolled or vectorized by a constant factor.

While all of these choices can be made in a low level, imperative C program, a change that is simple conceptually can affect the entire structure of the code. Halide decouples the algorithm from its schedule, and enables to the programmer to make these choices and explore the different options at a much higher level, without changing the underlying algorithm. The ugly details are handled by the compiler.

4.3 The Halide Compiler

The Halide compiler takes a Halide algorithm and schedule and produces imperative machine code. The same program can be compiled for different architecture targets, but the compilation process is mostly independent of this until the very end.

4.3.1 Compilation Stages

The compilation process is logically split into multiple stages that apply transformations to the internal representation of a pipeline and ultimately generate machine code.

Lowering

The first stage of the compilation process is called *lowering*. The compiler generates a set of loop nests, allocations, and load and store operations based on the algorithm and its schedule. The dependencies between stages are determined by working backwards from the definition of the output function.

At this stage in the process the dimensions of the allocations and the bounds of the loops are left as symbolic expressions that depend on the domains of the functions being computed and stored.

Bounds Interference

The next stage calculates the domain over which each function must be evaluated. This is done by evaluating dependencies, working backwards from the output function.

Storage Folding

Parts of the computation may be scheduled to compute a sliding window, where only the last N that were computed are required by a given loop iteration. Rather than allocating storage for the entire domain of the intermediate function being calculated, space need only be allocated for the last N values. This memory reuse saves space and also bandwidth.

Vectorization and Unrolling

Loops of a constant size are replaced with either vector operations or with their unrolled equivalent.

Backend

After a pass is made through the code to simplify expressions and eliminate dead code, most of the work of translating the final imperative internal representation into machine code is passed off to LLVM, an open source compiler infrastructure. The Halide compiler makes some platform-specific optimizations that LLVM might miss, for example using Intel SIMD instructions on x86 systems, or recognizing opportunities to perform dense vector loads over clamped buffer indices.

4.3.2 GPU Targeted Compilation

The stages described in the previous section are exactly the same for code intended to be executed on the GPU. Loop nests corresponding to GPU computation are annotated as such, and their indices are set to correspond to block and thread indices.

The notion of blocks and threads comes from NVIDIA's CUDA platform. With CUDA, a function, or kernel, is executed in parallel by a number of threads, which are assigned x, y, and z indices (they can be thought of as existing on a multi-dimensional grid). Threads are grouped into blocks. These blocks are limited in size, but resources can only be shared between threads in the same block. Halide also supports another GPU backend, OpenCL, which has a similar model. Unfortunately these constructions are less applicable to OpenGL, and make some aspects slightly more difficult.

When the compiler encounters one of these annotated loop nests, it does a number of things. It generates a closure of the program state required by the loop. It compiles the loop body into a kernel for the target device. Finally it determines and inserts the required runtime calls to transfer data to and from the device and execute the kernel. More concrete detail about this is provided in the following chapter.

Halide's GPU backends support a clean expression of complex pipelines optimized for heterogeneous platforms.

Chapter 5

The Halide OpenGL Backend

This chapter presents the Halide OpenGL backend that was developed for this thesis, and discusses details of its implementation. All of Halide's GPU backends are split logically and functionally into two parts - the runtime module and the code generator. The runtime module is compiled separately and linked against the halide program; it provides an abstract interface for copying data to and from the GPU and for initializing and executing kernels. The code generator does its work at compile time, translating Halide's internal representation of a function - a set of imperative expressions inside a loop nest - into a kernel that can be run on the device, in this case, an OpenGL Fragment Shader.

For the runtime, the focus of the description here is on the conceptual mapping between OpenGL constructs and the functions that the Halide runtime module is required to provide. Where relevant, details of OpenGL and the inner guts of Halide are introduced; the hope is that these can be understood and appreciated given only the knowledge from previous chapters.

The code generator is left to handle some more of the details and mismatches between the internal language of Halide and the language of shaders. This section is presented at a more practical level, giving the challenges that arise and the solutions that were chosen.

In both sections, the similarities and differences between Halide and OpenGL's perspectives on the world provide a deeper understanding of both systems, and reveal what makes this project interesting, challenging, and worth discussing to begin with.

5.1 A Motivating Example

Suppose we have a function that we want to compute on the GPU. For example, this 2-dimensional function `g(x, y)`, which is defined in terms of a 2-dimensional input function `f(x, y)` some scalar value bar.

```
g(x, y) = bar * f(x, y) * f(x, y);
```

Assume that at this point in the pipeline, the value of function `f` will have already been computed over its entire domain (on the CPU) and the results will have been stored in an appropriately sized buffer (in CPU, or "host" memory). In the language of Halide schedules, this corresponds to the function `f` having been scheduled to run as "root". The value of `bar` also will have also been calculated somewhere (it is not a compile-time constant; otherwise Halide would replace the variable with its value).

The question is: what needs to happen in order to schedule and run the computation of `g` on the GPU?

- The description of the computation itself (load value of `f` at location `(x, y)`, square it, and multiply by the value of `bar`) must be translated into a kernel in the language of the device. This is the job of the code generator and is done at compile time.
- The buffer storing the values of `f` must be copied into GPU memory so that the computation of `g` can access them.
- Any arguments required for the computation of `g` also need to be given to the device. One of these arguments will be the value of `bar`; others may be necessary as well depending on the form of the kernel that was generated by the code generator. Practically, the dimensions of the input and output buffers are required, for reasons that will be discussed later on.
- Device memory must be allocated for the output.
- The appropriate kernel must be fetched, passed the appropriate arguments, and summarily executed.

- The result of the computation, which now lives in device memory, must be copied back to CPU memory so that other functions may access it. If the values of g were only going to be used by other functions executing on the device, this copy would not be necessary and would be skipped.
- Any buffers of data no longer required should be freed.

Before this project, Halide already had existing infrastructure to make these decisions about what needs to happen when - what data needs to be copied where, when it can be discarded, etc. It is up to the backend to provide the functions that take the appropriate actions for each target. These functions constitute the *runtime*.

5.2 The Halide OpenGL Runtime

The OpenGL runtime provides a set of functions for allocating and freeing device memory, copying data between the device and host, and initializing and executing kernels on the device. This section provides details about the OpenGL constructs involved and how they are used to provide this functionality

5.2.1 Textures

The analogue of buffers on the GPU is textures. Recall that textures are bitmaps usually used to add detail to 3D objects, and are referenced by the fragment shader (in OpenGL, using Sampler inputs). Textures work as a buffer analogue because, not only can they be sampled by a shader, but they can themselves be rendered into, by being attached to a framebuffer as the color attachment (more details in the next section).

The key differences between a texture on a GPU and a buffer on the CPU is that a texture is inherently typed and multidimensional. On the CPU, memory is a single-dimensional array of bytes. Neither the type of data nor its dimensionality need be known by the memory allocation - only the total size, which depends on the size of an element and the total number of them to be stored. A "multidimensional" buffer index is always converted to a single-dimensional offset

into the byte array based on the extents of the different dimensions represented and their arrangement in memory.

Most of the time, textures are 2-dimensional, meaning that they are accessed using a pair of coordinates, and allocated with width and height parameters. More accurately, these textures are 2-dimensional arrays of *vectors*, because each location stores up to 4 separate values (Luminance textures store a single value at each location; RGBA (Red, Green, Blue, Alpha) textures store 4 values at each location). OpenGL does support 1-dimensional textures as well, but the sizes of these are generally much more constrained (in either case, the maximum size is hardware-specific, though it must respect minimums set by the OpenGL standard).

Textures also have an associated data type - either float or integer - that is set when the texture is created. In this sense, texture allocation is not like CPU memory allocation, because it requires type rather than size information.

The final difference between textures and buffers is that textures are accessed using *normalized coordinates*, which range from 0 to 1 regardless of the pixel dimensions of the texture. Since these coordinates are continuous, but the texture itself still consists of individual pixels, one might ask how an index into a texture is converted into a color value. The required indexing gymnastics are primarily the concern of the code generation module, but the runtime texture allocation function makes the decision about how those indices are treated.

There are two options for texture indexing. One is to return the value of the texel (texture pixel) that is nearest to sample point (in Manhattan distance). Texels are identified by the location of their lower left corner, but distances are calculated with respect to the *center* of a texel. Thus, in texture coordinates, the center of the lower left texel is located at the position `(1/(2*w), 1/(2*h))`, where `w` and `h` are the width and height of the texture, respectively.

The second option for texture indexing is to use linear interpolation, which returns for a given sample point the weighted average of the nearest 4 texels (again measured by Manhattan distance). This is useful for smoothing a texture when it is sampled at a higher resolution than that of the texture itself.

The code generator tries to make sure that textures are always sampled at the center of a pixel, in which case the two indexing options should return the same value. To be safe and simplify things, the "nearest" option is used in practice.

Textures also have options for handling requests for values at locations that lie outside the bounds of the texture. This choice should not matter, since Halide should ensure that buffers are sufficiently sized. For completeness, the texture is configured to use clamp all values to the edge (this can be thought of as using the "nearest" option even for points lying outside the texture boundary). Other options include wrapping the texture coordinates by just considering the fractional part.

It is not uncommon for image processing pipelines to upsample images or clamp accesses to the edge of an image. The exploitation of OpenGL's built-in functionality for these operations is unexplored.

Outside of the issues raised here, the runtime functions for handling textures are straightforward.

5.2.2 Rendering

The computation of the output function itself employs a heavily abridged version of the graphics pipeline. Much of the work associated with traditional 3D graphics pipelines - transformations, clipping, lighting, depth buffering - is unneeded and happily skipped over.

At a high level, the rendering stage renders merely renders a pair of triangles that form a rectangle and cover the entire viewport. The rendering target is an application-created framebuffer that has the texture that has been allocated for the output as its color attachment. This allows the data to be used by later stages of the Halide pipeline on the GPU, or be easily copied back to host memory.

Not only does the vertex shader have very few vertices to shade, it has very little work to do per vertex. Unlike textures, whose coordinates range from 0 to 1, OpenGL defines the 2 dimensions of the viewport to range from -1 to 1. The vertex shader converts these values into pixel locations (ranging from 0 to width and 0 to height), which get passed to the fragment shader as *varying* variables. These values are interpolated across objects from their values at the vertices; in this case, each fragment sees a value corresponding to its unnormalized location. The vertex shader is the same for all functions and is defined in the runtime source itself; the differences between functions all lie within the definition of the fragment shader, whose source is generated by the code generator discussed in the following section.

In OpenGL Shading Language (GLSL), fragment shaders deal with two types of variables. As mentioned above, *varying* variables are interpolated from their values at the vertices of an object associated with a given fragment. In traditional graphics pipelines, these variables are often used for information such as color or surface normals. In the fragment shaders that Halide uses with the OpenGL backend, the only varying variable that the fragment shader uses is the pixel location. All other variables are *uniforms*. In OpenGL, these have the same value everywhere inside a primitive; in this pipeline, they have the same value everywhere.

The source for all required fragment shaders for a given Halide pipeline is generated at compile time. The runtime provides an initialization function that is called once before any other runtime functions are called. The initialization function for the OpenGL backend compiles the fragment shader sources generated by the code generator, and combines them with the vertex shader to form OpenGL *program objects*. These programs are stored in a map indexed by the kernel name, so that they can be fetched easily later on.

The mechanics of running an OpenGL shader are different from those of a, say, a function call in C. Functions in C (and, for that matter, OpenCL and the other backends used by Halide) take an ordered array of arguments (corresponding to values being pushed onto the stack). For OpenGL shaders, the values of uniform and varying variables are set by name with a special function that also depends on the type of the variable. Furthermore, not necessarily all of the variables declared in the shader source are expected when the shader is executed - the GLSL compiler eliminates arguments that do not get used. Thus, the runtime kernel execution function requires names for each of the arguments that get passed, and must check whether the referenced variables are in use before setting their values.

At this stage in the Halide compiler, unused variables will have been eliminated. It is still the case that the shader compiler may optimize away variables, and why this happens provides a useful insight. In short, it is because the location that each invocation of the fragment shader writes to is implicit.

As was mentioned in Chapter 4, and will be discussed in more detail in the following section, Halide translates functions into explicit, nested for loops that together loop over the entire output domain, or set of indices into the output buffer. Whatever other calculations may be involved (even if the output is simply being set to a constant value), every iteration of the

innermost loop of any function must calculate the index into the output buffer that it writes to. This calculation involves the loop indices and the extents of the loops.

In OpenGL, this loop occurs implicitly over the pixel locations touched by the primitive objects being rendered. A given invocation of the fragment shader can always find out its output coordinates, but it does not have to know them in order to write a value. Coordinates do have to be specified for a fragment shader to read from a texture value, and usually these coordinates are related to the coordinates of the fragment, but the output coordinate itself is not explicitly used. The color of the fragment is set by the fragment shader in the special, appropriately named variable `gl_Fragcolor`.

Furthermore, which texture the output value is written to is itself implicit; it depends which texture is bound to the current framebuffer as a color attachment. The OpenGL runtime must identify for each kernel what the name of the output is, and make sure that the texture corresponding to that argument gets bound appropriately.

5.3 Code Generation

The second half of the Halide OpenGL backend is the code generation module. It translates a series of imperative Halide expressions into fragment shader source code written in the OpenGL Shading Language (GLSL). This shader source is compiled by OpenGL inside the runtime initialization function.

GLSL syntax is based on that of C, so much of the less interesting syntactical work can be delegated to the C code generation module Halide already has. There are, however, some non-trivial syntactic and functional differences between the GLSL and C; this section focuses on those differences.

5.3.1 Function Definition

All OpenGL shaders are described by a main function that takes no arguments and returns void. For fragment shaders, the inputs are declared as global variables (outside of the function body), and may be either *uniforms* or *varying* variables. Recall that uniform variables have the same

value at every fragment that corresponds to a given object, while varying variables are interpolated from their values at the vertices that define that object.

The only varying variable required by the kernel is a pair of pixel coordinates (calculated by the vertex shader, which is the same for every kernel). This variable has a known name and is automatically included in the source of every fragment shader.

The remaining arguments to the shader are, naturally, uniforms. The Halide GPU infrastructure determines the set of arguments required to run a kernel on a device; these arguments are provided to the code generation module. Scalar arguments keep their types. Buffer references acquire the type "Sampler2D", which is OpenGL's way of referencing textures. For each buffer argument, a vector argument is added to hold the dimensions of that buffer. The dimensions will be useful for translating to and from normalized texture coordinates.

5.3.2 Loop Nests

Recall that Halide uses a function definition in tandem with its schedule and output domain to build an internal representation - a set of loop nests - that describe the computation imperatively. It is this internal representation that is passed into the appropriate code generation modules.

For GPU targets, the set of loop nests is fairly straightforward. It is split into loops over "block" and "thread" indices (think of blocks as rectangles covering chunks over the output domain, and threads as indices inside these blocks). This thread and block structure is related to models of computation used by other GPU backends; for the purposes of the OpenGL backend it suffices to think of the block and thread loops in each dimension as being paired up and simply looping over the entire extent of that dimension.

The function itself is computed in the innermost loop nest. As noted previously, at the very least, the loop indices and extents are used to calculate the 1-dimensional offset into the output buffer at which the computed value of the function is stored. There is always exactly one of these storage operations per loop, and each location in the output domain is written exactly once.

For an OpenGL fragment shader, the loop over output locations becomes implicit - the shader describes a functional mapping from the output location to output value, and is executed

over the entire output domain. Shaders are somewhat reminiscent of the functional definition of an algorithm used by Halide's front end.

If the loop indices were only used to calculate the output location, the fragment shader would be able to ignore them completely, since the output location is implicit in each invocation of the fragment shader. However, the same loop indices are often used elsewhere in the function computation, either explicitly or in the calculation of indices for loading values from other buffers.

The solution implemented in the OpenGL code generator is to, at the beginning of the function definition, work backwards to calculate what these index values would have to be in order to correspond with the already-known output location. Once these are calculated, they can be used normally elsewhere.

All of this index translation may seem like extra work, since often the 2-dimensional index itself is an intermediate in the calculations based on the loop index values. A good GLSL compiler should be able to recognize this, since the values involved in the translation are known at the time of compilation.

5.3.3 Store Operations

The store operation itself is quite painless, since, as the previous section described, the address (or index) is implicit. The value to be stored is simply assigned to the special variable gl_Fragcolor, which is a 4-element vector. Depending on whether a single value or vector of values is being stored, only the first element or the entire vector may be used.

5.3.4 Load Operations and Index Transformations

In Halide's internal representation, load operations have a single dimensional index, regardless of the number of dimensions represented by the data. In C code, a load would correspond to an appropriately type single-dimensional array access. In OpenGL, with buffers represented by textures, a load operation is accomplished as a texture lookup (using the texture2D function).

Texture lookups are inherently 2-dimensional, and use *normalized coordinates*. The single dimensional load index must be transformed before it can be used, taking both of these into account.

Recall that normalized texture coordinate range continuously from 0 to 1, regardless of the width or height of the texture. Recall also that the center of pixels is considered to lie at *half-integer coordinates*, so, for example, the pixel at (0, 0) has its center at (½, ½). For unambiguous texture accesses, we would like to access a given texel at its center. Thus, to convert from 2-dimensional pixel coordinates to texture coordinates, an offset of ½ is added to the x and y coordinates, after which they are divided by the width and height of the texture, respectively.

The calculation of the (x, y) position from the single dimensional load index also involves the dimensions. The x position is the single dimensional index modulo the width of the texture, while the y position is the floor of the index divided by the height.

This is extra work, though in many cases the compiler will be able to make optimizations to avoid going back and forth repeatedly. A possible advantage of this approach of having the single dimensional intermediate is that it decouples the dimensions of the function from the dimensions of the texture, making it possible to treat the texture as having an arbitrary number of arbitrarily-sized dimensions. This is an area for future development.

5.4 Texture Representation

As the previous sections have shown, much of the complexity of the OpenGL backend stems from the use of textures to store buffers on the GPU. The normalization of the indices used for texture lookups is the easiest to deal with; the fact that 2 indices are used rather than 1 makes loading values more complicated.

These concerns cause some hiccups with the compiler, but they are at least a property of the algorithm and the size of its output domain. There is a final aspect of textures that makes the separation between the algorithm and the schedule not clean at all. It has to do with the *format* of the texture - namely, whether the texture is a luminance texture (1 value at each location) or an RGBA texture (a vector of 4 values at each location).

5.4.1 Vectorization

The CPU is indifferent towards whether values in memory will be accessed as single values or in vectors of multiple values (through SIMD load operations). OpenGL is not. Each texel stores anywhere from 1 to 4 values (these would be described as Luminance and RGBA textures). The number of values per location is set when the texture is created.

Suppose we have an array of values representing a greyscale image with a given width and height. When copying it to the GPU, we have the option of treating it as a luminance texture (1 value at each location, texture dimensions are width and height), which relates most closely to the actual image format. When we load the image into device memory as a texture, we also have the option of treating that same buffer as an RGBA texture (4 values at each location). The RGBA version of the texture has a quarter the number of texels that the luminance texture does, so its width or height must be divided by 4 to account for this.

What makes textures strange in this case in a way that memory buffers on the CPU would not be is that the choice of the representation is closely tied to the choice of how computation is to be executed (defined by the Halide *schedule*). On the CPU, whether the computation will be vectorized is practically irrelevant to the memory allocation function.

The simplest solution is to only allow for 2-dimensional luminance textures, where each index stores a single value. This matches most closely to the model of a buffer on the CPU, and is what the backend does at the time of this thesis being written. The problem with this method is that it wastes much of the power of OpenGL to do vector computation at each texel location.

A more efficient solution would be to always treat textures as RGBA, and either vectorize or unroll the loop by 4, depending on the schedule. This adds complexity, but seems more reasonable in the long term.

In any case, the choice should be consistent everywhere. It cannot be dependent on the schedule, because the format of a texture is fixed. A RGBA output of one function cannot be a Luminance input for another one.

Chapter 6

Reflections

This thesis has presented a backend for Halide that uses the OpenGL API. We were able to accomplish much of what we set out to do, but there is still plenty of room for exploration and improvement.

Challenges arose from both OpenGL and Halide itself. For OpenGL, the main task was to figure out the mapping between OpenGL operations and the set of required Halide runtime operations. With Halide, we discovered that OpenGL had quite a different view of the world from those of OpenCL and CUDA, and that we were not able to reuse as much of those backends as we thought we would be able to. OpenGL requires more information to be stored and additional arguments for some runtime functions, such as data types for buffer allocations and argument names for kernel execution. The arguments names were a straightforward addition; data types for buffer allocations is still to be done, as the information exists, but multiple levels removed from the runtime allocation function. These challenges suggest that a larger rethinking of some parts of the backend is in order to better pass more useful information to later compilation stages.

The OpenGL backend could benefit from explicit performance testing. We make some assumptions about what would and would not be efficient, but these assumptions should be verified.

Our ultimate goal with the Halide OpenGL backend is to support GPU-based image processing on mobile devices. The system we have presented here has been only tested on desktop platforms, however it is based entirely on the OpenGL ES 2.0 API, so transitioning it to mobile platforms will require only minor tweaks.

Bibliography

A. Fournier and D. Fussell. 1988. On the power of the frame buffer. ACM Trans. Graph. 7, 2 (April 1988), 103-128.

COREIMAGE. Apple CoreImage programming guide. http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/CoreImaging.

E. Catmull, A Subdivision Algorithm for Computer Display of Curved Surfaces, PhD dissertation, Tech. Report UTEC-CSc-74-133, Dept. of CS, Univ. of Utah, Dec. 1974.

F. Jargstorff et. al. GPU Gems. https://developer.nvidia.com/content/gpu-gems-chapter-27-framework-image-processing

I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," Computer Surveys, Vol. 6, No. 1, Mar. 1974.

J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph., 31(4), 2012.

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In PLDI, June 2013.

M. Woo, J. Neider, and T. Davis. The OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1. Addison-Wesley Publishing, Jan. 1997. http://www.glprogramming.com/red/chapter03.html

OPENGL, 2010. OpenGL ES Common Profile Specification Version 2.0.25. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

PIXELBENDER. Adobe PixelBender reference. http://www.adobe.com/content/dam/Adobe/en/devnet/pixelbender/pdfs/pixelbender_reference.pdf.

W. Straßer. Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten, Dissertation, TU Berlin, submitted 26.4.1974