Programmable Graphics Pipelines

By

Anjul Patney
B.Tech. (Indian Institute of Technology Delhi) 2007
M.S. (University of California, Davis) 2009

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

John D. Owens, Chair

Nelson Max

Kent D. Wilken

Committee in Charge

2013

To my parents, Nirmal and Anil …

Contents

List of Figures

List of Tables

List of Algorithms

Abstract of the Dissertation

Programmable Graphics Pipelines

Programmability of graphics pipelines has proven to be an extremely valuable tool in the hands of shader authors and game programmers, and is one of the driving forces behind the ubiquity of high-quality visual content across the computing ecosystem. This dissertation presents the design, implementation, and evaluation of an abstraction that extends the programmability of graphics systems from individual shaders to entire pipelines, and addresses the performance, programmability, and flexibility of the resulting implementations.

To this end, we propose Piko, a high-level abstraction for designing flexible and efficient pipelines for computer graphics applications. Piko extends traditional pipelines by allowing each stage to consist of three phases, and uses 2D spatial tiling to help express information about not just the algorithm, but the granularity and schedule of computation as well. Having these different programs address different implementation concerns helps optimize a Piko pipeline for multiple scenarios, such as varying application inputs and target platforms.

The focus of this dissertation is twofold. First, we motivate the importance of efficient programmable pipelines, and survey existing graphics systems to explore implementation practices that help improve their efficiency. Specifically, we identify concurrency, locality, and fine-grained synchronization as important goals, and explore design practices that help achieve them. Second, we propose an abstraction to embody these practices as first-class constructs. We present a detailed study of how pipelines expressed using Piko could translate to two different architectures, study the behavior of resulting implementations, and compare it to the state of the art in high-performance graphics.

Acknowledgments

Six years ago I nervously made one of the biggest decisions of my life—to pursue a Ph.D. at UC Davis. Today I cannot imagine having a more successful and fulfilling graduate school experience. I would like to thank my adviser John D. Owens for making it so by being a constant source of academic as well as personal guidance. Throughout my graduate studies, he expected nothing short of the highest quality from my work, yet provided ample freedom for me to make tough decisions, including some wrong ones, in the face of research challenges. John was also a true supporter during my weaker moments as a Ph.D. student, and never hesitated to stand up for my interests. If in the future I get to become a professor, I will try to emulate John's unique style.

I would also like to thank Prof. Nelson Max and Prof. Kent Wilken for serving on my qualifying and graduation committees, and for providing their diverse insights to help me prepare and conduct the research in this dissertation. It is my honor to have such esteemed researchers evaluate my work. In my career, I hope to follow in their footsteps.

While working towards this dissertation, we had the rare privilege of receiving advice from several experts from the industry, without whose help it would have been nearly impossible to pursue such an ambitious project. In particular, I am thankful to Tim Foley, Matt Pharr, Mark Lacey, Aaron Lefohn, Chuck Lingle, and Jonathan Ragan-Kelley for going out of their way to take part in numerous discussions and provide guidance in making progress towards our research goals. I would also like to thank Justin Hensley who helped obtain the AMD Mecha model for use in our experiments.

I wish to thank Mohamed Ebeida and Scott Mitchell for allowing me to participate in solving a truly intriguing set of problems. What started as a chance collaboration quickly grew into a key project with several researchers, and I am glad to have participated in that process.

I am fortunate to have received financial support throughout my graduate studies. In making that possible, I would like to thank the Department of Energy (SciDAC Institute for Ultra-Scale Vi-

sualization DE-FC02-06-ER25777 and the Energy Early Career Principal Investigator Award), the National Science Foundation (grants CCF-0541448 and CCF-1032859), the NVIDIA and Intel Ph.D. Fellowship Awards, the Intel ISTC-VC grant, a UCD Summer GSR award, and a gift from AMD. Intel and NVIDIA also helped provide valuable experience through excellent internships, and I would like to thank my internship mentors Rob Cavin (Intel), Bill Dally (NVIDIA), Steve Keckler (NVIDIA), and Michael Shebanow (NVIDIA).

As a graduate student I spent a majority of my waking hours in the lab. Fortunately, I got to work alongside some really smart and friendly labmates. It was a blast hanging out with Stanley Tzeng and Andrew Davidson who were not only excellent collaborators on research projects, but great dinner buddies and teammates in Heroes of Newerth as well. Much of the work in this dissertation is the result of efforts by Stanley and Kerry Seitz. I am thankful for such awesome project partners. I have always looked up to Shubho, Pinar, and Yao for great advice, and had a wonderful time interacting with Ritesh, Kerry, Jason, Yangzihao, Edmund, Afton and Calina. I will miss you guys!

Lastly, I wish to thank my family. My parents let me travel across the world to get a Ph.D., and even though I was only able to visit only once a year, they encouraged and supported me unconditionally. My sisters Apekshikha and Aditi also showed immense patience during my education, and helped me get through some very tough times. I owe my girlfriend Divya a special thanks for tolerating a hopelessly unavailable boyfriend. I owe all my successes to my strong and loving family.

Thank you everyone. I wouldn't change anything about how you shaped my experience in graduate school.

Chapter 1

Introduction

Visual experiences are important to us. We constantly rely on our eyes to sense and perceive the world around us. Unsurprisingly, in today's technology-enabled society, our favorite gadgets go to great lengths in producing and consuming visual stimuli. We cherish our high-resolution cameras, big-screen TVs, and snappy video games. High-performance computer graphics is a central part of modern computing.

The hardware graphics processing unit (GPU) is responsible for graphics in a modern computer. It works in close harmony with carefully optimized and specialized software. The pervasiveness of such specialized solutions indicates that not only is visual computing valuable, it is also hard and computationally expensive. While the first GPUs were simple coprocessors that helped offload a CPU's graphics-specific operations, they rapidly evolved in performance as well as flexibility. Today's GPUs can rival and in many cases surpass CPUs in their computational capabilities. Despite this progress, however, applications continue to offer larger and more complicated workloads with aggressive time and power budgets.

In contrast to real-time graphics, offline graphics applications render high-quality images with a more generous time-budget. These applications also require extremely efficient systems. For this class, a single computer-generated image can take multiple hours to render [24], and a full-length movie hundreds of thousands of computing hours. Contemporary platforms for

offline graphics consist of large clusters of powerful CPUs, often paired with GPUs, and highly optimized software.

The performance of both real-time and offline graphics systems is complemented by their flexibility through programmable shaders, which allow a programmer to define logic to programmably compute geometry, materials, illumination, etc. The advent of programmable shading in computer graphics has resulted in a wave of new techniques that extend the capabilities of graphics architectures, without undue penalties in performance.

While individual parts of a single graphics system are programmable via shaders, the underlying system architecture is often hardwired for performance reasons. This rigidity affects the flexibility of target applications, because they are bound by the architecture's constraints. Programmable shading has reached a point where modern graphics processors are largely general-purpose massively-parallel processors. Yet, graphics programming models maintain rigid pipelines as a baseline, compelling programmers to implement novel graphics algorithms that involve either reconfiguring or bypassing existing infrastructure [4, 8]. There is a need for graphics abstractions to grow from shaders to entire pipelines, allowing targeting entirely new algorithms and techniques to high-performance architectures.

In summary, visual computing represents an important facet of modern technology. At the same time, it is a hard problem with a diverse set of application domains and a growing need for flexibility, speed and efficiency. It is thus a continuing research endeavor to investigate the design of systems for computer graphics.

To aid in this effort, we propose *Piko*, an abstraction to aid in the design of graphics systems that are both flexible in their construction, and efficient in their performance. By embracing *2D spatial tiles* as the fundamental unit of computation, Piko helps express graphics pipelines in a form that is intuitive to a programmer, but at the same time exposes a degree of concurrency that can sufficiently utilize today's parallel hardware.

1.1 Contributions

This dissertation makes the following contributions to the field of graphics systems:

- We design and implement a novel abstraction for graphics pipelines. Our abstraction leverages *spatial binning*, which we demonstrate to be a critical optimization for high-performance rendering. Spatial binning or tiling helps a graphics application expose parallelism, while at the same time providing an opportunity to preserve spatial as well as temporal locality. Since the method of expression of spatial binning varies widely across implementations, our abstraction allows the tiling policy to be programmable on a per-stage basis. The proposal of using programmable spatial binning as a core construct is our first contribution.

- In order to be able to express tiling policies without rewriting an algorithm and to be able to re-target a pipeline to varying architecture configurations, we propose to partition each pipeline stage into three *phases*: AssignBin, Schedule, and Process. Together, these phases allow us to flexibly exploit spatial locality and enhance pipeline portability, and consequently help achieve our goal of designing flexible and efficient graphics pipelines.

- Finally, we provide a detailed strategy to identify and exploit opportunities for obtaining optimized implementations given a pipeline description. With a target architecture in mind, we are able to recognize several key structural constructs within and between phases of a pipeline, and map them to implementation-level optimizations.

1.2 Outline

To start, we motivate the problem in Chapter 2 by providing empirical evidence for the importance of pipelines in computer graphics, and discuss the complexity of designing efficient implementations for graphics pipelines. We also present related work in the pursuit of alternative abstractions for graphics pipelines.

In Chapter 3, we present a discussion on the efficiency of implementations of graphics pipelines,

identifying the importance and the consequent preponderance of spatial binning (tiling). We then provide an overview of our abstraction, Piko, within the context of spatial binning, in Chapter 4, where we also cover the process of expressing a graphics pipeline in Piko, and some examples for common pipeline designs. Finally in Chapter 5, we present the implementation strategy of a pipeline expressed using our abstraction. Chapter 6 concludes this dissertation through a discussion of positive inferences and limitations of our exploration, and presents some interesting directions for future work.

Chapter 2

Pipelines as Abstractions for Computer Graphics

2.1 Introduction

Literature and API standards often express graphics algorithms and techniques through directed graphs, or pipelines—a design abstraction that partitions the complicated interplay of data and computation into well-defined interactions between distinct functional modules. Pipelines are easy to comprehend, and efficient implementations, especially those in hardware, are often organized analogously. In fact, early work in computer graphics suggests that the algorithmic abstraction is itself a reflection of pipelines as a common implementation philosophy in traditional graphics hardware [50]. Figure 2.1 shows an example pipeline comprising of a sequence of three stages A, B, and C, connected via inter-stage queues to carry intermediates between stages.

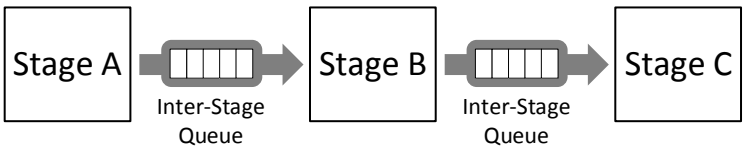
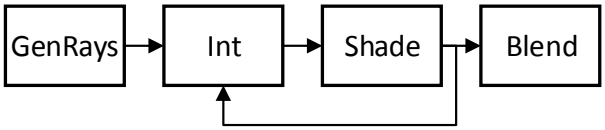
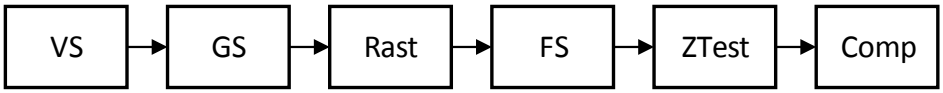


Figure 2.1: *A basic computational pipeline. Three stages A, B, and C are connected in a sequence, and inter-stage queues hold intermediate data between stages.*

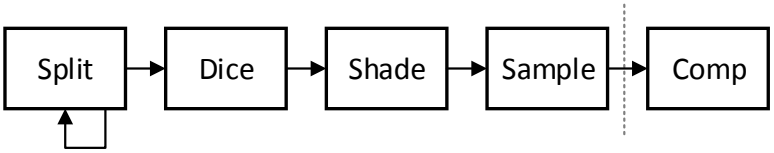


(a) Ray Tracing



(b) Forward Rasterization

(c) Deferred Rasterization



(d) Reyes

Figure 2.2: *Structures of some common graphics pipelines, showing the major stages and associated dataflow paths. Please see Table A.1 for expansions of names of stages.*

Pipelines serve as a common abstraction for both hardware (ASIC[1]) as well as software design. In addition to being easy to express, pipelines offer other benefits to help accelerate computation:

- **Concurrency** Different stages of a pipeline can often simultaneously work on independent units of work, since each works with its own set of inputs and outputs. Further, depending on the nature of computation, individual stages may themselves schedule computation to parallel processors.

- **Improved Throughput** In contrast to a monolithic computational unit, pipelines can hold many units of intermediate data within and between stages, improving the temporal

[1] Application-Specific Integrated Circuit

load-balance across these units. The result is a higher rate of input consumption as well as output production, despite having a similar (and often higher) overall latency.

- **Producer-Consumer Locality** Despite dividing computation into individual units, well-defined relationships between stages of a pipeline express the flow of data. This helps implementations exploit producer-consumer locality between stages, avoiding costly trips to external off-chip memory.

Given these advantages, it is not surprising that over the years graphics hardware has consistently embraced a pipeline-based design. Visual computations have always been a workload that demands high-performance and consequently aggressive implementation. Graphics applications also often involve processing large amounts of data. To a large extent, this workload is highly tolerant of latency but greatly benefits from high throughput [14].

2.2 Modern Graphics Pipelines

The goal of a graphics system is to map scene descriptions (geometry, materials, lights, camera, etc.) to 2D images. While ray-tracing-based techniques perform this mapping by iterating over image pixels at the highest level, rasterization-based techniques iterate over scene geometry to achieve a similar outcome. Both philosophies are well suited to a pipeline expression (Figure 2.2).

Ray tracing (Figure 2.2(a)) simulates the flow of one or more light rays for each image pixel. It models the propagation of these rays as they interact with the scene, and eventually computes the color for the originating pixel. Due to the high quality of images it is capable of generating, and the relatively low performance of implementations, ray tracing is primarily used in generating offline cinematic graphics.

Rasterization, on the other hand, computes the extent as well as colors for image regions covered by each scene primitive. This is analogous to computing interactions of light rays originating from objects and hitting the image plane. Rasterization is by far the most widely

used rendering algorithm for real-time graphics, as it forms the core technique inside modern GPUs. With interactive performance paramount in its design and evolution, rasterization is the prime example of a pipelined expression of graphics.

There are several flavors of a rasterization-based pipeline. The incarnation found within most modern GPUs, *forward polygon rasterization*, operates on triangles as primitives. Figure 2.2(b) shows how it transforms input triangles to image-space, tessellates them if needed, rasterizes them (maps them to screen-pixels), performs material and light calculations (shading) on covered pixels, and performs hidden-surface elimination on the shaded pixels. Figure 2.2(c) shows a common variant of the forward rasterization pipeline, a *deferred rasterization* pipeline, which shuffles the stages such that hidden-surface elimination happens before the more expensive pixel shading. Deferred rendering lends itself to hardware implementations [25] as well as software implementations built atop forward rasterization in modern GPUs [4].

Another flavor of rasterization-based pipelines is *Reyes* [10], a pipeline built to support high-quality graphics in a rasterization setup. Figure 2.2(d) shows the pipeline structure: input primitives (smooth surfaces) undergo dynamic-adaptive tessellation to generate pixel-sized micropolygons, which get displaced and shaded before optimized rasterization and high-quality hidden-surface elimination. Modern cinematic renderers often use a Reyes-style pipeline and use ray tracing during the shading phase for realistic lighting [8].

Several other flavors of rendering pipelines, both ray-tracing-based and rasterization-based, are useful for specific rendering workloads and applications. Examples of such pipelines include voxel rendering [9, 11, 38] and photon mapping [34].

2.3 Implementation Challenges and Motivation

2.3.1 Efficiency

Computer graphics as a workload has constantly pushed the boundaries of our computing capabilities. Regardless of whether the application domain is real-time or offline graphics, we

are far enough from being able to replicate visual reality that there is continuing demand for faster and more efficient implementations. Our technical capabilities are limited along all axes:

Figure 2.3: *A complicated synthesized image showing some of the complexities mentioned in Section 2.3. Image courtesy of Blender Foundation [16].*

- **Geometry** Extremely fine geometry is commonplace in today's video games as well as computer-generated movies. Characters with smooth, finely-displaced surfaces can easily offer geometry workloads measuring in tens of millions of polygons for real-time graphics, and an extra order of magnitude for offline graphics.

- **Materials and Shading**: Complex physically-based materials and shading effects ranging from subsurface scattering to hair, fur and volume effects pose an increasingly complicated computing challenge for real-time and offline shaders. Increasing texture detail has reached a point where it is no longer practical to have texture images resident in main memory, which is a problem that offers additional data management challenges [20].

- **Lighting**: Indirect lighting is now a standard component of real-time graphics as it has been for offline graphics. However, demands for increasing quality in both applications continues to grow. Believable shadows and illumination effects from multiple point- and area-lights adds to the growing complexity of synthesizing realistic images.

- **Camera Models** The use of physical camera models has a strong effect on the perceived realism of synthesized images. This involves simulating effects like motion-blur and depth-of-field, both of which require expensive integrals over large numbers of rendering samples. Rendering stereo 3D images adds further computational overheads, and in the near future we can expect to render detailed light fields for extremely immersive imagery [27].

- **Display Resolution** Increasing display resolution inevitably increases the workload-complexity of most stages of a graphics pipeline. Today's handheld devices with low-power processors are already supporting resolutions higher than 1920×1080 pixels [18, 19], while desktop displays are starting to reach 3840×2160 pixels [46]. Generating realistic images at these resolutions is challenging for all classes of graphics processing. The need to perform high-quality anti-aliasing on these images exacerbates this problem.

2.3.2 Flexibility

On one hand, graphics workloads are increasingly more challenging and consequently demand carefully optimized implementations, but at the same time target platforms like modern CPUs and GPUs are getting increasingly diverse and individually heterogeneous. In this broad ecosystem, we want to avoid the need for programmers to re-design an entire implementation for each application-architecture pair. Instead, we want to provide abstractions that enable them to create high-performance customized pipelines, but at the same time we want them to be able to re-purpose pre-optimized pipelines in different application scenarios or on different target platforms. Going forward, the flexibility of graphics pipelines is another growing concern.

To summarize, generating realistic images continues to be a growing implementation challenge. Workloads are getting increasingly larger and more complicated, while computational budgets continue to get tighter. At the same time, computing platforms and application scenarios are getting diverse. Software and hardware rendering solutions have consequently themselves evolved to become extremely complicated implementations. They offer high performance and efficiency, but they take a lot of effort to build, and offer limited flexibility and portability. We

argue that pipelines as abstractions only offer limited relief in this pursuit, so we offer Piko as an alternative abstraction, which augments a pipeline with information that helps express parallelism and locality, and helps reconfigure and reuse existing implementations.

2.4 Related Work

Historically, graphics pipeline designers have attained flexibility through the use of programmable shading. Beginning with a fixed-function pipeline with configurable parameters, user programmability began in the form of register combiners, expanded to programmable vertex and fragment shaders (e.g. Cg [33]), and today encompasses tessellation, geometry, and even generalized compute shaders in Direct3D 11. Recent research has also proposed programmable hardware stages beyond shading, including a delay stream between the vertex and pixel processing units [2] and the programmable culling unit [23].

The rise in programmability has led to a number of innovations not just within the OpenGL/Direct3D pipeline, but in its extension and in some cases complete replacement to implement novel rendering techniques. In fact, many modern games already implement a deferred version of forward rendering to reduce the cost of shading [4].

Recent research uses the programmable aspects of modern GPUs to implement entire pipelines in software. These efforts include RenderAnts, which implements an interactive Reyes renderer [52]; cudaraster [26], which explores software rasterization on GPUs; VoxelPipe, which targets real-time GPU voxelization [38]; and OptiX, a high-peformance programmable ray tracer [39]. The popularity of such explorations demonstrates that entirely programmable pipelines are not only feasible but desirable as well. These projects, however, target a single specific pipeline for one specific architecture, and as a consequence their implementations offer limited opportunities for flexibility and reuse.

A third class of recent research seeks to rethink the historical approach to programmability, and is hence most closely related to our work. GRAMPS [49] introduces a programming model that provides a general set of abstractions for building parallel graphics (and other)

applications. By and large, GRAMPS addresses expression and scheduling at the level of pipeline organization, but does not focus on handling efficiency concerns within individual stages. Instead, GRAMPS successfully focuses on programmability, heterogeneity, and load balancing, and relies on the efficient design of inter-stage sequential queues to exploit producer-consumer locality. The latter is itself a challenging implementation task that is not addressed by the GRAMPS abstraction. The principal difference in our work is that instead of using queues, we use 2D tiling to group computation in a manner that helps balance parallelism with locality. While GRAMPS proposes queue sets to possibly expose parallelism within a stage (which may potentially support spatial bins), it does not allow any flexibility in the scheduling strategies for individual bins, which, as we will demonstrate, is important to ensure efficiency by tweaking the balance between spatial/temporal locality and load balance. GRAMPS also takes a dynamic approach to global scheduling of pipeline stages, while we use a largely static global schedule due to our use of a multi-kernel design. In contrast to GRAMPS, our static schedule prefers launching stages farthest from the drain first, but during any stripmining or depth-first tile traversal, we prefer stages closer to the drain in the same fashion as the dynamic scheduler in GRAMPS. We explain these choices in detail in Chapter 5.

FreePipe [31] implements an entire OpenGL/Direct3D pipeline in software on a GPU, and explores modifications to the pipeline to allow multi-fragment effects. FreePipe is an important design-point. To the authors' knowledge, it is the first published work that describes and analyzes an optimized GPU-based software implementation of an OpenGL/Direct3D pipeline, and is thus an important comparison point for our work. We demonstrate that our abstraction allows us to identify and exploit optimization opportunities beyond the FreePipe implementation.

Halide [43] is a domain-specific embedded language that permits succinct, high-performance implementations of state-of-the-art image-processing pipelines. In a manner similar to Halide, we hope to map a high-level pipeline description to a low-level efficient implementation. However, we employ this strategy in a different application domain, programmable graphics, where data granularity varies much more throughout the pipeline and dataflow is both more dynamically varying and irregular. Spark [15] extends the flexibility of shaders such that

instead of being restricted to a single pipeline stage, they can influence several stages across the pipeline. Spark allows such shaders without compromising modularity or having a significant impact on performance, and in fact Spark could be used as a shading language to layer over pipelines created by Piko. We share design goals that include both flexibility and competitive performance.

Targeting more general-purpose applications with heterogeneous hardware (CPU-GPU and CPU-MIC), systems like StarPU [7] and AHP [40] generate pipelines from high-level descriptions. They concentrate more on task decomposition between heterogeneous processors and do not focus on the locality optimizations that we pursue in this work. Luk et al. [32], with Qilin, and Lee et al. [29] both split single kernels across heterogeneous processors. Qilin uses a training run to determine the split and use their own API. Lee et al. input CUDA programs and automatically transform them.

Chapter 3

Efficiency of Graphics Pipelines

3.1 Introduction

In this chapter, we discuss some implementations of graphics pipelines in traditional as well as contemporary high-performance renderers. Individually, these implementations present case studies in the co-optimization of algorithms, software, and hardware toward application-specific goals (e.g. performance, flexibility, quality, and power), but when considered together, they provide a deep insight into certain fundamental principles of the efficiency of computer graphics. Piko is an attempt to distill the most prominent of these principles into a flexible, programmable abstraction for graphics pipelines.

For the purposes of this discussion, we primarily consider implementations of rasterization-based pipelines. This is because there is a large number of historical and contemporary realizations of rasterization pipelines, which spans the spectrum of application goals and resource constraints. Studying this diverse set of pipeline implementations helps us draw inferences about the core ideas behind efficiently computing graphics workloads. We can then encapsulate these ideas into an abstraction that also offers the ability to reuse stages across implementations, and implementations across architectures.

Table 3.1 shows a summary of characteristics of some implementations of graphics pipelines.

In the interest of space, the table only covers prominent features like the underlying pipeline, work granularity, and the use of spatial tiling. We will now discuss these in detail.

3.2 Common Traits of Graphics Pipelines

To understand the design choices of pipeline implementations, we start by looking at some application-driven traits of graphics pipelines.

Parallelism

Large-scale parallelism is a fundamental characteristic of graphics applications. Inputs and intermediates routinely consist of millions of vertices, triangles, fragments, and pixels per frame. Individual computations on these primitives are also often independent.

Locality

Graphics applications also demonstrate significant computational and data locality. Due to the physical coherence of geometry, materials, and illumination in realistic scenes, computations that operate on these values also tend to be localized. Exploiting this locality is an important opportunity for improving implementation performance.

Changing Work Granularity

Another important aspect of how concurrency manifests in graphics pipelines is the heterogeneity of work granularity. Stages of a graphics pipeline usually work on different kinds of input. As data travels through, it changes form—vertices assemble into triangles, triangles rasterize into fragments, and so on. At each point, the presentation of parallelism as well as locality is different.

Implementation	Pipeline Flavor	Work Granularity	Use of Spatial Tiling
PixelFlow [13]	Deferred Raster	Per-primitive, per-pixel	Bucketing
Pixel Planes 5 [17]	Deferred Raster	Per-primitive, per-Pixel	Load-balanced parallel-tiling
AT&T Pixel Machine [41]	Forward Raster	Per-primitive, per-pixel	Static parallel-tiling
SGI InfiniteReality [35]	Forward Raster	Per-primitive, per-pixel	Tiled-Interleaved fragment generation
Pomegranate [12]	Forward Raster	Per-primitive, per-pixel	Static tiled fragment processing
Freepipe [31]	Forward Raster	Per-primitive	None
Modern GPUs [42]	Forward Raster	Multiple	Multiple Stages
Cudaraster [26]	Forward Raster	Per-primitive, per-pixel	
VoxelPipe [38]	Forward Voxelized Raster	Per-primitive, per-pixel	Sort-middle Tiling
PowerVR [25]	Deferred Raster	Per-vertex, per-primitive, per-pixel	Sort-middle Tiling
ARM Mali [48]	Forward Raster	Per-vertex, per-primitive, per-pixel	Sort-middle Tiling
Renderman [5]	Reyes	Per-patch, per-upoly, per-pixel	Bucketing at Split
RenderAnts [52]	Reyes	Per-patch, per-upoly, per-pixel	Bucketing at Split, Sample
GRAMPS [49]	Flexible	Flexible	Via Queue-Sets

Table 3.1: *A summary of characteristics of implementations of rasterization pipelines.*

Dependencies

Another common aspect of graphics pipelines is the need to manage dependencies between and within pipeline stages. Often, to ensure correct operation, we need to wait for a certain set of operations to finish before others can proceed. A common example of this dependence is during implementation of order-independent transparency [52]. All input fragments must be available before they can be blended in a back-to-front sorted order. Further, to ensure consistent rendering, graphics APIs sometimes require pipeline outputs to be committed in the same order in which inputs are presented.

3.3 Common Traits of Pipeline Implementations

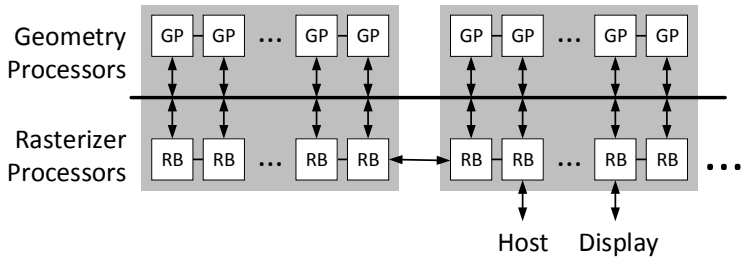
We now discuss how the above characteristics manifest in traditional as well as modern graphics architectures. These implementation trends help us gain insight into techniques that are both versatile and performance-critical. Consequently, we believe that an abstraction like Piko must also fundamentally embrace them.

Parallel Processing

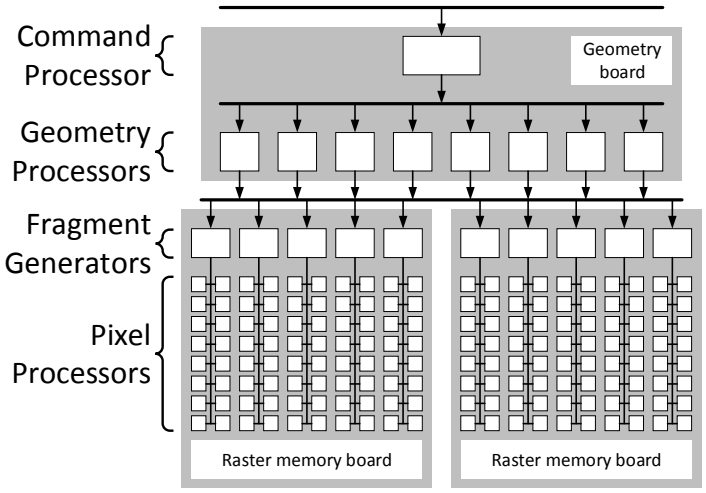
Since long before the mass-adoption of parallel processing in everyday computing, graphics hardware has consistently employed multiple units processing on similar units of data. Due to the demand for high performance, Single Instruction, Multiple Data (SIMD) as well as Multiple Instruction, Multiple Data (MIMD) styles of computing were popular even in the early days of graphics hardware [13, 17, 35]. They continue to be an integral part of today's visual computers [14].

The use of SIMD in graphics hardware already exploits the coherence in computations. To exploit the locality in data access, such as geometry, materials, textures, and framebuffer, implementations rely on carefully organized local stores and caches.

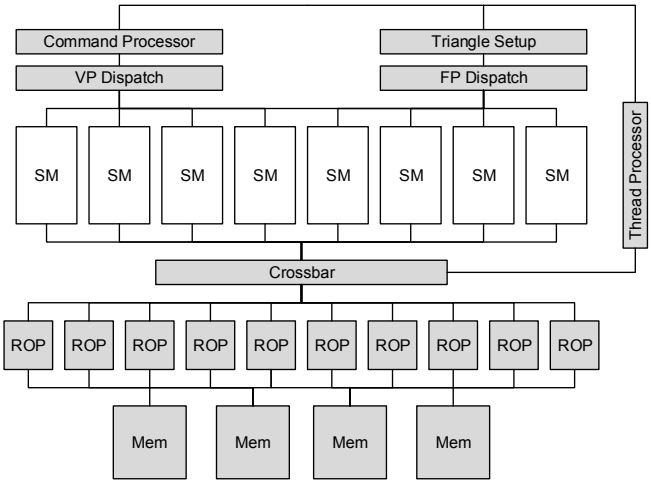
Implementations account for changing work granularity in pipelines. Early graphics hard-



(a) PixelFlow Architecture



(b) SGI RealityEngine



(c) NVIDIA G80 Architecture

Figure 3.1: *Work granularity across some graphics implementations. From the very beginning, implementations have had multiple domains of parallelism. (a) and (b) show the PixelFlow [13] and SGI RealityEngine [3] architectures, respectively, which have two primary parallelism domains: geometry and pixels. Modern GPUs [30] (c) unify these and other computations into a shared class of parallel processors.*

ware included sequential processing for geometry (transformation, clipping, and lighting), and switched to parallel processing for pixel operations (shading, depth-testing, blending) [13, 17, 41]. The hardware soon grew to perform parallel processing for geometry as well, although with fewer processors [35]. As workloads grew further, it became pragmatic to reuse the same set of cores for parallel processing across the pipeline [30]. However, granularities of operations still vary as different stages assign threads to different data entities (vertices, patches, triangles, fragments) and occupy varying number of processors. Figure 3.1 shows three different implementations of graphics pipelines with varying work granularity.

Fine-Grained Synchronization

Satisfying dependencies for a graphics pipeline is possible to achieve in a conservative manner. For example, for order-independent transparency, we may choose to only composite primitives when all fragments are available. In the same way, to respect pipeline order, we may only process primitives in-order for all stages. However, to meet performance goals, implementations tend to synchronize computation on a much finer scale, often at the level of individual or groups (tiles) of primitives or pixels. This helps reduce unnecessary pipeline stalls, resulting in higher efficiency. Our abstraction must allow opportunities to express such fine-grained synchronizations.

3.4 The Role of Spatial Tiling

A core contribution of this dissertation is the observation that spatial tiling, or grouping of pipeline data and computation into spatially organized bins, is an essential component of high-performance implementations of graphics pipelines. Most instances of graphics pipelines in Table 3.1 utilize spatial tiling for one or more purposes. In this section we will first provide a definition of spatial tiling and associated terms as used in this dissertation. Then we will discuss the benefits of spatial tiling in computer graphics, and how they map to high-performance implementations.

3.4.1 Terms Related to Spatial Tiling

Primitive We call any unit of data flowing through a graphics pipeline a primitive. Examples of primitives are triangles, surface patches, fragments, and pixels. We generally don't consider random-access state objects, like transformation matrices, materials, and lights, to be primitives, but some pipelines (e.g. deferred shading with light culling) might use them in the same way as primitives.

Tile or Spatial Bin A tile or spatial bin is any group or batch of primitives, such that the membership of the tile is defined using spatial properties (e.g. position, bounding box) of primitives. In many contexts a tile can be physically interpreted as an axis-aligned bounding box. In this dissertation we exclusively consider 2D tiles generated using a uniform tessellation of the screen.

Tiling or Spatial Binning We use the terms tiling and spatial binning interchangeably to refer to any technique that involves allocating input primitives into a set of tiles before processing them. In the literature the terms tiling and binning often also imply that the processing unit will process input tiles in sequence, one at a time. Our definition of these terms does not include any specification of processing order. We use the terms bucketing and chunking (below) for that purpose. Note that our abstraction currently only works for 2D uniform tiles.

Bucketing or Chunking Given a set of input tiles, a common way to process the data contained in them is for all processing cores to work exclusively on one tile at a time, sequentially traversing the set of tiles. We call this technique bucketing or chunking. Bucketing is often the technique of choice when peak memory usage or bandwidth consumption is a concern, because it tends to reduce the working set of computation, which consequently may fit entirely within fast and power-efficient local memory.

Parallel Tiling An alternative to bucketing is to process multiple tiles concurrently, a technique we call parallel tiling. In this setup, multiple tiles are in flight at the same time. Parallel tiling can be advantageous in many scenarios, including those where achieving high-

performance is more important than preserving bandwidth, and those where cores may not be able to share any significant amount of local memory.

3.4.2 Benefits of Spatial Tiling

Spatial tiling techniques are important in a variety of scenarios in graphics systems. They deliver multiple benefits depending on the application characteristics and usage context. We observe that spatial tiling is significantly helpful in matching the traits in Section 3.2 to those in Section 3.3:

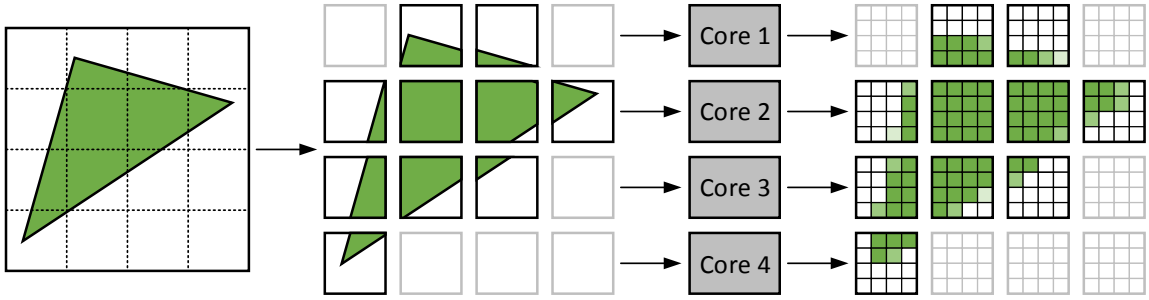
- It helps expose parallelism in operations that otherwise may not be amenable to parallel processing. For example, conservative rasterization helps split coverage determination for one triangle into multiple tiles, allowing parallel processing for the latter.

- It helps maintain spatial relationships between nearby graphics data, preserving the inherent coherence of computation and locality of data. Examples of computations that tend to be spatially-localized within tiles include texture mapping [22] and illumination.

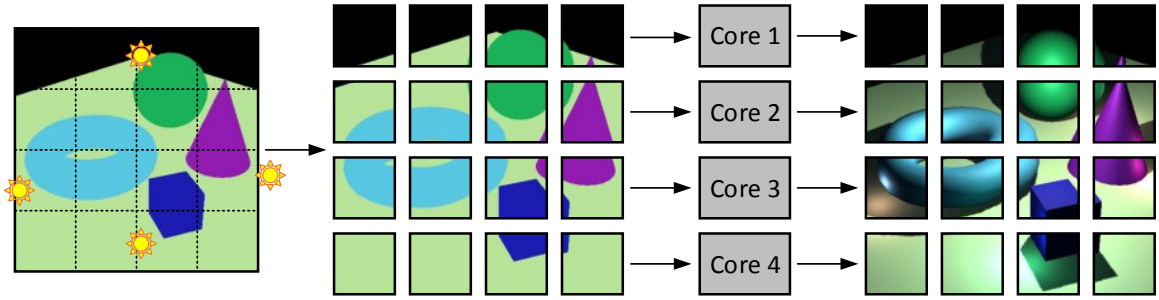
- Tiling patterns often belong within a pipeline stage, and evolve across stages as the work granularities change. This allows a graphics system to have varying degrees or domains independent of the use of spatial tiling.

- Spatial tiling helps isolate synchronization regions, as dependencies are often confined within tiles, and helps achieve fine-grained synchronization within a pipeline. For example, order-independent transparency techniques may require all fragments for a pixel to be ready before blending them into the framebuffer. Tiling this computation helps schedule some tiles before others are ready, allowing for better utilization of available resources.

Figure 3.2 shows examples of how tiling exposes concurrency and locality in real-life pipeline stages. Table 3.1 documents the prevalence of spatial tiling in existing graphics systems, and also shows the diversity of tiling patterns and associated computation for different implemen-



(a) Tiled rasterization of a large triangle helps parallelize the expensive task across multiple cores. In the presence of a large batch of primitives containing both large and small triangles, this results in a balanced workload independent of triangle size. Without such a technique, larger triangles would occupy their cores for an unevenly large amount of time.



(b) Tiled shading for a scene with multiple lights and materials. Within each tile, only a subset of lights need evaluation, and materials maintain coherence. This helps eliminate redundant work, and maximize execution locality (among SIMD lanes) and memory locality (texture cache efficiency). Images courtesy of Wikipedia user astrofra [6]

Figure 3.2: *These diagrams provide examples of how spatial tiling can be useful in exposing parallelism as well as preserving locality. In (a), tiling helps split the workload for rasterizing a large triangle across four execution cores, and in (b), tiling helps preserve spatial locality while shading a scene with multiple materials and lights.*

tations. While some pipelines use simple bucketing schemes to work on one screen tile at a time [5, 13, 52], others distribute work to tiles that operate in parallel [26, 35, 42, 48]. Similarly, some implementations use tiling just once [5, 13], but others tile at multiple points along the pipeline [42, 52]. Despite the frequent use of tiling in renderers and the variety of their use cases, APIs and programming models rarely expose the structure and computation for spatial tiling. With Piko we attempt to address this problem, by making tiling a first-class member of the abstraction.

For simplicity, we chose to restrict the abstraction to uniform tiling in 2D screen-space, even though tiling can be used in a wider range of scenarios. Extensions to Piko that address

generalized spatial tiling is interesting future work.

Chapter 4

Piko: An Abstraction for Programmable Graphics Pipelines

Based on the design priorities from Chapter 3, we now present the details of our abstraction. Recall that our goal is allow a programmer to define not just the function of a graphics pipeline, but also the structure of its computation as it relates to the underlying platform. The use of 2D spatial tiling is an important tool in this process, and hence it forms a central part of our abstraction. To allow exploring the gamut of tiling configurations in a flexible and portable manner, we have partitioned the abstraction such that tiling logic is largely decoupled from the stage algorithm.

In this chapter, we will cover three distinctive features of our abstraction:

- Separating pipeline stages into 3 phases, which both permits pipeline optimizations and factors pipeline descriptions into architecture-independent and -dependent parts;
- the development of a carefully chosen, small set of directives to capture scenarios for common pipeline optimizations; and
- most importantly, the deep integration of spatial binning into the abstraction to capture spatial locality.

We start our discussion by presenting a brief description of the target architectures as they pertain to our abstraction. We then provide an overview of the system from the programmer's as well as the implementer's perspectives. Finally, we walk through the various steps in defining a pipeline with Piko, and follow that with a discussion on how to express common scenarios for graphics pipelines using chosen directives.

4.1 Architecture Model

Piko aims to target a broad range of modern computing architectures, yet we wish to take advantage of important performance-focused hardware features, such as caching and SIMD parallelism. We thus restrict our implementation targets to a class of processors with multiple computational cores[1], each offering in-core parallelism via multiple threads or SIMD lanes, and possibly a cache hierarchy or other low-latency local memory to exploit spatial and temporal locality. Within this class of architectures, our most important targets are many-core GPUs and heterogenous CPU-GPU architectures. However, in the future we also would like to target clusters of CPUs and GPUs, and custom architectures like FPGAs.

4.2 System Overview

Figure 4.1 provides a programmer's perspective of Piko in building a forward rasterization graphics pipeline with shadow mapping. Recognizing the advantages of a traditional pipeline model, we also organize the graphics systems in a similar fashion. This is where a programmer should start expressing a Piko pipeline. The next step is to individually describe the functionality as well as the structure and scheduling of the tiles in each stage, which the programmer does by expressing the three phases for each stage. Next, this abstraction (a pipeline with three phases per stage) is input to the implementation framework, which is a compilation step that builds on the input to generate an optimized implementation for a given platform. In this dissertation, the output of this process is essentially a sequence of CUDA / OpenCL kernels that run on the

[1] In this paper we define a "core" as a hardware block with an independent program counter rather than a SIMD lane. For instance, an NVIDIA streaming multiprocessor (SM).

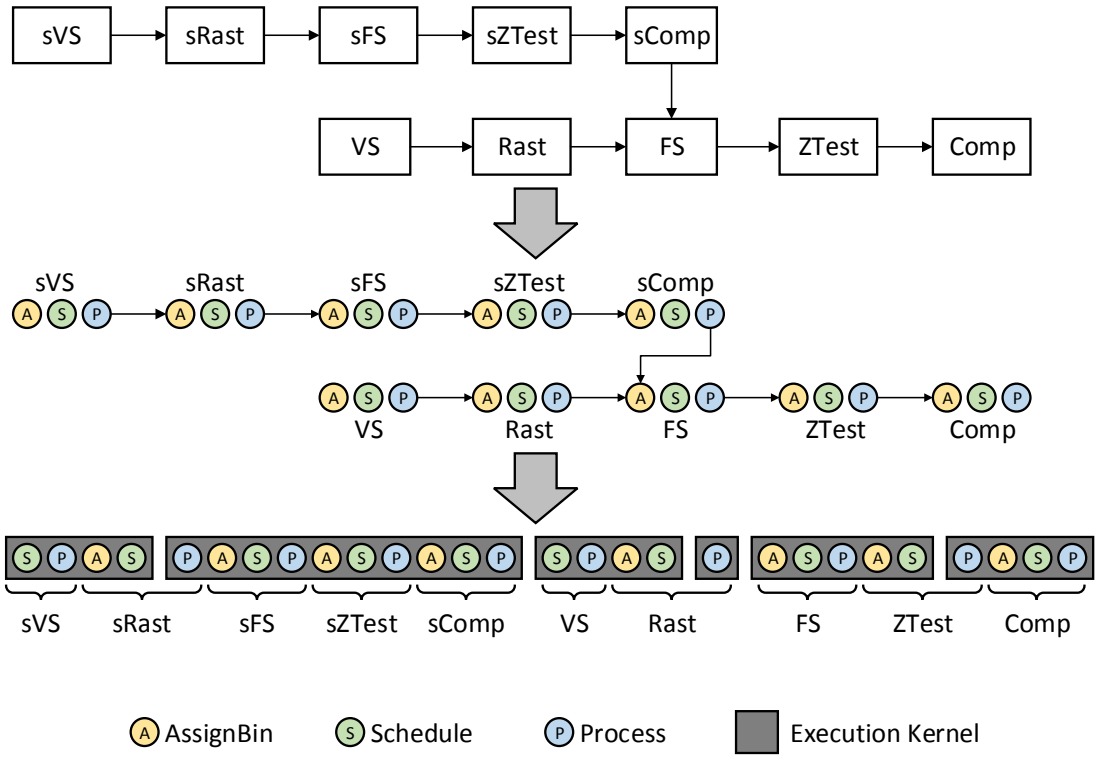


Figure 4.1: *Programmer's view of a Piko implementation, showing the three main steps to realizing an example graphics pipeline (forward rasterization with shadow mapping). Starting from a conventional pipeline (top), the programmer first expresses each stage using its AssignBin, Schedule, and Process phases. This (middle) forms the input to the implementation framework, whose output (bottom) is the final sequence of optimized kernels.*

target architecture.

From the implementer's perspective, Figure 4.2 provides an overview of this model. The user supplies a pipeline definition where each stage is separated into three phases: AssignBin, Schedule, and Process. Piko analyzes the code in this set of programs, and generates a pipeline skeleton that contains vital information about the flow of data in the pipeline. From this skeleton, Piko then performs synthesis, an operation that merges pipeline stages together into the kernel mapping; an efficient set of kernels to represent the original pipeline definition; and a schedule for running the pipeline (i.e., the order in which to launch kernels). A code generator then uses the kernels in conjunction with the schedule to generate the pipeline implementation. For our studies, implementations are in the form of software that runs on a high-performance

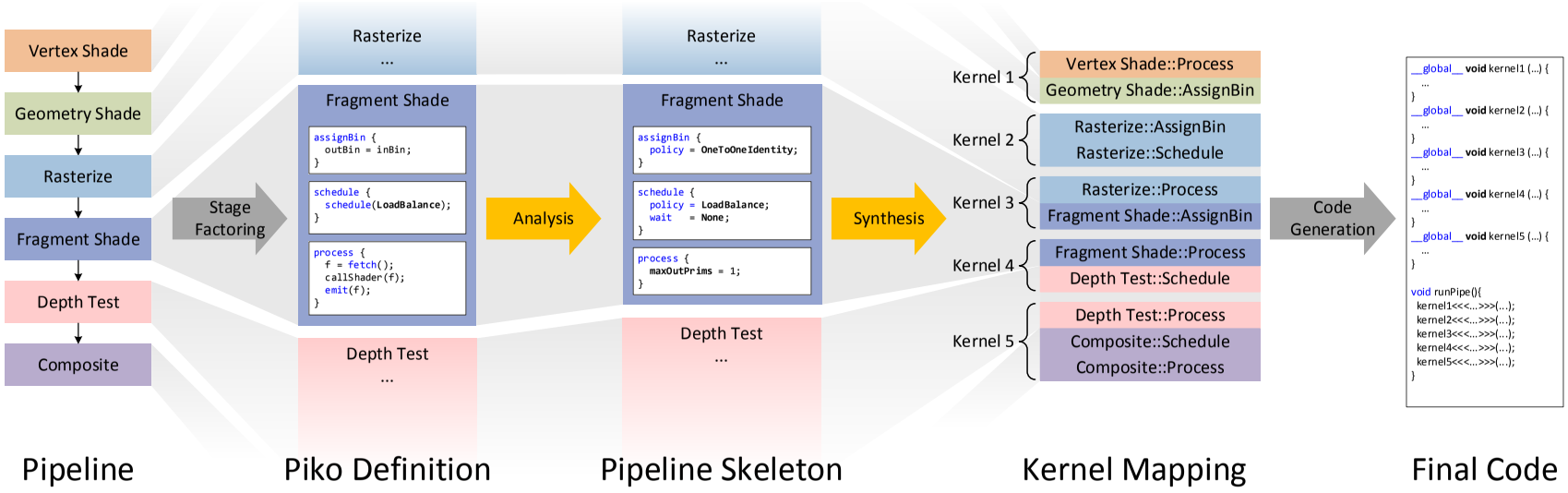


Figure 4.2: *Our Piko abstraction applied to a forward raster pipeline. The user writes a pipeline definition in terms of stages factored into AssignBin, Schedule, and Process stages. Piko runs an analysis step to generate a pipeline skeleton, then a synthesis stage. In the example above, Piko fuses the Depth Test and Composite stages into one kernel. The output of Piko is a set of efficient kernels optimized for multiple hardware targets and a schedule for running the pipeline.*

CPU, GPU, or a hybrid architecture. Thus, the final output is an optimized code for a target architecture. Chapter 5 provides a detailed discussion of synthesis optimizations and the various runtime implementations of Piko.

Our abstraction balances between enabling productivity and portability by using a high-level programming model, but specifying enough information to allow high-performance implementations. Using spatial tiling to achieve this goal is an important choice in this context—tiling is well-known and an intuitive construct in computer graphics. It just isn't an explicit part of conventional abstractions.

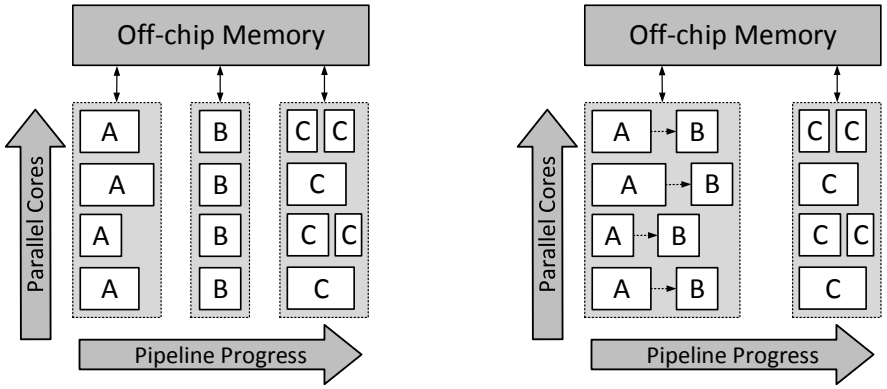
4.3 Pipeline Organization

At a high level, a Piko pipeline is identical to a traditional one. That is, it expresses computation within distinct stages, and dataflow as communication between stages (Chapter 2).

Consider a strawman system that would implement each stage as a separate kernel, distributing its work to all available parallel cores, and connect the output of one kernel to the input of the next through off-chip memory. The runtime of the strawman would then launch each kernel for the pipeline in a synchronous, sequential manner; each kernel would run over the entire scene's intermediate data, reading its input from off-chip memory and writing its output back to off-chip memory. The strawman would have ordered semantics and distribute work in each stage in FIFO order.

Such a system (Figure 4.3) would make poor use of both the producer-consumer locality between stages, and the spatial locality within and between stages. It would also require a rewrite of each stage to target a different hardware architecture. Piko, on the other hand, allows for explicit expression of spatial locality (through tiles), and exposes information about producer-consumer locality. It also factors each stage into architecture-dependent and -independent phases to enhance flexibility as well as portability.

Specifically, where the Piko abstraction differs from a conventional pipeline is inside a pipeline



(a) A Strawman Implementation (b) Optimized Strawman

Figure 4.3: *A strawman pipeline implementation (a), and potentially optimized version (b). The strawman loses all opportunities to exploit producer-consumer locality, since stages communicate through off-chip memory. Piko allows the designer to explore optimizations like the one in (b), where stage A sends intermediate data to stage B through local on-chip memory, reducing expensive off-chip transfers.*

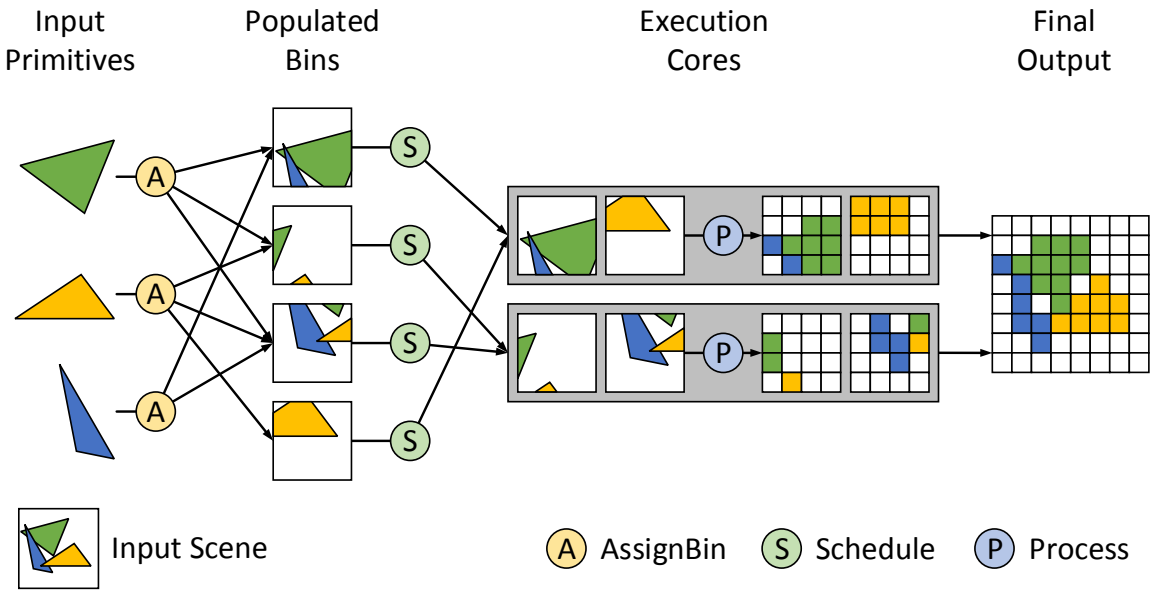


Figure 4.4: *The three phases of a Piko stage. This diagram shows the role of AssignBin, Schedule, and Process in the scan conversion of a list of triangles using two execution cores.*

stage. Programmers divide Piko pipeline stages into three *phases* (summarized in Figure 4.4). The input to a stage is a list of primitives (vertices, triangles, fragments, etc.), but phases, much like OpenGL/Direct3D shaders, are programs that apply to a single input primitive or a small

set thereof. More specifically, each phase works within a single screen-space tile, or bin. The first phase, AssignBin, specifies how a primitive is mapped to a user-defined bin or 2D tile. The second phase, Schedule, schedules an input bin to one or more computational cores. The third phase, Process, performs the actual computation on the primitives in a bin. Allowing the programmer to specify both how primitives are binned as well as how bins are scheduled onto cores allows a Piko implementation to take advantage of spatial locality in the stage. And because both AssignBin and Process are often architecture-independent, the task of porting a Piko pipeline to another architecture is most often just rewriting the Schedule phase only.

AssignBin The first step for any incoming primitive is to identify the tile(s) that it belongs to or may otherwise influence. Because this depends on both the tile structure as well as the nature of computation in the stage, the programmer is responsible for defining the mapping from primitives to bins. Algorithm 1 contains an example of a common AssignBin routine that simply assigns an incoming triangle to all bins overlapping it. More complicated examples may use an extended bounding box (for blurry primitives), or a tighter mapping using a conservative rasterization algorithm.

Algorithm 1 AssignBin phase for the stage in Figure 4.4

```
1: Input: Triangle T_i
2: for all bins B_j overlapping T_i do
3:     Add T_i to B_j's input list
4: end for
```

Schedule Once primitives are distributed into bins, the Schedule phase allows the programmer to specify how and when bins are scheduled onto cores. The input to Schedule is a reference to a spatial bin, and the routine can chose to dispatch computation for that bin if needed. Thus, Schedule allows the programmer to express scheduling priorities and dependencies in a programmable fashion. Two examples of scheduling constraints are very common in graphics pipelines: the case where all bins from one stage must complete processing before a subsequent stage can begin, and the case where all primitives from one bin must complete processing before any primitives in that bin can be processed by a subsequent stage. Either of these is easy to express during this phase. The other purpose of Schedule involves assigning scheduling pri-

orities, examples of which include creating batches of primitives to improve SIMD-efficiency, distributing bins to preferred cores to improve cache access. Algorithm 2 shows an example of a Schedule phase that schedules batches of N triangles to cores in a dynamic, load-balanced fashion.

Algorithm 2 Schedule phase for the stage in Figure 4.4

```
1: Input: Bin B_j
2: if B_j contains at least N primitives then
3:     Find least busy execution core C_k
4:     Schedule B_j to execution core C_k
5: end if
```

Because of the variety of scheduling mechanisms and strategies on different architectures, we expect Schedule phases to be architecture-dependent. For instance, a many-core GPU implementation may wish to take advantage of hardware support for load balancing when assigning tiles to cores, whereas a hybrid CPU-GPU may wish to preferentially assign tasks to either the CPU or the GPU, or split between the two.

Schedule phases specify *where* and *when* to launch computation for a bin. For instance, a programmer may know that a particular hardware target will only be efficient if it launches computation in batches of a minimum size. Further, the programmer may specify dependencies that must be satisfied before launching computation for a bin. For instance, an order-independent compositor may only launch on a bin once all its fragments are available. It is important to note that realization of such scheduling constraints depends on the implementation strategy: a persistent-threads implementation [21] might use a scoreboard to resolve dependencies, while a multi-kernel implementation might schedule kernels in an order that conservatively resolves dependencies. To summarize, the purpose of Schedule is to link an algorithm with an underlying architecture, without explicitly providing the implementation.

Process While AssignBin defines how primitives are grouped for computation into bins, and Schedule defines where and when that computation takes place, the Process phase is the one that defines the actual computation. The most natural example for a Process phase is a vertex or fragment shader, but it could also be an intersection test, a depth resolver, a subdivision

task, or any other piece of logic that often forms a standalone stage in a conventional graphics pipeline description. The input to Process is a bin and a list of primitives contained in the bin, and the output is a set of zero or more primitives for the next stage. Process phases use *emit* to identify output primitives.

Algorithm 3 Process phase for the stage in Figure 4.4

```
1: Input: List of triangles, T, belonging to bin B_j
2: for all triangles T_i ∈ T do      {in parallel}
3:     for all pixels P_m ∈ B_j do
4:         if T_i covers P_m then
5:             Create fragment F_{i,m} for T_i at P_m
6:             Emit F_{i,m}
7:         end if
8:     end for
9: end for
```

We expect that one instance of a Process routine (attached to a bin) will occupy an entire computational core, utilizing its available parallelism and local memory for efficiency. To this end, the programmer can usually exploit the independence and locality (respectively) of primitives in bin. But this is not a rule: if a single primitive offers sufficient parallelism or cache efficiency (e.g. while performing surface tessellation), the programmer is free to use single-element lists as input. For instance, in Algorithm 3, the Process routine uses thread identifier i to distinguish between threads/lanes within a single core.

4.4 Defining a Pipeline with Piko

While language design is not the goal of this work and will be an interesting area for future work, for purposes of evaluation we express pipelines using a simple C-like language. There are two parts to defining a pipeline: a structural definition, and a functional definition. We first discuss the latter.

Functional Definition

For each stage that is a part of a graphics pipeline, a programmer must provide a description of its behavior. First, he/she must either define all three phases which form the stage, or pick a stage from a library of commonly-used stages (e.g. a compositor). Algorithms 1, 2, and 3 show an example expression.

Structural Definition

After specifying the behavior of all stages used in the pipeline, the programmer needs to define their connectivity. While most pipelines organize stages as linear sequences with minor variations, we allow the programmer to express arbitrary directed graphs. But this freedom comes with a caveat—we do not guarantee that every pipeline graph will run to completion. Section 5.4.1 provides examples of scenarios where this might be a problem, and how a programmer might be able to avoid it. As with previous work [49], it is the programmer's responsibility to manage resources as well as dependencies. However, through Schedule we provide more control over exactly how and when to schedule the work for a given stage, which can help avoid deadlocks at run time.

Directed graphs allow each node to be connected to zero or more upstream as well as downstream nodes. Within a stage, this may create the need to distinguish between different upstream/downstream stages. Based on our observation of a several pipelines, we don't find it necessary to have separate upstream stages. However, we need the programmer to tell us to which downstream stage we should send an outgoing primitive. As an example, the split stage in Reyes can send its output (a subdivided patch) to either dice stage or back to the split stage. How can one express such behavior?

Taking inspiration from system-design principles, we allow each stage to have zero or one input port(s) to receive input primitives, and zero or more output port(s) to direct output primitives (Figure 4.5). Process can now simply emit output primitives to one of its output ports. Moreover, specifying stage connectivity simply becomes a matter of hooking together

input and output ports from various stages.

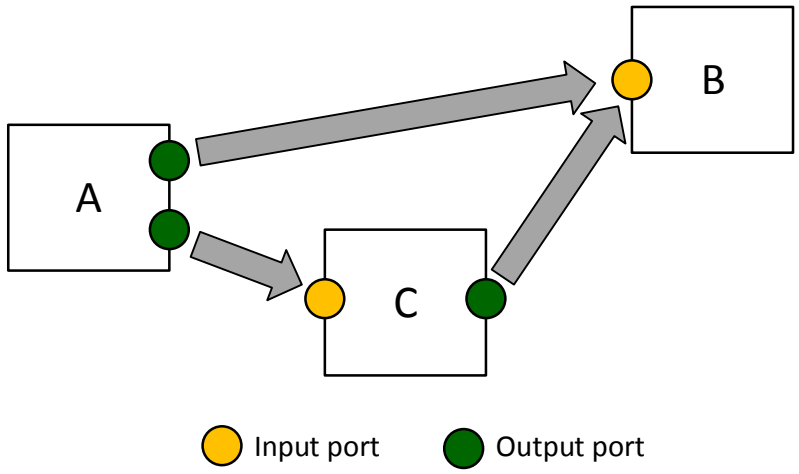


Figure 4.5: *Expressing stage connections using ports allows directing different stage outputs to different stages. In the above diagram, output ports are shown in green, and input ports are shown in orange.*

4.5 Using Directives When Specifying Stages

To target Piko pipelines to a given implementation strategy, we need an understanding of the behavior of both the pipeline structure, and that of individual stages. We can use this understanding to formulate several implementation-specific optimizations, which we describe in detail in Chapter 5. Using a simple clang-based[2] analyzer, we could derive some information through simple static analysis. However, we found many behavioral characteristics that were either hard or impossible to infer at compile-time.

For these difficult characteristics as well as some commonly used easier characteristics, we added several simple *directives* for programmers to provide hints about pipeline behavior. We summarize these directives in Table 4.1 and discuss their use in Chapter 5.

We divide directives into two categories. The first is *policies*, which are used to succinctly express common patterns while also passing additional information to the optimizer. For instance, we might direct the scheduler to use either static (scheduleDirectMap) or dynamic

[2]Clang is a front-end for C-based languages based on the LLVM compiler framework [28].

Phase	Name	Category	Purpose
AssignBin	assignOneToOneIdentity	Policy	Assign incoming primitive to same bin as previous stage
	assignOneToManyRange	Policy	Assign incoming primitive to a contiguous range of bins, e.g. a bounding-box
	assignOneToAll	Policy	Assign incoming primitive to all bins
	maxOutBins	Hint	The maximum number of bins to which each primitive can be assigned
Schedule	scheduleDirectMap	Policy	Statically schedule each bin to available cores in round-robin fashion
	scheduleLoadBalance	Policy	Dynamically schedule bins to available cores in a load-balanced fashion
	scheduleSerialize	Policy	Schedule each bin to a single core
	scheduleAll	Policy	Schedule a bin to all cores (used with tileSplitSize)
	tileSplitSize	Policy	Size of chunks to split a bin across multiple cores
	scheduleBatch(k)	Policy	Prefer to launch stages when at least k primitives are ready
	scheduleEndStage(X)	Policy	Wait until stage X is finished
	scheduleEndBin	Policy	Wait until previous stage for the current bin is finished
Process	processOneToOne	Hint	No data expansion during Process
	processOneToMany	Hint	Possible data expansion during Process
	maxOutPrims	Hint	The maximum number of output primitives that can result from each input primitive

Table 4.1: *The list of directives the programmer can specify to Piko during each phase. The directives provide basic structural information about the workflow and facilitate optimizations.*

(scheduleLoadBalance) load balancing. The second category of directives is *hints*, which provide information to the optimizer that could theoretically be derived from the pipeline description, but where such a derivation would be difficult for complex cases. For example, we may wish to statically determine whether a Process stage emits exactly one output per input, which is straightforward in some cases but may be quite difficult in the presence of loops and conditionals. Hints allow the programmer to express those conditions easily. If the programmer specifies no hints, the optimizer will generate correct, conservative code, but may miss opportunities for optimization. To summarize, policies help provide a concise description of common behavior that a programmer might wish to realize, and hints provide extra information that might aid optimization.

While synthesizing and implementing a pipeline, we combine directives with information that Piko derives in its "analysis" step to create what we call a "pipeline skeleton". The skeleton is the input to Piko's "synthesis" step, which we describe in Chapter 5.

4.6 Expressing Common Pipeline Preferences

We now present a few commonly encountered design scenarios and strategies for graphics pipelines, and how one can express and interpret them in our abstraction. Please refer to Section 3.4.1 for a list of how this dissertation interprets terms related to spatial tiling.

No Tiling

In cases where tiling is not the pragmatic choice, the simplest way to indicate it in Piko is to set bin sizes of some or all stages to a large value (e.g. screen width and height). Alternatively, a bin size of 0×0 indicates the same to the runtime. This is often useful for pipelines (or stages) that exhibit parallelism at the primitive level. To specify the size of primitive groups to be sent to each core, one can use tileSplitSize in Schedule to specify the size of individual primitive-parallel chunks.

Bucketing Renderer

Due to resource constraints, often the best way to run a pipeline to completion is through a depth-first processing of bins, that is, running the entire pipeline (or a subset of pipeline stages) over individual bins in a sequential order. In Piko, it is easy to express this preference through the use of a scheduleAll policy, which indicates that any bin of the stage should map to all available cores at any given time. Our synthesis scheme prioritizes depth-first processing in such scenarios, preferring to complete as many stages as possible for one bin before processing the next bin.

Sort-Middle Tiled Renderer

A very common design methodology for forward renderers (in both hardware and software) divides the pipeline into two phases: object-space geometry processing and screen-space fragment processing. Since Piko allows a different bin size for each stage, we can simply use screen-sized bins with primitive-level parallelism in the geometry phase, and smaller bins for the screen-space processing in the fragment phase.

Use of Fixed-Function Hardware Blocks

Renderers written for modern GPUs try their best to use available fixed-function hardware units for performance- as well as power-efficiency. These renderers include both those written using a graphics API (e.g. DirectX and OpenGL) and to some extent those written using general-purpose GPU languages (e.g. CUDA, OpenCL). Hence, it is important for pipelines written using Piko to be able to do the same. Fixed-function hardware accessible through CUDA or OpenCL (like texture fetch units) is easily integrated into Piko using the mechanisms in those APIs. However, to use standalone units like a hardware rasterizer or tessellation unit that cannot be directly addressed, the best way to abstract them in Piko is through a stage that implements a single pass of an OpenGL/Direct3D pipeline. For example, a deferred rasterizer could use OpenGL/Direct3D for the first stage, followed by a Piko stage for the shading pass.

4.7 A Strawman Pipeline in Piko

To illustrate designing pipelines using Piko, we describe how to express our strawman from Section 4.3. First, the strawman uses no tiling, so we set all bin sizes so 0×0. As a direct consequence, AssignBin phases across the pipeline simply reduce to identity. Finally, Schedule for each stage simply partitions the input into equal-size chunks to distribute across the architecture. During implementation, this would get reduced to a two-level thread hierarchy between and within cores, and be replaced by the built-in hardware or software scheduler. In other words, an optimized implementation of our strawman would simply be a sequence of synchronous kernel launches, one for each stage.

Chapter 5

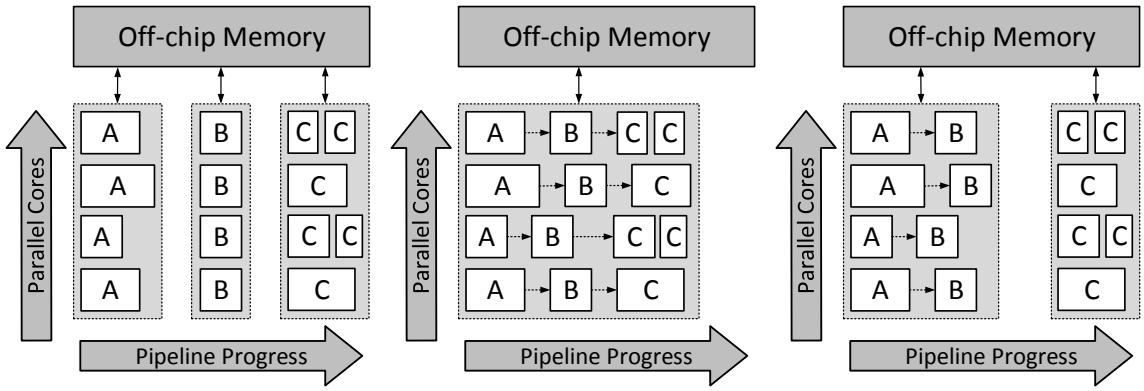
Implementing Flexible Pipelines using Piko

So far, we have presented the design of Piko as an abstraction that not only helps express the functionality of a graphics pipeline, but also provides an interface to express efficiency concerns like batching computation, resolving dependencies, and scheduling computation. We have also discussed how one can express a few of the most common pipeline behaviors within our framework. In this chapter, we present a strategy for implementing pipelines expressed in Piko.

We start by describing our choice of implementing parallel applications using a multi-kernel model. We then describe the various steps that take a pipeline described in Piko as input, and output an optimized software implementation to run on the target architecture (Figure 4.2). This includes pipeline analysis, where we discover structural information about the pipelines, and synthesis, during which we map a pipeline skeleton to a list of kernels. We then discuss our runtime implementation where we convert the kernel mapping into a final implementation for two specific architectures. Finally, we present an evaluation of the abstraction as well as our implementation along four axes: programmability, efficiency, flexibility, and portability.

5.1 Multi-kernel Design Model

Targeting architectures described in Section 4.1, we follow the typical convention for building complex parallel applications using APIs like OpenCL and CUDA. We instantiate a pipeline as a series of kernels, each of which represents machine-wide computation for one or more pipeline stages. Rendering each frame consists of a sequence of kernel launches scheduled by a host CPU thread. In this model, individual kernel instances run synchronously and may not share data with other kernels through local caches or local shared memory. However, we may choose to package multiple stages to the same kernel to enable data sharing between them.



(a) Multi-kernel (b) Persistent Threads (c) Optimized Multi-kernel (Piko)

Figure 5.1: *Alternatives for Piko implementation strategies: (a) represents our strawman's multi-kernel model, which suffers from load imbalance and does not exploit producer-consumer locality. (b) represents a persistent-thread model, which achieves both load-balance and helps exploit locality, but requires a more complicated implementation and suffers from overheads. Our choice, (c), represents a variant of (a), where kernels A and B are opportunistically fused to preserve locality and load balance, without excessive overheads or complicated implementation.*

One alternative to such a multi-kernel design is a persistent-kernel design [1, 21], such as that used in OptiX [39]. In a persistent-kernel design, the host launches a single kernel containing the entire pipeline. This single kernel is responsible for managing work scheduling, data communication, and dependencies for every stage of a pipeline, all within its execution. Such an implementation bypasses built-in hardware scheduling mechanisms, and hence the design is potentially more flexible than a multi-kernel one. A persistent-kernel also offers opportunities

in maintaining intermediate data on-chip between pipeline stages.

Either strategy is valid for Piko-based pipeline implementations, but we prefer the multi-kernel strategy for two reasons:

- By requiring the programmer to write their own parallel scheduler and dependency resolution, persistent thread formulations are more complex to write and tend to have higher overheads. They also often suffer from inefficiencies like high register pressure and a low hit rate in the instruction cache.
- One of the major advantages of persistent threads is the ability to capture producer-consumer locality. But the optimizations we present in this chapter are designed precisely to capture that locality in an opportunistic fashion, without the need for a complex persistent-kernel implementation.

5.2 Compiling Piko Pipelines

Taking as input a pipeline designed using the Piko abstraction, i.e. a structural definition expressing the pipeline graph and a functional definition characterizing the three phases for each stage, the following sections describe how a compiler framework would arrive at the final program that runs on the target architecture. The first step in this process is Pipeline Analysis (Section 5.3), which extracts information from the pipeline description to help fuel the optimizations we perform during Pipeline Synthesis (Section 5.4). Finally, we describe architecture-specific aspects of the implementation in Section 5.5.

5.3 Pipeline Analysis

The functional and structural definitions of a pipeline form the input to our implementation framework (Section 4.4). As the first step in this process, we use a simple clang-based tool to extract a summarized representation of the pipeline's structure and function, or the pipeline skeleton. The goal of this tool is to supplement the information already available in the form

of directives (Table 4.1). At a high-level, the pipeline skeleton contains (a) organization of the pipeline graph, (b) tile sizes for all stages, and (c) a list of user-supplied and automatically determined directives for each phase, summarizing its operation. Static analysis can only extract such information in a conservative fashion, hence user-supplied directives are still extremely important for an optimized implementation.

Our Clang analyzer works by walking the abstract syntax tree (AST) of the Piko pipeline code. It starts by parsing the structure of the pipeline graph and extracting AssignBin, Schedule, and Process for each of its stages. For each AssignBin, the analyzer looks for existing directives (e.g. assignOneToOneIdentity) first, and for the remaining code it tries to detect the policy statically. It also tries to determine maxOutBins if statically possible by counting the bin assignments in the phase. It performs a similar set of operations for each Schedule, detecting whether the tile scheduling maps to one of the simple cases (e.g. scheduleLoadBalance) or a custom scheduler (e.g. a work stealing scheduler). Finally, the analyzer looks at each Process program to attempt to count the number of output primitives for each input, and setting either processOneToOne or processOneToMany, and in some cases estimating maxOutPrims.

5.4 Pipeline Synthesis

The most important part of our implementation strategy is pipeline synthesis. This step inputs the pipeline skeleton, applies several key optimizations to the pipeline, and constructs a kernel plan that describes the contents of each kernel together with a schedule for running them.

During synthesis, our goal is to primarily perform architecture-independent optimizations. We do this in two steps. First, we identify the order in which we want to launch the execution of individual stages. With this high-level stage ordering, we then optimize the organization of kernels to both maximize producer-consumer locality, and to eliminate any redundant/unnecessary computation. The result of this process is a *kernel mapping*: a scheduled sequence of kernels and the phases that make up the computation inside each. The following sections discuss our approach in detail.

5.4.1 Scheduling Pipeline Execution

Given a set of stages arranged in a directed graph, what should be the order of their execution? The Piko philosophy is to use the pipeline skeleton with the programmer-specified directives to build a static schedule[1] for these stages. A static ordering of stages is appropriate for our implementation for two reasons: first, it simplifies compilation by separating the task of optimizing stage order from optimizing stage computation, and second, we expect Piko inputs to be more regular in their order of stage execution as compared to non-graphics applications. Since this may not be true for all potential graphics as well as non-graphics use cases for Piko, we plan to revisit this decision in the future.

The most straightforward schedule is for a linear, feed-forward pipeline, such as a simple OpenGL/Direct3D rasterization pipeline. In this case, we schedule stages in descending order of their distance from the last (drain) stage.

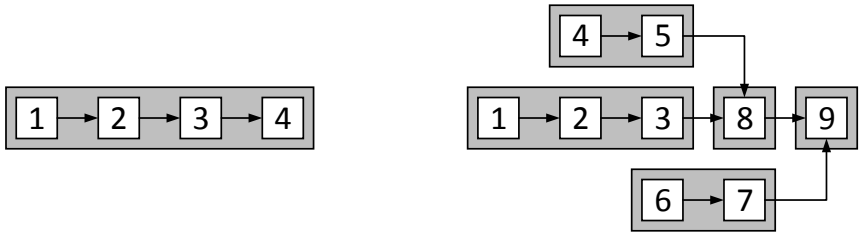


Figure 5.2: *These examples show how we decompose a pipeline graph into linear branches (denoted by gray regions). The simple example on the left has a single branch, while the more complicated one on the right has five branches. We determine schedule order by prioritizing branches that start farther from the pipeline's drain stage.*

Given a more complex pipeline graph, we partition it into linear *branches* at compile-time. Analogous to a basic-block from compiler terminology, a branch is simply a linear subsequence of successive stages such that except for the first stage of the branch, each stage has exactly one preceding stage, and except for the last stage of the branch, each stage has exactly one succeeding stage. Figure 5.2 shows the subdivision of two pipeline graphs into individual branches. In order to partition the graph into disjoint branches, we traverse the graph and mark

[1] Please note that the scheduling described in this section is distinct from the Schedule phase in the Piko abstraction. Scheduling here refers to the order we run kernels in a generated Piko pipeline.

all stages that must start a new branch. Then, for each stage that starts a branch, we assign all successive unmarked stages to that branch. A stage begins a new branch when it either a) has no previous stage, b) has multiple previous stages, c) is the child of a stage with multiple next stages or d) depends on the completion of another stage (e.g. using scheduleEndStage).

This method results in a set of linear, distinct branches with no stage overlap. For scheduling within each branch, we use the simple technique described previously. For inter-branch scheduling, we use Algorithm 4. We attempt to schedule branches in decreasing order of the distance between the branch's starting stage and pipeline's drain stage, but we constrain the schedule in two ways: first, at least one predecessor for each branch must be scheduled before it is scheduled, and second, if one stage depends on the completion of another stage, the branch containing the former may not be scheduled before the latter. Thus, a branch may only be scheduled if both these constraints (predecessors and dependencies) have been satisfied. If we encounter a branch where this is not true, we skip it until constraints are satisfied. As an example, Rasterization with shadow mapping (Figure 4.1) requires this more complex branch ordering method; the branch of the pipeline that generates the shadow map must execute before the main rasterization branch, so that the latter's pixel shader can use the shadow map to compute illumination [2].

While we can statically schedule non-cyclic pipelines, cycles create a dynamic aspect because we often do not know at compile time how many times the cycle will execute. Fortunately, most graphics pipelines have relatively small cycles—often within the same stage, and rarely including more than a handful of stages—and hence we resort to a simple heuristic. For cycles that occur within a single stage (e.g. Reyes's Split), we repeatedly launch the same stage until the cycle completes. Multi-stage cycles (e.g. the trace loop in a ray tracer) pose a bigger stage ordering challenge. For these cases, we first traverse the pipeline graph and identify all cycles, and the stages that they contain. Then, while scheduling any branch with multiple predecessors, we allow the branch to be scheduled if at least one of the predecessors is not part

[2] Strictly speaking, we only want shadow-map-generation to finish before fragment shading in the main rendering branch. However, it is our branch scheduling strategy that conservatively restricts the schedule, not the choices made in the abstraction.

Algorithm 4 Scheduling pipeline branches in Piko. In this algorithm, predecessors of a stage are all stages that provide input to it, and dependencies of a stage are all stages who must finish before the former can execute. For a branch to be ready to schedule, all dependencies and at least one predecessor must have been scheduled.

Input: List of branches B
$D \leftarrow$ Distances from drain for starting stages in B
$B' \leftarrow$ Sort B in descending order of D
$L \leftarrow \phi$ {final branch schedule}
$i \leftarrow 0$
while not $B' = \phi$ **do**
  $H'_i \leftarrow B'[i]$'s starting stage
  $P \leftarrow$ Branches containing predecessors of H'_i
  $C \leftarrow$ Branches comprising cycles containing H'_i
  $W \leftarrow P \setminus C$ {set difference}
  $S \leftarrow$ Branches containing dependencies to H'_i
  if $(W \subseteq L)$ **and** $(S \subseteq L)$ **then**
    Append $B'[i]$ to L
    Remove $B'[i]$ from B'
    $i \leftarrow 0$ {restart search}
  else
    $i \leftarrow (i+1)$ {try next branch}
  end if
end while
return L

of a cycle containing this branch, and has already been scheduled. Essentially we prioritize launching stages that provide entry into a cycle. Algorithm 4 explains our heuristic in detail, and Figure 5.3 shows an example. Appendix A shows some real schedules built using this method.

By default, a stage will run to completion before the next stage begins. That is, all available bins for the stage will launch within in the same kernel call. However, we deviate from this rule in two cases: when we fuse kernels such that multiple stages are part of the same kernel (discussed in Section 5.4.2.1), and when we must launch stages for bins in a depth-first fashion (e.g. for a bucketing renderer). The latter is essentially a form of stripmining, where we let multiple stages work on a part of the input before processing the next part. We encounter this scenario when a stage specification directs the entire machine to operate on a stage's bins in sequential fashion (e.g. the scheduleAll directive). We continue to launch successive stages for

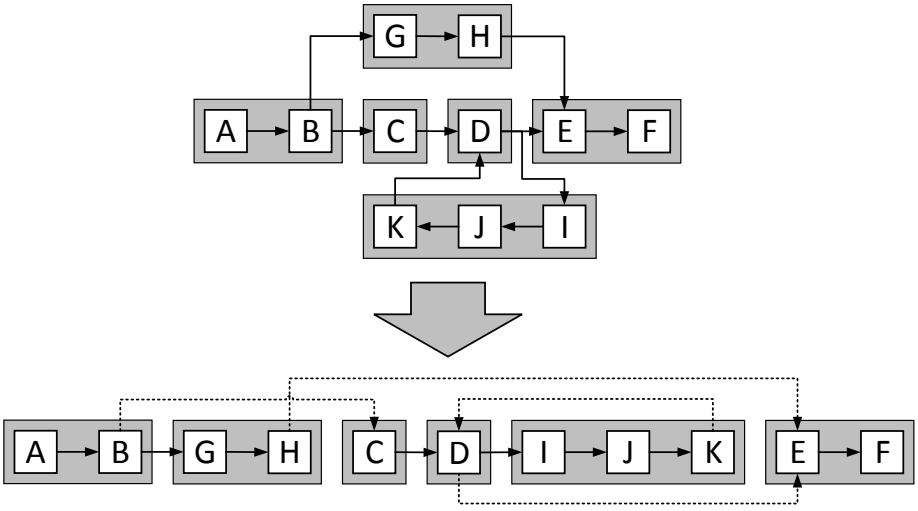


Figure 5.3: *Example showing how Algorithm 4 would organize the order of execution for a complex pipeline graph. The kernels G–H and C are equidistant from the drain stage F, and may be launched in any order after A–B. The backward-edge in the final schedule represents the loop that the host would run, repeatedly calling kernel D and kernel I–J–K in succession, until the cycle is satisfied.*

each bin as long as it is possible; we stop either when we reach a stage that has larger tile sizes than the current stage, or has an dependency that prohibits execution. In other words, when given the choice between launching the same stage on another bin and launching the next stage on the current bin, we choose the latter. As we discussed in Chapter 2, this decision prioritizes draining the pipeline over generating new intermediate data, and aligns with the choices of Sugerman et al. [49].

The reader should note that our heuristics for scheduling pipeline stages are largely opportunistic, and can sometimes result in suboptimal or even impractical schedules. Examples of these scenarios include:

- We may miss inter-stage optimization opportunities across branch boundaries. For example, in the shadow mapping pipeline from Figure 4.1, the Rast stage is prohibited from shading fragments as it generates them despite the fact that in the final schedule, all inputs to FS are available by the time we launch Rast.

- By externalizing multi-stage cycles to host-driven loops, we cannot schedule them to

run within a single kernel, which will sometimes be the most optimal implementation choice. The prime example of this situation is ray tracing, where the pragmatic choice is to perform the intersect-shade cycle within the same kernel, but our branch scheduling logic does not allow that scenario. Dynamic parallelism on recent GPUs [37] might address this problem by allowing dynamically scheduled cycles containing multiple stages. As we note in Chapter 6, this is an interesting area for extending Piko in the future.

- We may generate too much dynamic intermediate state, which for storage reasons may prohibit subsequent stages from executing, causing a runtime failure. In such cases it is usually appropriate for the programmer to resort to a bucketing scheme (which helps limit peak memory usage), but our heuristics do not recognize that situation automatically.

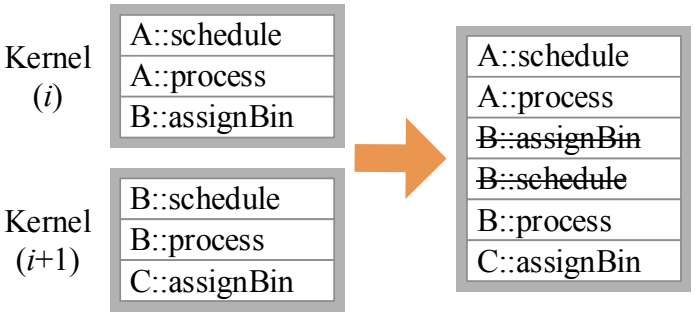
Fortunately, graphics applications rarely present complicated cycles and dependencies, and as a result our method works on most inputs without problems. In the worst cases, the programmer can use Schedule directives to force specific scheduling choices, for example, a bucketing renderer. Nevertheless, complicated pipelines may be input to Piko, so we anticipate that optimal stage ordering will be an important and interesting target for future work.

5.4.2 Optimizing Pipeline Execution

The next step in generating the kernel mapping for a pipeline is determining the contents of each kernel. We begin with a basic, conservative division of stage phases into kernels such that each kernel contains three phases: the current stage's Schedule and Process phases, and the next stage's AssignBin phase. This structure realizes a simple and often inefficient baseline in which each kernel fetches its bins according to Schedule, executes Process on them, and writes the output to the next stage's bins using the latter's AssignBin. The purpose of Piko's optimization step is to use static analysis and programmer-specified directives to find optimization opportunities starting with this baseline.

5.4.2.1 Kernel Fusion

Combining two kernels into one—"kernel fusion"—is a common optimization aimed at not just reducing kernel launch overheads, but also allowing an implementation to exploit producer-consumer locality between kernels. The same is true for Piko kernels, in cases where two or more successive functionally-distinct stages share similar computational characteristics.



Opportunity The simplest case for kernel fusion is when two subsequent stages (a) have the same bin size, (b) map primitives to the same bins, (c) have no dependencies/barriers between them, (d) receive input from only one stage and output to only one stage, and (e) contain Schedule phases that map execution to the same core. For example, a rasterization pipeline's Fragment Shading and Depth Test stages may pass this test. Once requirements are met, a primitive can proceed from one stage to the next immediately and trivially, so we fuse these two stages into one kernel. These constraints can be relaxed in certain cases (such as a scheduleEndBin dependency, discussed below), allowing for more kernel fusion opportunities. We anticipate more complicated cases where kernel fusion is possible but difficult to detect; however, detecting the simple case above is by itself significantly profitable.

Implementation Two stages, A and B, can be fused by merging A's *emit* and B's *fetch* and consequently eliminating a costly memory trip. We can also fuse more than two stages using the same approach. A code generator has several options for implementing kernel fusion, in decreasing order of desirability: a direct connection through registers within a single lane (what we most often use in practice); through a local (shared) memory within a single core; or

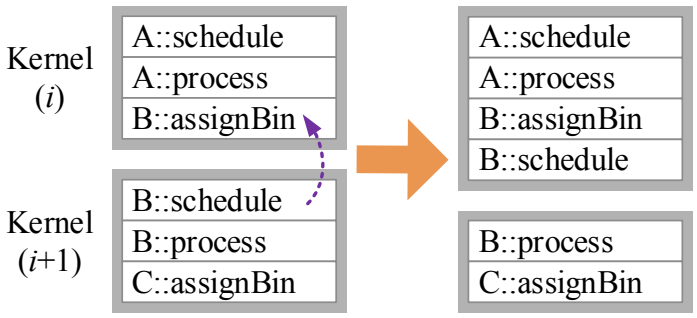
relying on on-chip caches to implicitly capture the producer-consumer locality.

Impact With some exceptions when we choose Schedule poorly, kernel fusion is generally a win in all the above cases. For instance, we automatically fuse all kernels of our strawman rasterizer if all instances of Schedule request dynamic load balance, producing an implementation that mirrors the FreePipe design [31]. It is a good choice for the Buddha scene, which consists of a large number of similarly-sized small triangles. It results in a 26% reduction in the runtime of a frame.

5.4.2.2 Schedule Optimization

While we allow a user to express arbitrary logic in a Schedule routine, we observe that the most common patterns of scheduler design can be reduced to simpler and more efficient versions. Two prominent cases include:

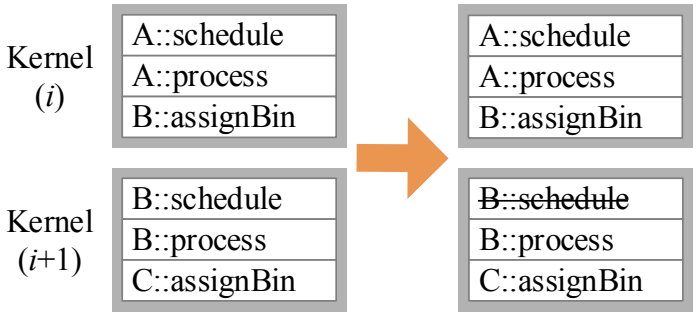
Pre-Scheduling



Opportunity For many Schedule phases, pairing a bin with a core is either static or deterministic given the incoming bin ID (specifically, when scheduleDirectMap, scheduleSerialize, or scheduleAll are used). In these scenarios, we can pre-calculate the target core ID even before Schedule is ready for execution (i.e. before all dependencies have been met). This both eliminates some runtime work and provides the opportunity to run certain tasks (such as data allocation on heterogeneous implementations) before a stage is ready to execute.

Implementation The optimizer detects the pre-scheduling optimization by identifying one of the three aforementioned Schedule policies. This optimization allows us to move a given stage's Schedule phase into the same kernel as its AssignBin phase so that core selection happens sooner and so that other implementation-specific benefits can be exploited.

Schedule Elimination



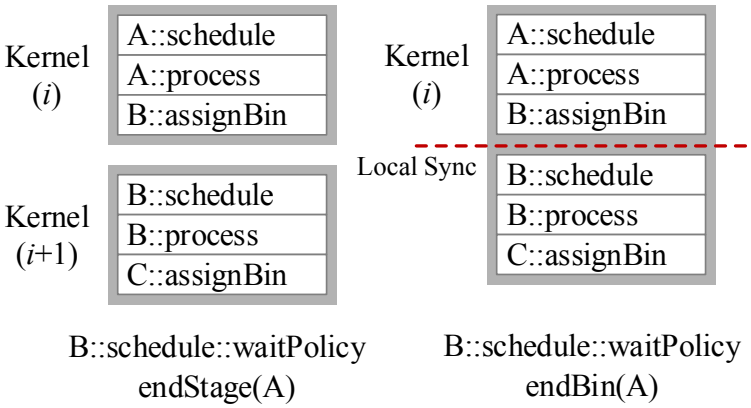
Opportunity Modern parallel architectures often support a highly efficient hardware scheduler which offers a fair allocation of work to computational cores. Despite the limited customizability of such a scheduler, we utilize it whenever it matches a pipeline's requirements. For instance, if a designer requests bins of a fragment shader to be scheduled in a load-balanced fashion (e.g. using the scheduleLoadBalance directive), we can simply offload this task to the hardware scheduler by presenting each bin as an independent unit of work.

Implementation When the optimizer identifies a stage using the scheduleLoadBalance policy, it removes that stage's Schedule phase in favor of letting the hardware scheduler allocate the workload.

Impact To measure the difference between the performance of hardware and software schedulers, we implemented two versions of Fragment Shader in our rasterizer pipeline: one using the hardware scheduler, and the second using an atomic counter to dynamically fetch work. Using a uniform Phong shader on the Fairy Forest scene, the hardware scheduler saved us 36% of fragment shader runtime on the GPU. Due to the relatively slow speed of atomic operations,

the same experiment on an Ivy Bridge architecture yielded savings of 67% in fragment shader runtime.

5.4.2.3 Static Dependency Resolution



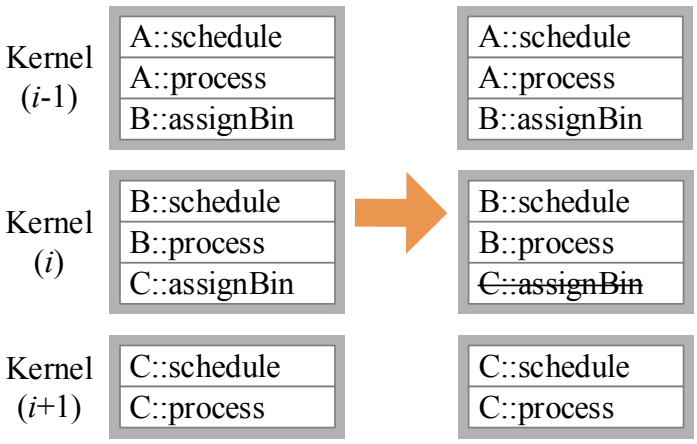
Opportunity The previous optimizations allowed us to statically resolve core assignment. Here we also optimize for static resolution of dependencies. The simplest form of dependencies are those that request completion of an upstream stage (e.g. the scheduleEndStage policy) or the completion of a bin from the previous stage (e.g. the scheduleEndBin policy). The former dependency occurs in rasterization pipelines with shadow mapping, where the Fragment Shade stage cannot proceed until the pipeline has finished generating the shadow map (specifically, the shadow map's Composite stage). The latter dependency occurs when synchronization is required between two stages, but the requirement is spatially localized (e.g. between the Depth Test and Composite stages in a rasterization pipeline).

Implementation We interpret scheduleEndStage as a global synchronization construct and, thus, prohibit any kernel fusion with the next stage. By placing a kernel break between stages, we enforce the scheduleEndStage dependency because once a kernel has finished running, the stage(s) associated with that kernel are complete.

In contrast, scheduleEndBin denotes a local synchronization, so we allow kernel fusion and place a local synchronization within the kernel between stages. However, this strategy only

works if a bin is not split across multiple cores using tileSplitSize. If a bin is split, then we must prohibit kernel fusion.

5.4.2.4 Bin Structure Optimization



Opportunity Often, two successive stages with the same bin size will assign primitives to the same bins. When this occurs, writing a new bin structure for the second stage would be redundant and inefficient. In the cases where these stages fuse into a single kernel, the redundancy is eliminated by the fusion. However, when these stages cannot fuse, we still have the chance to save memory overhead by eliminating unnecessary reads and writes.

Implementation We identify this optimization by identifying two successive stages with the same bin size that do not satisfy the criteria for kernel fusion. If the latter of the stages uses the assignOneToOneIdentity policy, then we can modify the implementation of the latter stage to use the bins of the former. The stage(s) following these two will then refer to the first stage's bins when retrieving primitives.

Impact To measure the impact of eliminating redundant bin structure updates, we implemented two versions of a Fragment Shader: one that wrote to new set of bins for the next stage (Depth Test), and another where the next stage reused Fragment Shader's bins. On a GPU, the performance improvement was about 20% of the frame run time, with an 83% improvement for

Fragment Shader. On our Ivy Bridge back end, the numbers were 5% and 15%, respectively. We believe that the difference in these numbers is primarily due to the relatively more capable cache hierarchy in the Ivy Bridge architecture.

5.4.3 Single-Stage Process Optimizations

Currently, we treat Process phases as architecture-independent. In general, this is a reasonable assumption for graphics pipelines, but it isn't accurate in all circumstances. We have noted some specific scenarios where architecture-dependent Process routines might be desirable. For instance, with sufficient local storage and small enough bins, a particular architecture might be able to instantiate an on-chip depth buffer, or with a fast global read-only storage, lookup-table-based rasterizers become possible. Exploring architecture-dependent Process stages is another area of future work.

5.5 Runtime Implementation

The output of the optimizer is a kernel mapping that lists the operations that form each kernel and a schedule to launch these kernels. The next step is a code generator that inputs a kernel mapping and generates output kernels as (for example) CUDA, PTX[3], OpenCL, or Direct3D Compute Shader. The scope of this dissertation encompasses hand-assisted code generation, i.e. we manually wrote the final code using the kernel mapping as our input. Our ongoing work addresses the issues in designing an automated code generator. We discuss some of those challenges in Chapter 6.

Despite hand-assisted code generation, we make most implementation decisions up front in a predictable, automated way. While Section 5.4 covers the pipeline synthesis step of our system, this section describes the runtime design for two fundamentally distinct architectures: a manycore GPU and a heterogeneous CPU-GPU architecture. We first describe common traits of our multi-kernel implementation strategy. We then discuss concerns that influence

[3]PTX (Parallel-Thread eXecution) is the intermediate representation for programs written in CUDA.

the design of pipelines for these architectures. Finally, we present some finer details of our implementation, specific to these architectures.

5.5.1 Common Implementation Strategy

Certain aspects of our runtime design span both architectures. The uniformity in these decisions provides a good context for comparing differences between the two architectures. For instance, we note that the degree of impact of optimizations from Section 5.4.2 varies between architectures, which helps us tweak pipeline specifications to exploit architectural strengths. Apart from the use of multi-kernel implementations, our runtimes share the following characteristics:

Bin Management For both architectures, we consider a simple data structure for storing bins: each stage maintains a list of bins, each of which is a list of primitives belonging to the corresponding bin. The latter may contain either actual primitives or references to primitives in a central structure. Storing actual primitives helps eliminate indirection during loading and storing primitives, while storing references helps reduce overhead when a single primitive belongs to many bins. We use a simple heuristic to balance the two concerns: whenever maxOutBins is larger than 2 and the size of individual primitives is larger than twice that of a reference (4 bytes in our case), we store references to primitives instead of values. Note that often our optimizations eliminate the need to write data to bins. Currently, both runtimes use atomic operations to read and write to bins. However, using prefix-sums for updating bins, while maintaining primitive order, is an interesting alternative.

Work-Group Organization In order to accommodate the most common scheduling policies of static and dynamic load balance, we simply package execution work groups in CUDA / OpenCL such that they respect the policies we described in Section 4.4:

LoadBalance As discussed in Section 5.4.2.2, for dynamic load balancing we simply rely on the hardware scheduler for fair allocation of work. Each packet (CUDA block / OpenCL work-group) represents a single incoming bin.

DirectMap In this case we also rely on the hardware scheduler, but we package data such

that each thread-group represents computation that belongs to a single core. In this way, computation belonging to the same core is guaranteed to run on the same physical core.

5.5.2 Architecture-Specific Implementation

Due to the distinct features of the two architectures and the accompanying programming languages, our implementations also differ for the two architectures. We first discuss the platform-specific aspects of our two implementations, and how we realize them using Schedule modifications. We then discuss details of how the two runtimes differ from each other.

5.5.2.1 Manycore GPU

Our manycore GPU implementation of Piko pipelines exclusively uses a discrete GPU for computation. The CPU acts as the host that (a) prepares the state (e.g. input scene, display) for the pipeline, (b) launches the kernels output from the code generator in the scheduled sequence, and (c) ensures that any cycles (resulting from either depth-first bin processing or inherent pipeline graph) run to completion. Our implementation uses CUDA on an NVIDIA GeForce GTX 460. This device has the following salient characteristics with respect to pipeline and runtime design:

- A high-end discrete GPU typically has a large number of cores. These cores, in turn, are organized in a wide-SIMD fashion, which can best be utilized by ensuring that we supply sufficiently large-sized chunks of uniform work to the GPU.
- Cache hierarchies on GPUs are relatively underpowered due to the GPU's emphasis on streaming workloads.
- Each core has a small user-managed local shared memory, which can be used to capture producer-consumer locality between fused kernels.

Influence on Pipeline Definitions The most important concern while writing Piko pipelines for a GPU is to keep all lanes of all cores busy. Thus, we prefer Schedule with a LoadBalance policy whenever appropriate. Further, for successive stages without a dependency, we try to

modify Schedule such that it collects enough primitives to fill all SIMD lanes. In a multi-kernel implementation, this primarily affects work organization within a fused kernel. In our example pipelines, we generally only prefer a DirectMap scheduling policy when we can make specific optimizations that use the local shared memory.

Influence on Runtime Design To maximize the efficiency of Piko implementations on a GPU, we make the following design decisions for the runtime:

- **Warp-synchronous Design** For DirectMap schedules, we use multiple warps (groups of 32 SIMD lanes) per core, but instead of operating on one bin at a time per core, we assign a separate bin to each warp. This way we are able to exploit locality within a single bin, but at the same time we avoid losing performance when bins do not have a large number of primitives.

- **Compile-Time Customization** We use CUDA templates to specialize kernels at compile-time. For example, the sizes of tiles for a pipeline as well as capacities of primitive buffers in our implementation are compile-time constants. Thus, each of our kernels inputs them as template parameters, resulting in some run time logic to be offloaded to compile time.

5.5.2.2 Heterogeneous CPU / GPU

A heterogeneous chip brings a CPU and GPU together on a single die. This design is geared towards low-power, efficient processors with moderately powerful graphics processors. We implemented Piko on an Intel Core i7-3770K Ivy Bridge and also experimented with AMD Trinity architecture. Both are heterogeneous processors which use OpenCL to target both the CPU and GPU (OpenCL "devices"). The two architectures are also fundamentally similar in their design: a single die containing both a CPU and a GPU, and a fast local memory for communication between the two. The AMD architecture differs from the Intel architecture in the balance between the capabilities of the CPU and the GPU. While the former has a powerful GPU paired with a CPU, the latter has a more balanced architecture with both CPU and GPU having similar capabilities. The experiments and results in this dissertation apply primarily to

the Intel Ivy Bridge architecture, but wherever relevant we note the qualitative differences to our implementation on AMD processors.

With a shared memory system between the two devices, heterogeneous processors allow work to be executed simultaneously on both. A Piko scheduler can divert some of the bins onto the CPU and run the kernel on both devices. Of the two processors, the CPU has higher priority over power resources and caches. When the CPU and GPU both require power that exceeds the available amount, the CPU will get the available power it needs. The CPU OpenCL drivers allow full usage of the CPU's caches, but on the GPU, the L1 and L2 caches are unavailable from OpenCL and are instead reserved for driver-specific uses. All of these CPU advantages make offloading parts of the kernel workload to the CPU an attractive idea.

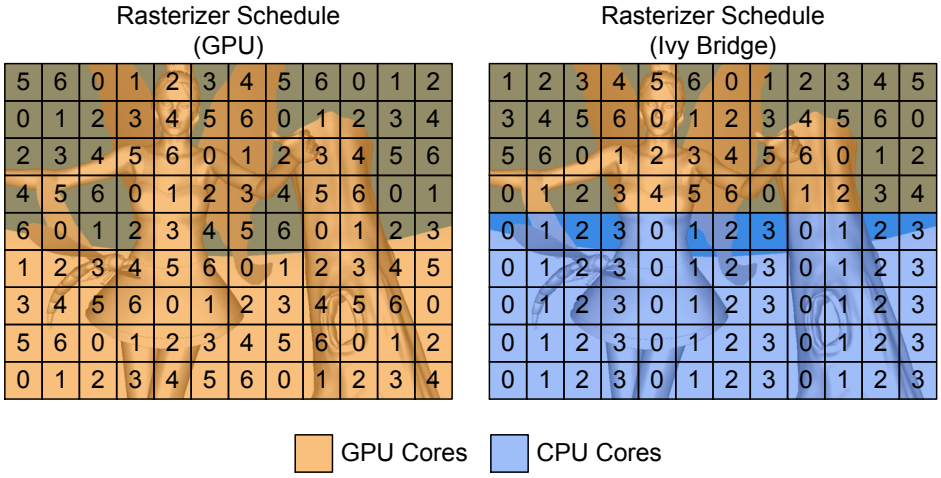


Figure 5.4: *Piko supports using multiple architectures simultaneously. On the left, Schedule for a discrete GPU requests load-balanced scheduling across the device. However, on the right, we maximize utilization of an Ivy Bridge architecture by partitioning the framebuffer into two distinct contiguous regions, one handled by CPU cores and the other by GPU cores.*

The Piko scheduler on Ivy Bridge sends work to both the CPU and GPU simultaneously to efficiently process a pipeline stage (Figure 5.4). The scheduler determines a split ratio of work between the two devices, partitions the bins, and launches a kernel on each device.

Split Ratio In choosing the partition point, we aim to minimize the difference in execution times on both devices while keeping the overall execution time low. This split ratio is determined mainly by power and CPU load, and these can be difficult to poll accurately at runtime

for the scheduler to make a decision.

Our solution is to find the split ratio by profiling each of the kernels, as in Luk et al. [32]. Profiling the kernels allows the scheduler to find the optimal split for each kernel. The scheduler starts with a 50/50 split ratio between the two devices and then adjusts the workload accordingly until the difference in execution time is minimized. A rough average across our kernels was a 60/40 CPU/GPU split between an 8 core CPU and a 16 core GPU on the Intel Ivy Bridge. For the AMD Trinity architecture, the best implementation had a 0/100 CPU/GPU split, i.e. the fastest pipeline ran entirely on the on-die GPU.

Bin Partitions Once the split ratio is determined, then the next step is to partition the bins between the two devices. In our implementation, bin partitions follow two rules: devices have exclusive ownership of a partition of bins, and the bin partition on a device lies in a contiguous region of memory.

For hardware memory consistency on writes, each device owns its own set of bins, though either processor can read from either partition. Memory consistency is only guaranteed *after* kernel completion since the caches on both devices need to be flushed. This means that both devices cannot write to the same bin at the same time.

In the worst case, the Piko scheduler will duplicate the set of bins in the next pipeline stage so that the CPU and GPU both own their own set of bins. We can often apply an (automatic) optimization: if the primitives in bin A will only scatter to a specific set of bins B in the next stage and B will only get its input from A, then a device can own both A and B. (For example, a coarse-raster 128×128 bin will only scatter to the set of fine-raster 8×8 bins that cover the same area.) In this situation, a single device can own both A and B, requiring no extra duplication of bins.

In Piko, bin partitions are contiguous primarily because of memory addressing issues in OpenCL, which does not support pointers but instead handles, making disjoint regions in memory difficult to manage. This hinders our choices for load-balance strategies. Currently we cannot, for example, interleave bins across CPU and GPU, though a more fine-grained

tiling across the screen would almost certainly benefit overall load balance. As heterogeneous programming toolkits advance and we can distribute more fine-grained work across devices, we expect to improve the load-balance behavior of Piko-generated code.

5.6 Results

We evaluate our system from four distinct perspectives: programmability, efficiency, portability, and flexibility. Our primary test implementation consists of a set of different scheduling choices applied to a forward raster pipeline, but we also study a load-balanced deferred renderer and a load-balanced Reyes renderer. Appendix A presents a study showing the synthesis results for a broader set of Piko pipelines, including a deferred triangle rasterizer, a ray tracer, a Reyes micropolygon pipeline, and a hybrid-rasterizer-ray-tracer.

We implemented four primary versions of a traditional forward raster pipeline: rast-strawman, rast-freepipe, rast-loadbalance, and rast-locality. From the perspective of Piko pipeline definitions, rast-strawman and rast-freepipe use fullscreen tiles throughout the pipeline (i.e. no tiling), and all stages specify a LoadBalance schedule. On the other hand, rast-locality and rast-loadbalance use full-screen bins for Vertex Shade, 128 $\times$ 128-pixel bins for Geometry Shade, and 8 $\times$ 8-pixel bins for all other stages (Rasterize, Fragment Shade, Depth Test, and Composite). While rast-locality emphasizes locality in Fragment Shade by preallocating cores using a DirectMap scheme, rast-locality uses a LoadBalance scheme to maximize core utilization. For details on the policies used for each pipeline, please refer to Appendix A.

From the perspective of pipeline synthesis, rast-strawman does not utilize any of our optimizations and runs just one kernel per stage. rast-freepipe (based on FreePipe [31]), on the other hand, fuses all stages into a single kernel. rast-locality's preferences result in a fused kernel that performs Rasterize, Fragment Shade, Depth Test, and Composite without writing intermediate results to off-chip memory. rast-loadbalance, on the other hand, dynamically load-balances the Fragment Shade stage into its own kernel that reads rasterized fragments from off-chip memory.

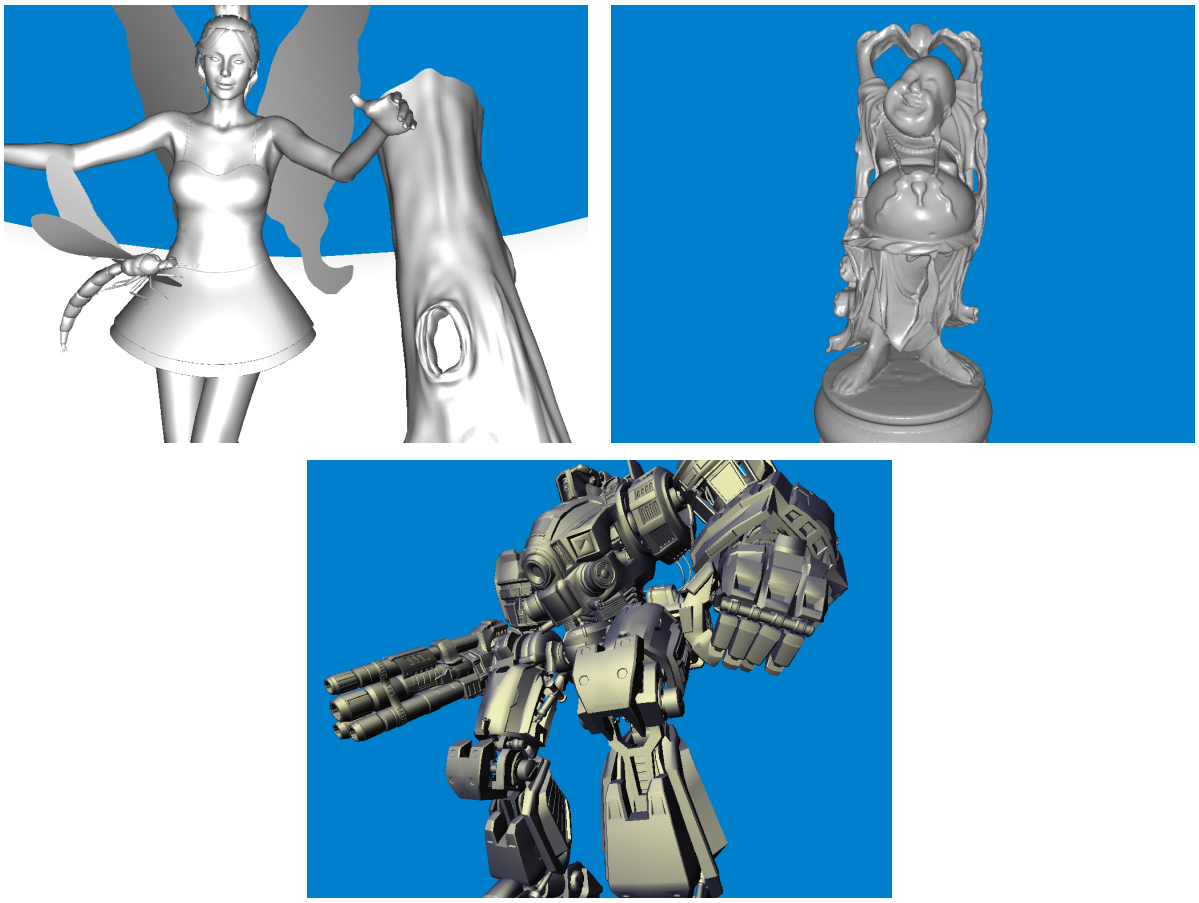


Figure 5.5: *We use these scenes for evaluating characteristics of our rasterizer implementations. Fairy Forest (top-left) is a scene with 174K triangles with many small and large triangles. Buddha (top-right) is a scene with 1.1M very small triangles. Mecha (bottom) has 250K small- to medium-sized triangles. Each image is rendered at 1024×768 resolution. Thanks to Ingo Wald for the Fairy Forest scene [51], to Stanford Computer Graphics Laboratory for the Happy Buddha model [47], and to AMD for the Mecha model.*

5.6.1 Programmability

Perhaps the most important concern of many designers and one of our primary motivations is the ease of using Piko to implement traditional as well as novel graphics pipelines. The evidence we supply for programmability is simply that we can use Piko to express and synthesize numerous pipelines, which we detail in Appendix A, including multiple rasterization pipelines, a Reyes-style renderer, a deferred renderer, and a hybrid rasterization-ray tracing pipeline.

Target	Discrete GPU (ms/frame)			Ivy Bridge (ms/frame)		
	Fairy Forest	Mecha	Buddha	Fairy Forest	Mecha	Buddha
rast-strawman	74.4	25.3	14.3	325.07	147.85	101.6
rast-freepipe	185.3	50.6	10.6	603.89	298.51	86.6
rast-locality	42.6	48.7	43.4	51.97	141.15	87.3
rast-loadbalance	20.3	22.3	32.0	91.45	158.18	125.9

Table 5.1: *Rendering times (in milliseconds per frame) of different scenes on different Piko-based rasterization pipelines. Note how scene characteristics influence the relative performance of rasterizers built with varying priorities. Please see Section 5.6.2 for details.*

5.6.2 Efficiency

Our second priority is performance; we would like to ensure that pipelines written using Piko achieve reasonable efficiency without requiring a heroic programming effort. To demonstrate this, we used Piko to implement a triangle rasterization pipeline along the lines of cudaraster [26].

We ran our pipeline on three different scenes: Fairy Forest, Buddha, and Mecha (Figure 5.5). Of these three scenes, Fairy Forest has the widest variation of triangle sizes, while Buddha consists of over a million very small rather uniform triangles.

Table 5.1 shows the runtimes of our 4 different forward-raster pipelines. First, we note that we are able to run each scene on either processor at (roughly) interactive rates on scenes of reasonable complexity, indicating that Piko produces software-generated pipelines of reasonable efficiency with no special-purpose hardware support. For the Fairy Forest scene, our best implementation is about 3—4$\times$ slower than cudaraster, state of the art in hand-optimized GPU software rasterization. Second, our optimizations are effective: consider the Fairy Forest scene (with the broadest triangle distribution) on the GPU, for instance, and note that the performance optimizations between the simple strawman or Freepipe rasterizers compared to the locality and loadbalance rasterizers produce speedups between 2$\times$ and 9$\times$. Third, we motivate programmable pipelines in general by noting that different scenes benefit from different rasterizer formulations. Figures 5.6 and 5.7 estimate how varying scheduling policies affect the extent of exploiting locality and load balance in a pipeline, respectively. In the former, we

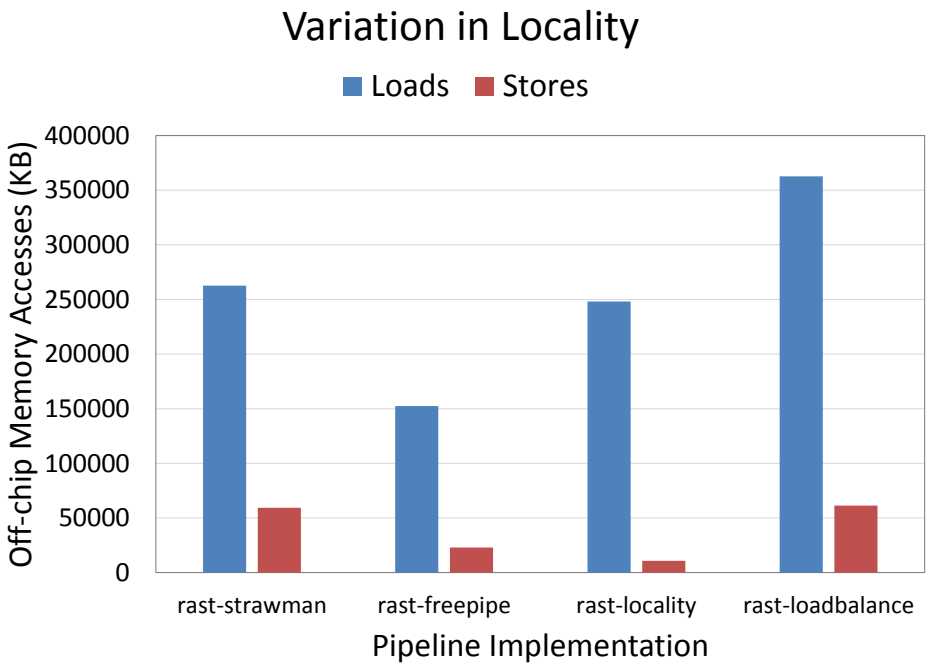


Figure 5.6: *Different Schedule policies allow varying the degree of exploiting locality in a pipeline. Here, we show how the aggregate off-chip accesses change as we vary pipeline expression for rendering the Fairy Forest scene. The two leftmost bars are instances of our strawman, without any tiling infrastructure. rast-strawman is a simple sequence of one kernel per stage, and rast-freepipe is a single kernel with all stages fused. rast-locality uses tiling but due to a locality-preserving Schedule, has relatively fewer loads and stores than both strawman pipelines. rast-loadbalance, on the other hand, uses many more loads and stores due to more frequent work redistribution.*

note how despite the overhead of 2D tiling, rast-locality reduces the aggregate loads and stores when compared to rast-strawman. In the latter, we note how simple kernel fusion can result in load imbalance for rast-freepipe, and hence even though it is a great implementation choice for scenes with a large number of uniformly-sized triangles, it is bad for scenes with more irregular workloads. rast-locality emphasizes static load balance and is hence much better, but if load balance is our most important goal (as is the case in a modern GPU), rast-loadbalance is probably the best choice. More broadly, future rendering systems may benefit from choosing pipeline formulations that are customized to a particular scene or goal.

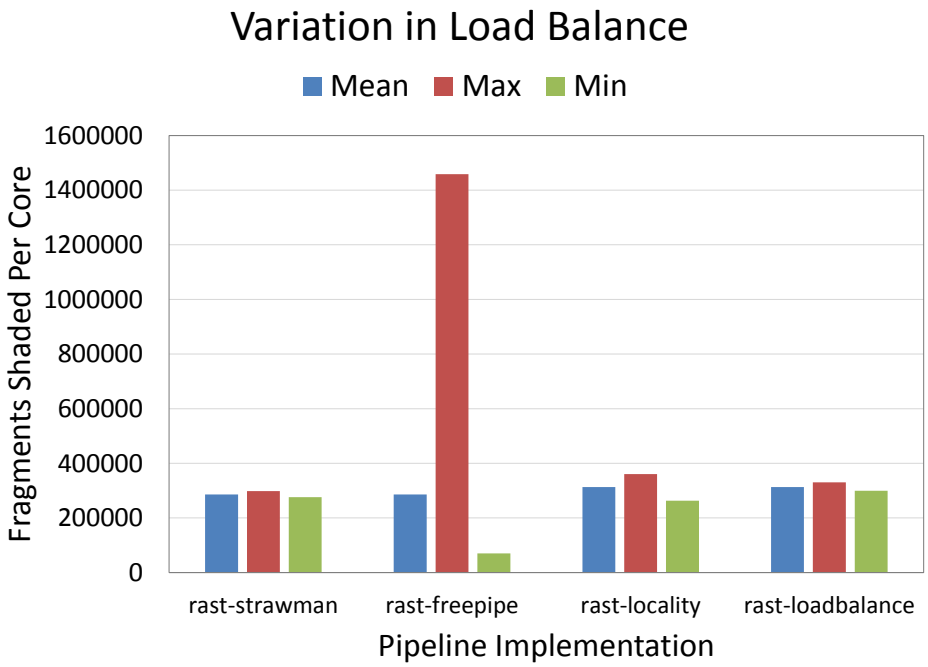


Figure 5.7: *By measuring the number of fragments shaded by each core, we can estimate the impact of Schedule preferences on the load balance of a GPU-based triangle rasterizer. For the Fairy Forest scene, our strawman (first) exhibits low variation in the work per core. A FreePipe-based rasterizer (second) shows a high degree of variation due to the lack of any work redistribution. rast-locality (third) uses a DirectMap scheduling policy, which enforces static load balance and hence the difference between work per core is not large. rast-loadbalance (fourth) on the other hand, uses a hardware-enabled dynamic scheduling which provides a slightly better load balance.*

5.6.3 Flexibility

One of the primary benefits of programmable pipelines is the potential reuse of pipeline stages (and/or phases) across multiple pipelines. Recall that we built four rasterization pipelines with similar overall structure, but that differ in the bin structure and the AssignBin and Schedule phases. We achieve substantial reuse across these implementations: Using the MOSS "measure of software similarity" [45], we observe that rast-locality and rast-loadbalance share over 75% of the final lines of code on the GPU, and over 63% of the final lines of code on Ivy Bridge.

Our Piko implementations also reuse stages across pipelines that implement different rendering algorithms. With AssignBin and Schedule in charge of preparing work granularity and execution priorities for Process, many stages are easily transferable from one pipeline to another. The

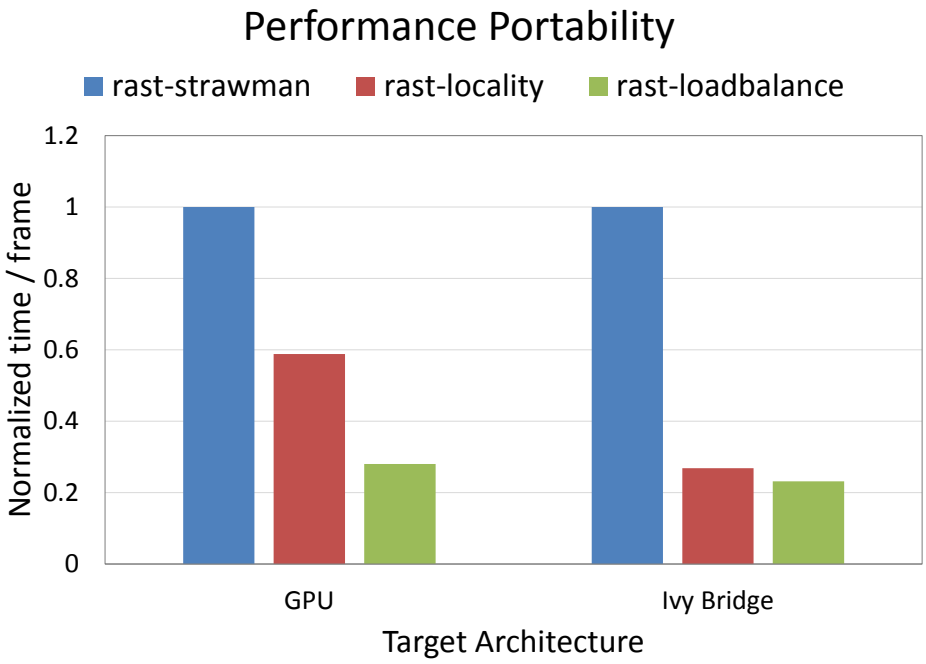


Figure 5.8: *The variation in performance of rendering Fairy Forest using our rasterizer implementations on a GPU compared and those on an Ivy Bridge processor. Due to the difference in the capabilities of the two devices, we have normalized the y-axis to the performance of rast-strawman. Notice how the locality-preserving rasterizer improves in relative performance when we move to an architecture with a more capable cache hierarchy.*

easiest of these stages are shader stages like Fragment Shader. In Appendix A, we color-code stages to show how we reuse them across multiple pipelines.

5.6.4 Portability

Portability enables efficient implementations across multiple architectures without requiring a completely redesigned implementation. Given the significant hardware variations across modern devices, we certainly do not expect to achieve this with no changes to the design. However, decoupling scheduling priorities and constraints using Schedule separates most of the architecture-dependent aspects, and in most cases that is all we need to modify from one architecture to another. We demonstrate this by porting rast-strawman, rast-locality, and rast-loadbalance to our Ivy Bridge target by appropriately modifying only the Schedule routines.

Figure 5.8 shows the results of rendering Fairy Forest with the runtime normalized to the straw-

man for each architecture. The similarity in performance behavior for the three implementations suggests that pipelines designed using Piko are portable across different architectures. We note a peculiar effect during this transition: while rast-loadbalance is the best overall choice for both a discrete GPU and heterogeneous CPU-GPU architecture, rast-locality is particularly beneficial for the latter. The heterogenous chip benefits more from the locality-preferring scheme for two reasons: the CPU is responsible for a significant chunk of the work and benefits from per-processor locality, and the CPU has a more powerful cache hierarchy. Specifying these pipelines for both targets, however, uses an identical pipeline definition with mainly API-specific changes (CUDA vs. OpenCL) and the necessary architecture-dependent differences in Schedule (for instance, those that allow the Ivy Bridge to use both the CPU and GPU).

Chapter 6

Conclusion

This dissertation proposes an alternate abstraction for graphics pipelines, using practices that are already used across implementations, but are usually not exposed to application programmers. In the past chapters we presented the development and evaluation of this abstraction:

- We presented a detailed survey of graphics pipelines and characteristics of their realizations across a several generations. From this survey we recognized a set of underlying practices that helped achieve efficiency goals in these implementations.

- We designed an abstraction by extending a conventional pipeline model to accommodate a programmable expression for the above principles. The extension consists of expressing a pipeline stage using three phases: AssignBin, Schedule, and Process, and using these phases to tune the behavior of an implementation to match those of the application and the underlying architecture.

- We outlined an implementation framework that can take pipelines expressed using Piko, and generate high-performance software implementations for two target platforms. We studied several optimization principles in an architecture-independent as well as architecture-specific contexts. We evaluated our system by observing the relative behavior of four different implementations of a rasterization pipeline written using Piko.

In the process of building an abstraction for graphics pipelines and the implementation strategy for the abstraction, we made several design choices, and having completed that process once, we were able to re-evaluate them in hindsight. The following sections provide a discussion of these thoughts. Specifically, for the abstraction and our implementation strategy, we present our inferences, some limitations, and avenues of future work.

6.1 Abstraction

In a nutshell, Piko's abstraction provides an interface for a programmer to extend a pipeline's functional definition to include policies to allow constructing batches of computation, and scheduling those batches onto available cores. In hindsight, adding distinct phases for this purpose was an appropriate choice. Once we had made that decision, it became easy for us to distinguish between various pipeline implementations by simply assessing the differences between specific phases. For example, the presence of a ScheduleAll keyword distinguishes bucketing or chunking renderers from those that work on the entire screen. We are also happy about our choice of using directives to express user preferences. Although this puts our phases somewhere between being purely imperative and purely declarative, it lets a programmer describe commonly-used policies alongside full programs to describe new and complicated pipeline execution.

6.1.1 Limitations

We acknowledge the following limitations with the abstraction presented in Chapter 4:

- **Use of 2D Uniform Tiling** To keep things simple, Piko only supports 2D uniform screen-tiles. This simple choice covers a large variety of graphics pipelines. However, to support a wider class of graphics pipelines and even non-graphics pipelines like those in physics simulation, we need to provide a more natural domain for computation in these pipelines. Specifically, we would like to extend Piko to include tiles in three dimensions, non-uniform tiles, adaptive tiles, and tiles in non-spatial domains.

- **Decoupling Functionality from Efficiency** To be truly architecture-independent, Piko pipelines do not allow architecture-specific optimizations in the Process and AssignBin phases. This limits the use algorithms and techniques that may benefit from the use of an architecture's unique features, e.g. special hardware instructions or local memory storage. Portable Piko pipelines are restricted to use architecture-independent functionality in these two phases.

6.1.2 Future Work

We hope to continue to build on lessons learnt during this effort. In the future, we would like to improve our abstraction in the following directions:

- **Language Design** For our current work, we did not pursue the design of a language to describe Piko phases, since it was orthogonal to our goals. However, we feel that it would be an interesting target for future work, specifically because it involves expressing logic that must run efficiently on multiple architectures.

- **The role of Schedule** We realized that there were two shortcomings with the way we defined Schedule. Firstly, it couples correctness concerns (dependencies) with efficiency concerns (scheduling policies), causing it to be a part of the functionality of the pipeline, and making it harder to port a pipeline from one architecture to another. Secondly, it is unclear where Schedule should run. In our implementations, it can either run with the previous stage, with the current stage, in its own kernel, or centrally on the host (using appropriate directives). This variety makes implementations much harder to realize, and in a future version of our abstraction, we would like to decouple the two concerns. For example, for order-independent transparency we always have the scheduleEndBin dependency for correctness. Yet, it is included in the architecture-specific Schedule phase. In an improved version, we might include such a dependency as a part of the Process or AssignBin phase.

- **Centralized Schedule** If we were to rethink the entire abstraction, it would be interesting to study a pipeline abstraction that, in contrast to our current three-phases-per-stage model, partitions task organization and scheduling into a single, central program for the entire pipeline. In choosing this direction, we are inspired by Halide [44], an abstraction for image-processing algorithms that allows for a central schedule. Halide's inputs differ from ours in two ways. First, the primary input primitive is an image, and second, the input algorithmic pipeline is often linear and very short. A central schedule is appropriate in this scenario, yet we would like to explore an abstraction for graphics pipelines which defines the AssignBin and Process phases as in Piko, but lets the programmer write a single program that contains the scheduling policies and constraints for the entire pipeline. For example, if several stages in succession use the same tiling and scheduling policies, a central might express this structure more compactly than the AssignBin and Schedule phases of all stages.

6.2 Implementation

As described in Chapter 5, our implementation framework seeks to translate pipelines expressed in Piko into optimized implementations. Figure 4.2 describes this process, which is divided into three steps: analysis, synthesis, and code-generation. In hindsight, we feel satisfied with this division, as it clearly separates the goals during each step. The choice also helped us prioritize our work—studying synthesis offered the most insight, so we invested maximum efforts in that direction.

6.2.1 Ongoing Work

Implementation strategy is an area where we have multiple ongoing efforts:

- **Automated code generation** We are in the process of developing an automated code generator, which can take the kernel mapping and auotmatically generate final code for a GPU architecture. Ongoing efforts are aimed at reaching a point where the path

from the pipeline abstraction to the final implementation requires no hand-assistance. Currently we are able to compile simple pipelines in this manner, and are working towards supporting more complicated ones.

- **Pipeline Study** Automatic code generation enables us to start investigating a wider variety of graphics pipelines. In particular, we are exploring pipelines that perform photon mapping, voxelized rendering, and a hybrid Reyes-ray-tracing pipeline.

6.2.2 Future Work

Our primary investment of effort in the future lies in improving our implementation framework. In this pursuit, we consider the following to be the most valuable directions of work:

- **Dynamic Scheduling** It would be interesting to experiment with alternatives to a multi-kernel programming model. A persistent-threads model, for example, would let us dynamically schedule and fetch work in a pipeline. CUDA dynamic parallelism [37] or programming models supporting similar dynamic task-generation would let us support dynamic work-granularities. Such implementation would help realize more complicated scheduling programs, for instance a dynamic work-stealing scheduler.

- **Hardware Targets** We would like to be able to compile Piko pipelines to hardware-description languages like Verilog, so that we can evaluate the abstraction's capabilities in navigating architectural choices rather than generating software implementations.

- **Optimal Branch Scheduling** It would be interesting to investigate improvements to our heuristics for scheduling branches of a pipeline. Even though in practice we get good results with most graphs, a more intelligent strategy might take into account the size of intermediate state, the length and frequency of cycles in the pipeline, and dependencies between branches.

- **Limits of static optimizations** Most of our optimizations are driven by a combination of static analysis and user-provided directives. Thus, synthesis is likely to miss opportunities

for optimizing a pipeline based on its dynamic behavior.

- **Further Evaluation** In the pursuit of exploring ideal abstractions for computer graphics, we hope to perform more detailed studies on Piko to understand the behavior of our implementations. We would like to implement a much bigger class of example pipelines and their variants, and target them to a wider class of platforms including single- and multi-core CPUs. These experiments would provide a deeper insight into how changing AssignBin and Schedule preferences affect pipeline efficiency for different architectures, and help us gain a comprehensive understanding of the limits of Piko.

6.3 Summary

To conclude this dissertation, we re-assert the importance for a high-level abstraction for programmable pipelines. As computer graphics continues to evolve in diversity and complexity, programmable pipelines offer a new opportunity for existing and novel rendering ideas to impact next-generation graphics systems. Our contribution, the Piko framework, addresses one of the most important challenges in building programming abstractions: how to achieve high performance without sacrificing programmability. To this end, we have spent considerable effort in trying to distill high-level principles that contribute to the efficiency of existing graphics implementations. Piko's main design decisions are the use of spatial tiling to balance parallelism against locality as a fundamental building block for programmable pipelines, and the decomposition of pipeline stages into AssignBin, Schedule and Process phases to allow high-level performance optimizations and enhance programmability and portability.

We envision Piko to be a helpful abstraction for both beginners and experts. For beginners, it allows assembling existing pipeline stages into novel pipelines and optimizing the implementations by tweaking the AssignBin and Schedule phases. For the experts, Piko provides control over the underlying organization of how work is grouped and scheduled onto execution cores, enabling finer optimization. In our opinion, such an abstraction would be beneficial in areas beyond computer graphics as well. We can imagine large complicated systems being built

out of pipelines just like graphics, with their work-granularity and schedule optimized using phases similar to AssignBin and Schedule.

We believe the Piko is the first step towards truly flexible high-performance graphics pipelines. However, there are several directions of potential improvement to the abstraction as well as its realization. One important contribution of this dissertation to offer a new way to think about implementing graphics applications—by incorporating spatial tiling as a first-class construct, and using it to find the balance between locality and parallelism. We hope that future abstractions for graphics pipelines will employ similar constructs for enabling a high-level control of efficiency, and allow flexible re-purposing of pipelines across a diverse set of implementation scenarios.

Appendix A

Example Pipeline Schedules

Here we present examples of how our implementation of Piko's synthesis step applies to some example graphics systems. In Section 4.2 we discussed that the input to this step is a pipeline skeleton, or a summarized representation of the pipeline's topological as well as functional characteristics. Section 5.4 discusses the optimizations that we use to reorganize the skeleton and output an optimized kernel mapping.

We wish to clarify that the goal of this Appendix is to exclusively study the behavior of the synthesis step in our system, since this is where we perform the bulk of our optimizations. The input in these examples is a pipeline skeleton, and the output is the kernel mapping. We only implemented a subset of these as final programs on our target architectures. Specifically, we did not implement the ray tracing and hybrid-rasterization-ray-tracing pipelines. Implementation-specific concerns for this class of pipelines are topics of future exploration.

We start by presenting the various flavors of a forward triangle rasterizer from Table 5.1, showing how different phase specifications affect output schedule. We then present the schedule for a deferred shading rasterizer, studying three of its real-world instantiations: load-balanced deferred-shading, bucketing deferred-shading, and deferred-shading using fixed-function GPU-rasterization. Following that we present the schedule for a Reyes renderer in many-core load-balanced and a bucketed configurations, and a simple ray tracer. Finally,

Short name	Expansion	Pipelines using the stage
VS	Vertex Shade	Forward Rasterization, Deferred Rasterization
GS	Geometry Shade	Forward Rasterization, Deferred Rasterization
Rast	Rasterize	Forward Rasterization, Deferred Rasterization
FS	Fragment Shade	Forward Rasterization, Deferred Rasterization
ZTest	Depth Test	Forward Rasterization, Deferred Rasterization
Comp	Composite / Blend	Forward Rasterization, Deferred Rasterization, Reyes, Ray Trace, Hybrid Rasterizer-Ray-trace
DS	Deferred Shade	Deferred Rasterization
OGL	OpenGL / Direct3D	Deferred Rasterization
Split	Split	Reyes
Dice	Dice	Reyes
Shade	Surface Shade	Reyes, Ray Trace, Hybrid Rasterizer-Ray-trace
Sample	Micropolygon Sample	Reyes
GenRays	Generate Rays	Ray Trace
Int	Intersect	Ray Trace

Table A.1: *A list of short names for pipeline stages*

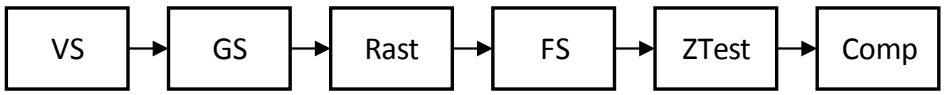
we show the schedule for a hybrid pipeline with a rasterizer augmented to perform ray tracing during its fragment-shading stage.

In the interest of clarity, the diagrams in this appendix use shortened names of pipeline stages. We refer the reader to Table A.1 for expansions of short names of stages in our library, and the pipelines which use each of those stages.

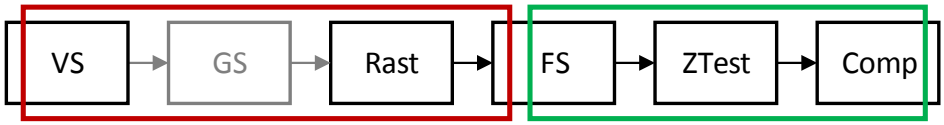
A.1 Forward Triangle Rasterization

We start with a high-level description of the pipeline as shown in Figure A.1, and present the results of synthesis three Piko implementations from Section 5.6: rast-strawman, rast-locality, and rast-loadbalance. Figures A.10, A.11, and A.12 show the directives used to implement these pipelines. In Figure A.1 (b), (c), and (d), we show how synthesis uses the directives to group phases of successive stages into kernels. For detailed kernel plans, see Figures A.10, A.11, and A.12.

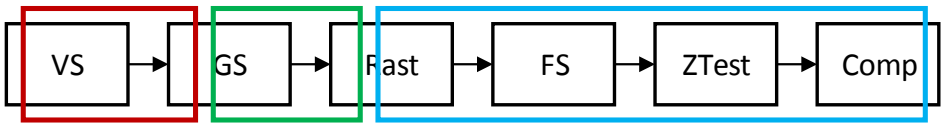
Note that for simplicity the Geometry Shader (GS) stage in all our implementations of rasteri-



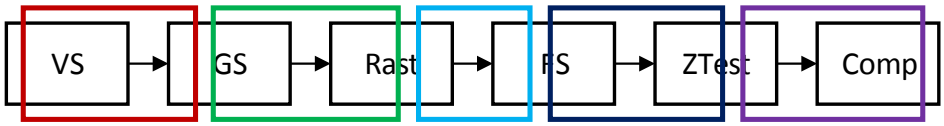
(a) Base pipeline



(b) Strawman Schedule



(c) Locality Schedule



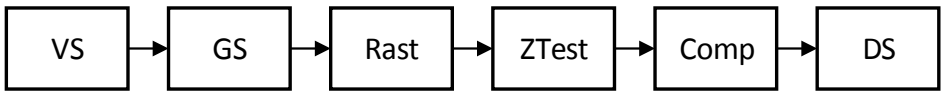
(d) Load-balanced Schedule

Figure A.1: *High-level organization of a forward rasterization pipeline, and kernel mapping for three varying Piko abstractions. The directives for these implementations are covered in Figures A.10, A.11, and A.12. Note that the GS stage, which has an empty Process phase but a non-empty AssignBin and Schedule, is completely absent in rast-strawman.*

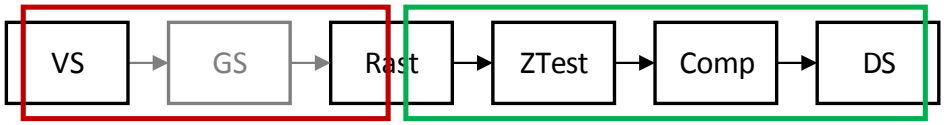
zation has an empty Process phase. However, we do use its AssignBin and Schedule phases to divide transformed triangles into 128 $\times$ 128 tiles. Since rast-strawman does not employ any tiling, this stage is completely absent (and hence greyed) in that implementation.

A.2 Triangle Rasterization With Deferred Shading

A popular variant of forward rasterization pipeline is a deferred shading pipeline. This pipeline defers shading computations until after hidden-surface removal, which helps in achieving a predictable and reduced shading workload, and in modern applications help realize complex illumination effects involving many lights. Deferred shading pipelines are currently in wide use



(a) Bucketed deferred shading rasterizer



(b) Kernel mapping for bucketed deferred shading rasterizer

Figure A.2: *High-level organization of a bucketed deferred shading rasterizer, based on a modern mobile GPU architecture [25]. Please see Figure A.13 for details. In order to match the mobile GPU's design, we omit the FS stage. The GS stage has identity AssignBin, Schedule, and Process phases, so it is omitted as well.*

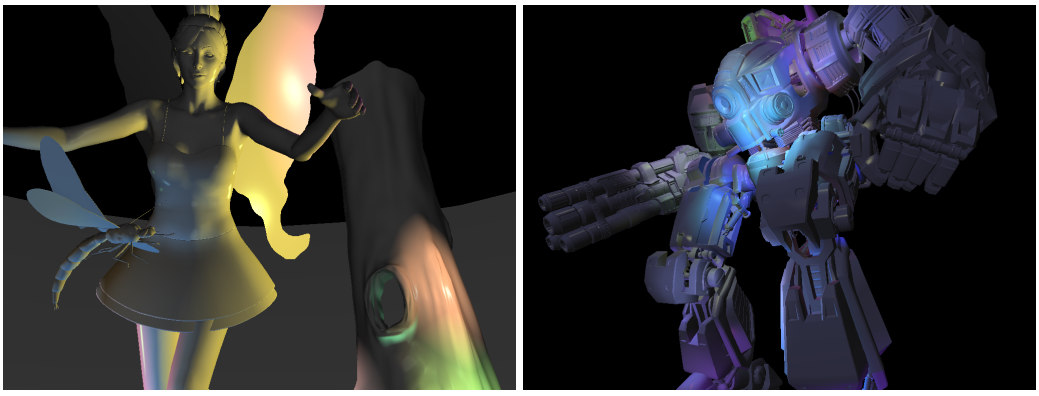
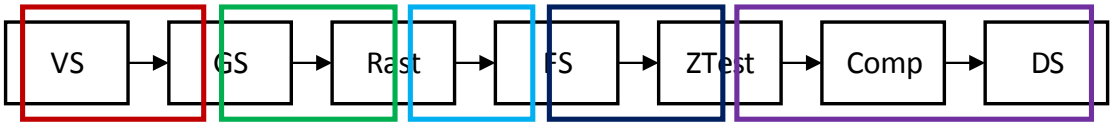


Figure A.3: *Scenes rendered using a load-balanced implementation of the deferred shading pipeline described in Figure A.4.*

in gaming, but are not natively supported by the contemporary GPUs. Consequently, current implementations are completely customized by programmers of game engine with little help from modern APIs.

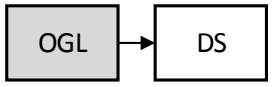
Figure A.2 shows a version of deferred shading that models a PowerVR mobile GPU [25]. Basing our design on the PowerVR pipeline, we omit the Fragment Shade (FS) stage. The Geometry Shade (GS) stage also gets eliminated due to identity AssignBin, Schedule, and Process phases. The front-end of this pipeline performs bucketing or tiling to reduce bandwidth usage, and the back-end performs deferred shading on rendered tiles. Figure A.13 shows the directives used as well the final kernel plan.

(a) Load-balanced deferred shading rasterizer

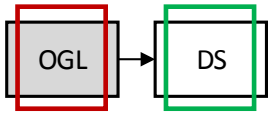


(b) Kernel Mapping for load-balanced deferred shading rasterizer

Figure A.4: *High-level organization of a load-balanced deferred shading rasterization pipeline. Please see Figure A.14 for details. Being a simple extension of our rast-loadbalance pipeline, this pipeline retains the GS and FS stages.*



(a) OpenGL-based deferred shading rasterizer



(b) Kernel Mapping for OpenGL-based deferred shading Rasterizer

Figure A.5: *High-level organization of a OpenGL / Direct3D deferred shading rasterizer. The OGL or OpenGL/Direct3D pipeline stage is shown in gray to denote that its implementation represents a fixed-function GPU pipeline. Please see Figure A.15 for details.*

Another implementation of deferred shading simply modifies our forward rasterization pipeline by adding a DS (Deferred Shading) stage at the end of the pipeline (Figure A.4). Directives for implementing such a pipeline, and the output kernel plan, are shown in Figure A.14. Being a simple modification to our previous forward rasterization implementation, this pipeline retains the Geometry Shade (GS) and Fragment Shade (FS) stages. Figure A.3 shows images of scenes rendered using this pipeline.

If we simply want to add another stage at the end of the rasterization pipeline, we have the option of using a fixed-function GPU pipeline before we perform deferred shading. This strategy is

Figure A.6: *Big guy model rendered using a load-balanced implementation of the Reyes pipeline described in Figure A.7. Big Guy model courtesy of Bay Raitt, Valve Software.*

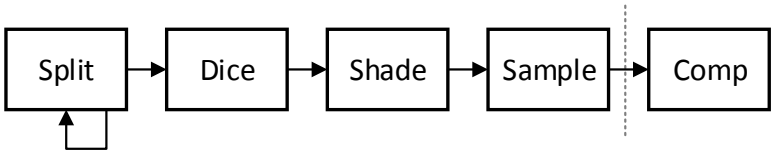
highly popular in today's video game engines. Considering the OpenGL / Direct3D pass that manages the forward pass as a single black-box stage, we can realize the pipeline as shown in Figure A.5. Please see Figure A.15 for the directives and kernel plan.

A.3 Reyes

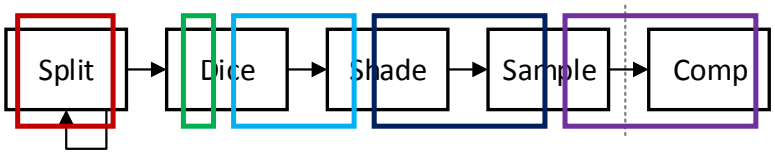
Figure A.7 shows Reyes, a pipeline designed for high-quality offline rendering. Note the dynamic-recursive Split stage, and the synchronization between Sample and Comp stages. We implement two versions of a Reyes pipeline in Piko: a many-core load-balanced Reyes pipeline, and a bucketed Reyes pipeline based on RenderAnts [52]. Figures A.16 and A.17 show the directives and kernel plans for the two versions, respectively. Figure A.6 shows an image rendered using the load-balanced implementation.

A.4 Ray Trace

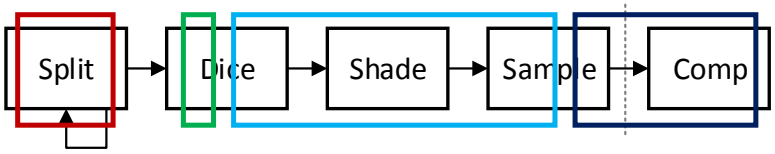
Figure A.8 shows a simple ray tracing pipeline, and how our synthesis tool groups stages together according to directives from Figure A.18. The tool inserts a loop around the shade-intersect cycle to allow for dynamic ray-generation and consumption.



(a) Reyes pipeline (base)



(b) Load-balanced Reyes pipeline



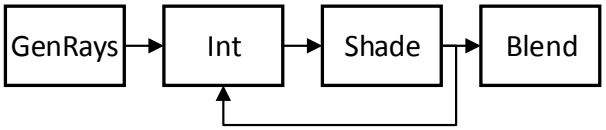
(c) Bucketed Reyes pipeline

Figure A.7: *High-level organization and kernel mapping of a Reyes pipeline. The dotted line indicates a dependency achieved using* ScheduleEndBin*. Please see Figures A.16 and A.17 for details about these pipeline definitions.*

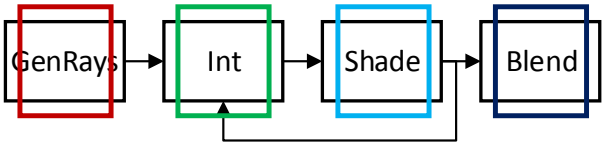
In this version of the Piko ray tracing pipeline, we maintain 16 $\times$ 16 tiles for all rays. In other words, even after Shade generates secondary rays, we maintain them in the same tiles as their corresponding primary rays. This helps preserve locality for systems which perform only one or two ray bounces, but for systems with more bounces and hence lesser ray coherence, setting tile sizes to the whole screen, such that inter-ray parallelism may be exploited, or tiling rays in a space that considers their origin as well as direction, might be a better alternative.

We also wish to point out that Piko synthesis misses a key optimization for ray tracing pipelines—implementing the recursive intersect-shade cycle within a single persistent kernel. Unlike modern high-performance ray tracers [36], the schedule synthesized using Piko does maps the cycle to a CPU loop, which is likely to be less efficient due to kernel launch overheads and load imbalance.

In the future, we would like to improve our branch scheduling heuristics from Section 5.4.1 to



(a) Ray tracing Pipeline (base)

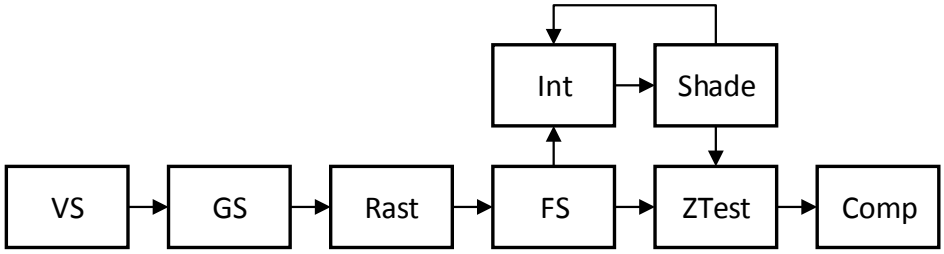


(b) Load-balanced ray tracing Pipeline

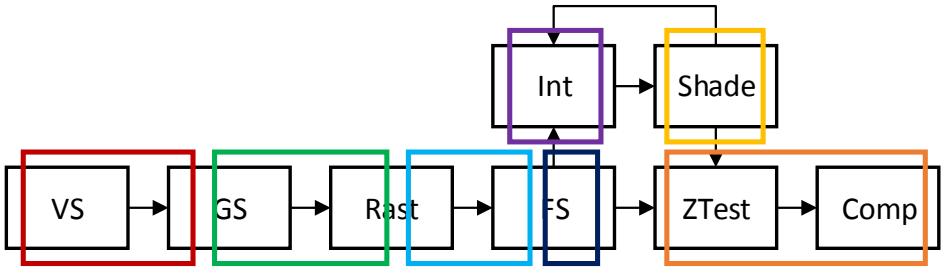
Figure A.8: *High-level organization of a ray tracing pipeline, with kernel mapping resulting by using the directives from Figure A.18.*

identify opportunities to place cycles like these within a single kernel.

A.5 Hybrid Rasterization-Ray-trace



(a) Hybrid Pipeline (base)



(b) Load-balanced Hybrid Pipeline

Figure A.9: *High-level organization of a hybrid-rasterization-ray-trace pipeline, showing a the kernel mapping as output during Piko synthesis. For details about the directives used, please see Figure A.19.*

Stage	Phase	Directive
Vertex Shade Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 1024
	Process	processOneToOne
Rasterize Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance tileSplitSize = 1024
	Process	processOneToMany
Fragment Shade Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToOne
Depth Test Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToOne
Composite Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance scheduleEndBin
	Process	Custom

Table A.2: *Piko directives for the strawman triangle rasterizer*

1: **Kernel** 1
2: VS::Schedule
3: VS::Process
4: Rast::AssignBin
5: Rast::Schedule
6: Rast::Process
7: FS::AssignBin
8: **End Kernel**
9: **Kernel** 2
10: FS::Schedule
11: FS::Process
12: ZTest::AssignBin
13: ZTest::Schedule
14: ZTest::Process
15: Comp::AssignBin
16: Comp::Schedule
17: Comp::Process
18: **End Kernel**

Algorithm 5: *Kernel plan for our strawman rasterizer*

Figure A.10: *Summary of directives (left) and synthesized kernel mapping (right) for the strawman version of a rasterizer pipeline.*

Finally, we demonstrate the synthesis of a hybrid-rasterizer-raytracer shown in Figure A.9. The fragment shader is capable of generating rays that help rendering complex effects like shadows, ambient occlusion, and secondary illumination. Figure A.9 also shows how we group these stages to respect the pipeline topology as well as programmer preferences. In Figure A.19 we show the directives supplied by the programmer, and the detailed kernel plan.

Again, we observe that the intersect-shade cycle is broken into two kernels. A hand-optimized implementation might have performed these steps within the same traversal kernel.

Stage	Phase	Directive
Vertex Shade Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 1024
	Process	processOneToOne
Geometry Shade 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleLoadBalance
	Process	processOneToOne Empty
Rasterize 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToMany
Fragment Shade 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToOne
Depth Test 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	processOneToOne
Composite 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	Custom

Table A.3: *Piko directives for a locality-optimized triangle rasterizer*

1: **Kernel** 1
2: VS::Schedule
3: VS::Process
4: GS::AssignBin
5: **End Kernel**
6: **Kernel** 2
7: GS::Schedule
8: Rast::AssignBin
9: Rast::Schedule
10: **End Kernel**
11: **Kernel** 3
12: Rast::Process
13: FS::AssignBin
14: FS::Schedule
15: FS::Process
16: ZTest::AssignBin
17: ZTest::Schedule
18: ZTest::Process
19: Comp::AssignBin
20: Comp::Schedule
21: Comp::Process
22: **End Kernel**

Algorithm 6: *Kernel plan for our locality-optimized rasterizer*

Figure A.11: *Summary of directives (left) and synthesized kernel mapping (right) for a locality-preserving version of a rasterizer.*

Stage	Phase	Directive
Vertex Shade Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 1024
	Process	processOneToOne
Geometry Shade 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleLoadBalance
	Process	processOneToOne Empty
Rasterize 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToMany
Fragment Shade 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance scheduleBatch(32)
	Process	processOneToOne
Depth Test 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	processOneToOne
Composite 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	Custom

Table A.4: *Piko directives for a load-balanced triangle rasterizer*

```
1: Kernel 1
2:     VS::Schedule
3:     VS::Process
4:     GS::AssignBin
5: End Kernel
6: Kernel 2
7:     GS::Schedule
8:     Rast::AssignBin
9:     Rast::Schedule
10: End Kernel
11: Kernel 3
12:     Rast::Process
13:     FS::AssignBin
14: End Kernel
15: Kernel 4
16:     FS::Schedule
17:     FS::Process
18:     ZTest::AssignBin
19:     ZTest::Schedule
20: End Kernel
21: Kernel 5
22:     ZTest::Process
23:     Comp::AssignBin
24:     Comp::Schedule
25:     Comp::Process
26: End Kernel
```

Algorithm 7: *Kernel plan for our load-balanced rasterizer*

Figure A.12: *Summary of directives (left) and synthesized kernel mapping (right) for a load-balanced version of a rasterizer.*

Stage	Phase	Directive
Vertex Shade 8×8 bins	AssignBin	Custom
	Schedule	scheduleAll tileSplitSize = 1024
	Process	processOneToOne
Rasterize 8×8 bins	AssignBin	Custom
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToMany
Depth Test 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	processOneToOne
Composite 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	processOneToOne
Deferred Shade 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	Custom

Table A.5: *Piko directives for a bucketed deferred triangle rasterizer*

1: **for all** VS Bins **do**
2: **Kernel** 1
3: VS::AssignBin
4: VS::Schedule
5: VS::Process
6: Rast::AssignBin
7: Rast::Schedule
8: **End Kernel**
9: **Kernel** 2
10: Rast::Process
11: ZTest::AssignBin
12: ZTest::Schedule
13: ZTest::Process
14: Comp::AssignBin
15: Comp::Schedule
16: Comp::Process
17: DS::AssignBin
18: DS::Schedule
19: DS::Process
20: **End Kernel**
21: **end for**

Algorithm 8: *Kernel plan for our bucketed deferred renderer*

Figure A.13: *Summary of directives (left) and synthesized kernel mapping (right) for a bucketing version of a deferred shading pipeline. Since we are modeling this pipeline after a real implementation, we omit the GS and FS stages accordingly.*

Stage	Phase	Directive
Vertex Shade Fullscreen bins	AssignBin	Empty
	Schedule	schedulescheduleAll tileSplitSize = 1024
	Process	processOneToOne
Geometry Shade 128×128 bins	AssignBin	Custom
	Schedule	scheduleLoadBalance
	Process	processOneToOne Empty
Rasterize 8×8 bins	AssignBin	Custom
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToMany
Fragment Shade 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance scheduleBatch(32)
	Process	processOneToOne
Depth Test 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	processOneToOne
Composite 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	processOneToOne
Deferred Shade 8×8 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	Custom

Table A.6: *Piko directives for a software load-balanced deferred rasterizer*

```
1: Kernel 1
2:     VS::Schedule
3:     VS::Process
4:     GS::AssignBin
5: End Kernel
6: Kernel 2
7:     GS::Schedule
8:     Rast::AssignBin
9:     Rast::Schedule
10: End Kernel
11: Kernel 3
12:     Rast::Process
13:     FS::AssignBin
14: End Kernel
15: Kernel 4
16:     FS::Schedule
17:     FS::Process
18:     ZTest::AssignBin
19:     ZTest::Schedule
20: End Kernel
21: Kernel 5
22:     ZTest::Process
23:     Comp::AssignBin
24:     Comp::Schedule
25:     Comp::Process
26:     DS::AssignBin
27:     DS::Schedule
28:     DS::Process
29: End Kernel
```

Algorithm 9: *Kernel plan for a load-balanced software deferred renderer*

Figure A.14: *Summary of directives (left) and synthesized kernel mapping (right) for a load-balanced version of a deferred shader. Note that we retain the GS and FS stages in this schedule, since it is a derivative of our original rasterization pipeline in Figure A.12.*

Stage	Phase	Directive
OGL Fullscreen bins	AssignBin	Empty
	Schedule	scheduleSerialize
	Process	Custom
Deferred Shade 8×8 bins	AssignBin	Custom
	Schedule	scheduleLoadBalance EndBin
	Process	Custom

Table A.7: *Piko directives for an OpenGL / Direct3D deferred triangle rasterizer*

```
1: OpenGL/ Direct3D Render Pass
2: Kernel 1
3:     DS::AssignBin
4:     DS::Schedule
5:     DS::Process
6: End Kernel
```

Algorithm 10: *Kernel plan for our OpenGL / Direct3D deferred renderer*

Figure A.15: *Summary of directives (left) and synthesized kernel mapping (right) for a deferred shading pipeline using OpenGL / Direct3D in the forward pass.*

Stage	Phase	Directive
Split Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 8
	Process	processOneToMany
Dice Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance tileSplitSize = 8
	Process	processOneToOne
Shade Fullscreen bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance tileSplitSize = 8
	Process	processOneToOne
Sample 128×128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance tileSplitSize = 8
	Process	processOneToMany
Composite 128×128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap tileSplitSize = 8 scheduleEndBin
	Process	Custom

Table A.8: *Piko directives for a load-balanced Reyes Renderer*

```
1: while Split not done do
2:     Kernel 1
3:         Split::Schedule
4:         Split::Process
5:     End Kernel
6: end while
7: Kernel 2
8:     Dice::AssignBin
9: End Kernel
10: Kernel 3
11:     Dice::Schedule
12:     Dice::Process
13:     Shade::AssignBin
14: End Kernel
15: Kernel 4
16:     Shade::Schedule
17:     Shade::Process
18:     Sample::AssignBin
19:     Sample::Schedule
20: End Kernel
21: Kernel 5
22:     Sample::Process
23:     Comp::AssignBin
24:     Comp::Schedule
25:     Comp::Process
26: End Kernel
```

Algorithm 11: *Kernel plan for a load-balanced Reyes pipeline*

Figure A.16: *Summary of directives (left) and synthesized kernel mapping (right) for a load-balanced Reyes pipeline.*

Stage	Phase	Directive
Split 128 × 128 bins	AssignBin	Custom
	Schedule	schedulescheduleAll tileSplitSize = 256
	Process	processOneToMany
Dice 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	schedulescheduleAll tileSplitSize = 256
	Process	processOneToOne
Shade 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	schedulescheduleAll tileSplitSize = 256
	Process	processOneToOne
Sample 32 × 32 bins	AssignBin	assignOneToOneIdentity
	Schedule	schedulescheduleAll tileSplitSize = 256
	Process	processOneToMany
Composite 32 × 32 bins	AssignBin	assignOneToOneIdentity
	Schedule	schedulescheduleAll scheduleEndBin
	Process	Custom

Table A.9: *Piko directives for a bucketed Reyes renderer*

1: **for all** buckets **do**
2: **Kernel** 1
3: Split::AssignBin
4: Split::Schedule
5: Split::Process
6: **End Kernel**
7: **Kernel** 2
8: Dice::AssignBin
9: **End Kernel**
10: **Kernel** 3
11: Dice::Schedule
12: Dice::Process
13: Shade::AssignBin
14: Shade::Schedule
15: Shade::Process
16: Sample::AssignBin
17: Sample::Schedule
18: **End Kernel**
19: **Kernel** 4
20: Sample::Process
21: Comp::AssignBin
22: Comp::Schedule
23: Comp::Process
24: **End Kernel**
25: **end for**

Algorithm 12: *Kernel plan for a bucketed Reyes renderer*

Figure A.17: *Summary of directives (left) and synthesized kernel mapping (right) for a bucketing Reyes pipeline.*

Stage	Phase	Directive
GenRays Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 4096
	Process	Custom
Intersect 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToOne
Shade 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToMany
Blend 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	Custom

Table A.10: *Piko directives for a load-balanced ray tracer*

```
1: Kernel 1
2:     GenRays::Schedule
3:     GenRays::Process
4: End Kernel
5: while numrays > 0 do
6:     Kernel 2
7:         Int::AssignBin
8:         Int::Schedule
9:         Int::Process
10:    End Kernel
11:    Kernel 3
12:        Shade::AssignBin
13:        Shade::Schedule
14:    End Kernel
15: end while
16: Kernel 4
17:    Comp::AssignBin
18:    Comp::Schedule
19:    Comp::Process
20: End Kernel
```

Algorithm 13: *Kernel plan for load-balanced ray tracer*

Figure A.18: *Summary of directives (left) and synthesized kernel mapping (right) for a load-balanced ray tracing pipeline.*

Stage	Phase	Directive
Vertex Shade Fullscreen bins	AssignBin	Empty
	Schedule	scheduleLoadBalance tileSplitSize = 1024
	Process	processOneToOne
Geometry Shade 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleLoadBalance
	Process	processOneToOne Empty
Rasterize 128 × 128 bins	AssignBin	Custom
	Schedule	scheduleDirectMap scheduleBatch(32)
	Process	processOneToMany
Fragment Shade 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance scheduleBatch(32)
	Process	processOneToOne
Intersect 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToOne
rtshade 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleLoadBalance
	Process	processOneToMany
Blend 16 × 16	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	Custom
Depth Test 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap
	Process	processOneToOne
Composite 128 × 128 bins	AssignBin	assignOneToOneIdentity
	Schedule	scheduleDirectMap EndBin
	Process	Custom

Table A.11: *Piko directives for a hybrid rasterizer-ray-tracer*

1: **Kernel** 1
2: VS::Schedule
3: VS::Process
4: GS::AssignBin
5: **End Kernel**
6: **Kernel** 2
7: GS::Schedule
8: Rast::AssignBin
9: Rast::Schedule
10: **End Kernel**
11: **Kernel** 3
12: Rast::Process
13: FS::AssignBin
14: **End Kernel**
15: **Kernel** 4
16: FS::Schedule
17: FS::Process
18: **End Kernel**
19: **while** numrays > 0 **do**
20: **Kernel** 5
21: Int::AssignBin
22: Int::Schedule
23: Int::Process
24: **End Kernel**
25: **Kernel** 6
26: Shade::Schedule
27: Shade::Process
28: **End Kernel**
29: **end while**
30: **Kernel** 7
31: ZTest::AssignBin
32: ZTest::Schedule
33: ZTest::Process
34: Comp::AssignBin
35: Comp::Schedule
36: Comp::Process
37: **End Kernel**

Algorithm 14: *Kernel plan for a hybrid rasterizer-ray-tracer*

Figure A.19: *Summary of directives (left) and synthesized kernel mapping (right) for a hybrid rasterizer-ray-trace pipeline.*

References

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics 2009*, pages 145–149, August 2009. doi: 10.1145/1572769.1572792.

[2] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 22(3):792–800, July 2003. doi: 10.1145/882262.882347.

[3] Kurt Akeley. RealityEngine graphics. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 109–116, August 1993. ISBN 1-55860-539-8. doi: 10.1145/166117.166131.

[4] Johan Andersson. Parallel graphics in Frostbite—current & future. In *Beyond Programmable Shading (ACM SIGGRAPH 2009 Course)*, pages 7:1–7:312, 2009. doi: 10.1145/1667239.1667246.

[5] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999. ISBN 1558606181.

[6] astrofra (Wikipedia user). Deferred shading. [Online, available under Creative Commons Attribution-Share Alike 3.0 Unported, 2.5 Generic, 2.0 Generic and 1.0 Generic license; accessed 17-November-2013], 2013. URL http://en.wikipedia.org/wiki/Deferred_shading.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, February 2011. ISSN 1532-0626. doi: 10.1002/cpe.1631.

[8] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie 'Cars'. In *IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006. doi: 10.1109/RT.2006.280208.

[9] Andrew Clinton and Mark Elendt. Rendering volumes with microvoxels. In *Proceed-*

ings of ACM SIGGRAPH: Talks, pages 47:1–47:1, 2009. doi: 10.1145/1597990.1598037.

[10] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987. doi: 10.1145/37402.37414.

[11] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. URL http://maverick.inria.fr/Publications/2011/Cra11.

[12] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A fully scalable graphics architecture. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 443–454, July 2000. doi: 10.1145/344779.344981.

[13] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. PixelFlow: The realization. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997. doi: 10.1145/258694.258714.

[14] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Communications of the ACM*, 51(10):50–57, October 2008. ISSN 0001-0782. doi: 10.1145/1400181.1400197.

[15] Tim Foley and Pat Hanrahan. Spark: modular, composable shaders for graphics hardware. *ACM Transactions on Graphics*, 30(4):107:1–107:12, July 2011. doi: 10.1145/2010324.1965002.

[16] Blender Foundation. Big buck bunny. [Online, available under under Creative Commons Attribution 3.0 License; accessed 17-November-2013], 2013. URL http://www.bigbuckbunny.org/.

[17] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 79–88, July 1989. doi: 10.1145/74333.74341.

[18] Google. Nexus 10 tech specs, 2013. URL http://www.google.com/nexus/10/specs. [Online; accessed 3-November-2013].

[19] Google. Nexus 7 tech specs, 2013. URL http://www.google.com/nexus/7/specs. [Online; accessed 3-November-2013].

[20] Larry Gritz. Compiler technology in Open Shading Language. Talk at SIGGRAPH 2011, August 2011. URL https://sites.google.com/site/s2011compilers.

[21] Kshitij Gupta, Jeff Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012. doi: 10.1109/InPar.2012.6339596. URL http://www.idav.ucdavis.edu/publications/print_pub?pub_id=1089.

[22] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 108–120, 1997. doi: 10.1145/264107.264152.

[23] Jon Hasselgren and Thomas Akenine-Möller. PCU: The programmable culling unit. *ACM Transactions on Graphics*, 26(3):92:1–92:10, July 2007. doi: 10.1145/1275808.1276492.

[24] Christopher Horvath and Willi Geiger. Directable, high-resolution simulation of fire on the GPU. *ACM Transactions on Graphics*, 28(3):41:1–41:8, July 2009. doi: 10.1145/1531326.1531347.

[25] Imagination Technologies Ltd. *POWERVR Series5 Graphics SGX architecture guide for developers*, 5 July 2011. Version 1.0.8.

[26] Samuli Laine and Tero Karras. High-performance software rasterization on GPUs. In *Proceedings of High Performance Graphics 2011*, pages 79–88, August 2011. doi: 10.1145/2018323.2018337.

[27] Douglas Lanman and David Luebke. Near-eye light field displays. In *ACM SIGGRAPH 2013 Talks*, pages 10:1–10:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2344-4. doi: 10.1145/2504459.2504472.

[28] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code*

Generation and Optimization (CGO'04), pages 75–86, Palo Alto, California, Mar 2004. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665.

[29] Changmin Lee, Won W. Ro, and Jean-Luc Gaudiot. Cooperative heterogeneous computing for parallel processing on CPU/GPU hybrids. *16th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 33–40, 2012. doi: 10.1109/INTERACT.2012.6339624.

[30] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March/April 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31.

[31] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 75–82, February 2010. ISBN 978-1-60558-939-8. doi: 10.1145/1730804.1730817.

[32] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669121.

[33] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003. doi: 10.1145/882262.882362.

[34] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Conference on High Performance Graphics*, August 2009. doi: 10.1145/1572769.1572783.

[35] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 293–302, August 1997. doi:

10.1145/258734.258871.

[36] NVIDIA. NVIDIA®OptiX application acceleration engine. http://www.nvidia.com/object/optix.html, 2009.

[37] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.nvidia.com/cuda, January 2013.

[38] Jacopo Pantaleoni. VoxelPipe: A programmable pipeline for 3D voxelization. In *High Performance Graphics*, pages 99–106, August 2011. doi: 10.1145/2018323.2018339.

[39] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010. doi: 10.1145/1778765.1778803.

[40] Jacques A. Pienaar, Srimat Chakradhar, and Anand Raghunathan. Automatic generation of software pipelines for heterogeneous parallel systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 24:1–24:12, 2012. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389029.

[41] Michael Potmesil and Eric M. Hoffert. The Pixel Machine: A parallel image computer. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 69–78, July 1989. doi: 10.1145/74333.74340.

[42] Tim Purcell. Fast tessellated rendering on Fermi GF100. In *High Performance Graphics Hot3D*, June 2010.

[43] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Fredo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, July 2012. doi: 10.1145/2185520.2185528.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, June 2013. doi: 10.1145/2462156.2462176.

[45] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, 2003. ISBN 1-58113-634-X. doi: 10.1145/872757.872770.

[46] Sharp. PN-K321, 32-inch class professional 4K ultra HD monitor, 2013. URL http://www.sharpusa.com/ForBusiness/PresentationProducts/ProfessionalLCDMonitors/PNK321.aspx. [Online; accessed 3-November-2013].

[47] Stanford Computer Graphics Laboratory. The Stanford 3D scanning repository, 2013. URL http://graphics.stanford.edu/data/3Dscanrep/. [Online; accessed 23-November-2013].

[48] Ashley Stevens. ARM Mali 3D graphics system solution. http://www.arm.com/miscPDFs/16514.pdf, 2006.

[49] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28(1):4:1–4:11, January 2009. doi: 10.1145/1477926.1477930.

[50] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. doi: 10.1145/63526.63532.

[51] Ingo Wald. The Utah 3D animation repository, 2013. URL http://www.sci.utah.edu/~wald/animrep/. [Online; accessed 23-November-2013].

[52] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics*, 28(5): 155:1–155:11, December 2009. doi: 10.1145/1661412.1618501.