Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

A Reconfigurable Architecture for Load-Balanced Rendering

Jiawen Chen[1] Michael I. Gordon[1] William Thies[1] Matthias Zwicker[1] Kari Pulli[2,1] Frédo Durand[1]

[1]Massachusetts Institute of Technology [2]Nokia Research Center

Abstract

Commodity graphics hardware has become increasingly programmable over the last few years but has been limited to fixed resource allocation. These architectures handle some workloads well, others poorly; load-balancing to maximize graphics hardware performance has become a critical issue. In this paper, we explore one solution to this problem using compile-time resource allocation. For our experiments, we implement a graphics pipeline on Raw, a tile-based multicore processor. We express both the full graphics pipeline and the shaders using StreamIt, a high-level language based on the stream programming model. The programmer specifies the number of tiles per pipeline stage, and the StreamIt compiler maps the computation to the Raw architecture.
We evaluate our reconfigurable architecture using a mix of common rendering tasks with different workloads and improve throughput by 55–157% *over a static allocation. Although our early prototype cannot compete in performance against commercial state-of-the-art graphics processors, we believe that this paper describes an important first step in addressing the load-balancing challenge.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture - Graphics processors C.1.2 [Processor Architectures]: Multiple Data Stream Architectures - Single-instruction-stream, multiple-data-stream processors (SIMD)

1. Introduction

"All processors aspire to be general-purpose."

– Tim Van Hook, Graphics Hardware 2001

And so it has been with commodity graphics processing units (GPUs) in the last few years. New features, such as floating-point per-pixel operations and flow control, offer new and exciting possibilities for shading as well as for general-purpose non-graphics applications.

Despite significant gains in performance and programmable features, current GPU architectures have a key limitation: a fixed resource allocation. For example, the NVIDIA NV40 processor has 6 vertex pipelines, 16 fragment pipelines, and a fixed set of other resources surrounding these programmable stages. The allocation is fixed at design time and remains the same for all software applications that run on this chip.

Although GPU resource allocations are optimized for common workloads, it is difficult for a fixed allocation to work well on all possible scenarios. For instance, during an expensive image-based special-effects rendering pass, the vertex engines sit idle. Conversely, when the bottleneck lies in the vertex shader because of complex deformations, the pixel engines are idle. And when an application spends much of its time rasterizing shadow volumes (e.g., in games such as *Doom 3*), almost the entire chip is idle. These are common scenarios that suffer from load imbalance due to a fixed resource allocation.

In this paper, we examine one approach for solving this load-imbalance problem using compile-time resource allocation. For our experiments, we have implemented a graphics pipeline on Raw, a parallel tiled processor. We compare a fixed resource allocation representative of current graphics architecture against compile-time flexible resource allocations on Raw. The flexible allocation takes into account the complexity of the calculation for a given rendering pass and the respective load on the various stages of the pipeline. We take an extreme stance and explore the hypothesis where all stages of the graphics pipeline are programmable. While many will argue that GPUs will retain a level of specialization in the foreseeable future (in particular for rasterization),

we show that our hypothesis allows us to explore a point in the design space that permits efficient load balancing. We hope that this work will inspire graphics hardware designs that are reconfigurable and yield better resource utilization through load-balancing.

1.1. Overview

The prototype implementation of our approach is made feasible by two unique technologies: a multicore processor with programmable communication networks and a programming language that allows us to specify the topology of the graphics pipeline using high-level language constructs. Our rendering pipeline is executed on the Raw processor [TKM*02], which is a highly scalable architecture with programmable communication networks. The programmable networks allow us to realize pipelines with different topologies; hence, we can allocate computation units to rendering tasks based on the demands of the application. We program Raw using StreamIt [GTK*02], which is a high-level language based on a stream abstraction. The stream programming model of StreamIt facilitates the expression of parallelism with high-level language constructs. The StreamIt compiler generates code to control Raw's networks and relieves the programmer of the burden of manually managing data routing between processor tiles. On the compiler side, our main challenge has been to extend StreamIt to handle the variable data rates present in 3D rendering due to the variable number of pixel outputs per triangle and shader lengths.

We should emphasize that our implementation is meant as a proof of concept. Beyond the implementation of load-balancing for 3D rendering in this particular environment, the thesis of this article is that load-balancing and increased programmability can be achieved through the following approach:

A multicore chip with exposed communication enables general-purpose computation and resource reallocation by rerouting data flow.

A stream-based programming model facilitates the expression of arbitrary computation.

A compiler approach to static load-balancing facilitates the appropriate mapping of computing units for each application phase. The programmer specifies the number of computing units allocated to each stage of the pipeline.

We focus on load balancing and the programming model at the cost of the following points, which we plan to address in future work. First, we do not emphasize the memory system, although we acknowledge its crucial influence on graphics hardware performance. Second, we focus on the high-level architecture of the chip and its resource allocation and do not address the design of individual processing elements (in particular, vector computation capabilities would likely improve our current performance). Third, we only explore load balancing at compilation time, where different rendering passes can have different static configurations. We leave dynamic load balancing, where the configuration changes within a rendering pass, as an exciting topic for future research. Finally, we push full programmability quite far and do not use specialized units for rasterization, the stage of the 3D pipeline that seems least likely to become programmable in the near future for performance reasons. Specializing triangle rasterizers to support other rendering primitives (e.g., point sprites) is a promising part of our ongoing research agenda.

Despite these limitations, and although the performance obtained by our simulation cannot compete with state-of-the-art graphics cards, we believe that this paper describes an important first step in addressing the load-imbalance problem in current graphics architectures. Solving this problem is important because doing so maximizes the use of available GPU resources, which in turn implies a more efficient and cost-effective rendering architecture. We hope to encourage the design of reconfigurable graphics architectures in the future.

Paper organization. The rest of this paper is organized as follows: After reviewing related work in Section 1.2, we provide an overview of the Raw processor and the StreamIt language in Section 2. We then discuss mapping a rendering pipeline to such a framework (Section 3), followed by a series of case studies illustrating the improvements in processor utilization that we achieve through flexible resource allocation (Section 4).

1.2. Related Work

Parallelism is a key source of the immense computation power of graphics processors. Recent studies have focused on various aspects such as parallel interfaces [ISH98] or load distribution and scalability with parallel pipelines [EIH00]. Both general-purpose processors [NK96] and PC clusters [HHN*02] have been used for parallel rendering. However, these systems do not apply compile-time load balancing for each application or rendering pass. Hence, they may suffer from the same load imbalances as special purpose hardware. In contrast, our system focuses on flexible resource allocation at a fine level of granularity.

Pioneering work in programmable and reconfigurable architectures include PixelFlow [EMP*97] and the Pixel Machine [PH89]. PixelFlow had programmable vertex and fragment processing and the ability to balance the load between the two on similar processing elements. The Pixel Machine featured a programmable pipeline implementation and the ability to map these algorithms onto its compute nodes. These designs were workstation architectures and required a large amount of hardware. With the increasing transistor budget provided by modern manufacturing processes, it has become more viable to add a certain level of programmability to various functional units in consumer GPUs,

e.g., the vertex unit [LKM01]. Also, a number of high-level languages and compilers are now available to program these units [PMTH01, MGAK03, MQP02]. Hence, computer graphics hardware has become more and more attractive for general purpose, high-performance computing. In particular, the architecture of rendering pipelines closely matches the concept of stream processing. For example, Buck et al. [BFH*04] presents a streaming language that permits the implementation of streaming algorithms on a graphics processor. We focus on expressing the rendering pipeline and mapping it to a general-purpose processor.

Owens et al. [ODK*00] characterizes rendering as a stream operation and demonstrates the implementation of a fixed-function pipeline on a data-parallel stream processor. We use a tiled stream processor, the Raw machine, and we make the pipeline completely programmable. The application developer can compile several pipelines, and change the graph topology at runtime, allowing for compile-time load balancing.

Our approach is also related to the shader algebra [MTP*04] where shaders can be combined and code analysis leads to efficient compilation and dead-code elimination.

In this paper, we start from the assumption that future GPUs will contain a single type of general purpose processing tile that can be assigned flexibly to different tasks (the upcoming unified shaders are going in this direction [Bly05]). Our goal is to study the implications and challenges that this scenario imposes on the "driver" of such a processor. The driver will be of critical importance because it allocates resources depending on the rendering task and ensures that the processor is used efficiently. We build on solutions in the stream compiler community to tackle this challenge.

2. Background

In this section, we give an overview of the two main technologies that our system is built upon: the Raw processor (Section 2.1) and the StreamIt language and compiler (Section 2.2).

2.1. The Raw Processor

The Raw processor [TKM*02, TLM*04] is a versatile architecture that achieves scalability by addressing the wire delay problem. Raw also aims to be as efficient as an ASIC while still running general purpose programs with reasonable performance. Raw approaches these challenges by exposing its rich on-chip resources, which include logic, wires, and pins, through a new ISA to the software. In contrast to other architectures, this allows Raw to more effectively exploit all forms of parallelism, including instruction, data, and thread level parallelism, as well as pipeline parallelism.

Tile-Based Processor Architecture. Raw is a parallel processor with a 2-D array of identical, programmable tiles. Each tile contains a *compute processor* as well as a *switch processor* that manages four networks to neighboring tiles. The compute processor is composed of an eight-stage in-order single-issue MIPS-style processor, a four-stage pipelined floating point unit, a 32kB data cache, and a 32kB instruction cache. The current prototype is implemented in an IBM 180nm ASIC process running at 425MHz; on one chip, it contains 16 uniform tiles arranged in a square grid. The theoretical peak performance of this prototype is 6.8 GFLOPS. In this paper, we gather results using *btl*, a cycle-accurate simulator that can model multiple tile configurations. We use a 64-tile configuration for our results. Though the prototype chip contains only 16 tiles, a 64-tile fabric is under construction.

On-Chip Communication Networks. The switch processors control four 32-bit full-duplex on-chip networks. The networks are register-mapped, blocking, and flow-controlled, and they are integrated directly into the bypass paths of the processor pipeline. As a key innovative feature of Raw, these networks are exposed to the software through the Raw ISA.

There are two *static* networks and two *dynamic* networks. The static networks are used for communication patterns that are known at compile time. To route a word from one tile to another over a static network, it is the responsibility of the compiler to insert a route instruction on every intermediate switch processor. The static networks are ideal for regular stream-based traffic and can also be used to exploit instruction level parallelism [TLAA03]. The dynamic networks support patterns of communication that vary at runtime. Items are transmitted in packets; a header encodes the destination tile and packet length. Routing is done dynamically by the hardware, rather than statically by the compiler. There are two dynamic networks: a *memory* network for trusted clients (data caches, I/O, etc.) and a *general* network for use by applications.

Memory System. On the boundaries of the chip, the network channels are multiplexed onto the pins to form flexible I/O ports. Words routed off the side of the chip emerge on the pins, and words put on the pins by external devices appear on the networks. Raw's memory system is built by connecting these ports to external DRAMs. For the 16 tile configuration, Raw supports a maximum of 14 ports, which can be connected to as many as 14 full-duplex DRAM memory banks, leading to a memory bandwidth of 47.6GB per second. While there are I/O ports only on the boundary of the chip, any tile can access memory by routing requests and data over the networks. Memory accesses from inner tiles are almost as efficient as boundary tiles because the on-chip network latency (1 cycle / hop) is negligible compared to the off-chip memory latency.

Raw as Graphics Hardware. We believe the Raw architecture is interesting for graphics hardware developers, because its design goals share a number of similarities with current GPUs. Raw is tailored to effectively execute a wide variety of computations, from special purpose computations that are often implemented using ASICs to conventional sequential programs. GPUs exploit data parallelism (by replicating rendering pipelines, using vector units), instruction level parallelism (in super-scalar fragment processors), and pipeline parallelism (by executing all stages of the pipeline simultaneously). Raw, too, is capable of exploiting these three forms of parallelism. In addition, Raw is scalable: it consists of uniform tiles with no centralized resources, no global buses, and no structures that get larger as the tile count increases. In contrast to GPUs, Raw's computational units and communication channels are fully programmable, which opens up almost unlimited flexibility in laying out a graphics pipeline and optimizing its efficiency.

On the other hand, the computational power of the current 16 tile, prototype Raw processor is more than an order of magnitude smaller than the power of current GPUs. The obvious reason is that the number of parallel operations on Raw is much smaller than on GPUs (Raw's computation units do not perform vector computation). In addition, Raw is a research prototype implemented with a 180nm process; an industrial design with a modern 90nm process would achieve higher clock frequencies.

Hence, in this paper we do not intend to compete with current GPUs in terms of absolute performance, but we show the benefits of a flexible and scalable architecture for efficient resource utilization. The optimization of the Raw architecture for graphics pipelines is an exciting direction for future research.

2.2. The StreamIt Programming Language

StreamIt [TKA02, GTK*02] is a high-level stream language that aims to be portable across communication-exposed architectures such as Raw. The language exposes the parallelism and communication of streaming programs without depending on the topology or granularity of the underlying architecture. The StreamIt programming model is based on a *structured stream abstraction*: all stream graphs are built out of a hierarchical composition of filters, pipelines, split-joins, and feedback-loops (described below).

As we will describe in more detail in Section 3, the structured stream graph abstraction provided by StreamIt lends itself to expressing data parallelism and pipeline parallelism that appear in graphics pipelines. In particular, we will show how to use StreamIt for high-level specification of rendering pipelines with different topologies. As previously published, StreamIt permits only fixed data rates. In order to implement a graphics system that allows different triangle sizes, support for variable data rates is necessary. In this work, we add variable data rates to the StreamIt language and compiler.

Language Constructs. The basic unit of computation in StreamIt is the *filter*. A filter is a single-input, single-output block with a user-defined procedure for translating input items to output items. Filters send and receive data to and from other filters through FIFO queues with compiler-checked data types. StreamIt distinguishes between filters with *static* and *variable* data rates. A static data rate filter reads a fixed number of input items and writes a fixed number of output items each time it fires, whereas a variable data rate filter may read or write a varying number of items.

In addition to the filter, StreamIt provides three language constructs to compose *stream graphs*: pipeline, split-join, and feedback-loop. We call each of these constructs, including a filter, a *stream*. In a pipeline, streams are connected in a linear chain so that the outputs of one stream are the inputs to the next stream. In a split-join configuration, the output from a stream is split and sent to multiple (not necessarily identical) streams that have the same input data type. The data can be either duplicated or placed in a weighted round-robin scheduling policy. The parallel streams must either be joined somewhere downstream or the split-join must serve as the sink for the entire stream graph. The programmer can specify data-parallelism by varying the width of the split-join. The feedback-loop enables a stream to receive input from downstream, for applications such as MPEG.

2.3. Compiling StreamIt to Raw

A compiler for mapping static data rate StreamIt to Raw has been described in previous work [GTK*02]. Compilation involves four stages: dividing the stream graph into load-balanced partitions, laying out the partitions on the chip, scheduling communication between the partitions, and generating code. In this paper, automatic load-balancing is disabled: as we assume that the programmer has domain-specific knowledge about the work requirements, each filter in the program is directly mapped to a single tile. We summarize the operation of the layout and communication scheduling stages below and describe how to extend them to variable data rates in Section 3.3.

Layout The layout stage assigns each filter in the stream graph to a Raw tile while minimizing communication and synchronization. Since an exhaustive search for the optimal layout is computationally intractable, we use a simulated annealing algorithm that incrementally adjusts the layout to optimize a cost function. The cost function measures the memory latency and communication overhead for a given layout, as well as the synchronization imposed when independent communication channels are mapped to intersecting routes on the chip. For example, placing memory intensive filters closer to the boundary will reduce the memory latency component of the cost function. If there are fewer filters than tiles, certain tiles will remain unmapped; unmapped tiles will be utilized for routing. For all layouts used in our ex-

periments, the simulated annealing algorithm ran in under 5 minutes on a Pentium Xeon 2.8 GHz machine.

Communication Scheduling Communication scheduling maps the abstract communication channels of the stream graph to Raw's static network, maximizing throughput while avoiding deadlock. As multiple logical channels from the stream graph might be multiplexed over a single physical network link, routing operations are sequenced to minimize the amount of time that a given tile is idle waiting for another pair of tiles to communicate. This static communication schedule is calculated by simulating the firing of filters in the stream graph and recording the communication pattern for each switch processor.

3. Flexible Graphics Pipelines Using StreamIt

To address the load-balancing problem, we designed a *flexible* graphics pipeline using general purpose hardware, where the allocation of resources to tasks can be changed to adapt to the input. We first describe how we express the graphics pipeline using StreamIt, before presenting our extension of the compiler to enable variable data rates.

The StreamIt philosophy is to implement filters as interchangeable components. In the flexible pipeline, the different stages are implemented as StreamIt filters and allocated to Raw by the StreamIt compiler. The programmer is free to vary the pipeline topology by rearranging the filters and recompiling. The flexible pipeline has several advantages over a fixed pipeline on the GPU. First, any filter (i.e., stage) in the pipeline can be changed. For example, in the first two passes of shadow volume rendering, texture mapping is not used, and we can perform dead-code elimination. The entire pipeline is changed so that texture coordinates are neither interpolated nor part of the dataflow, and the pixel shader stage is removed. For the third pass, these functions are restored. Second, the topology does not even need to conform to any traditional pipeline configuration. In our image processing case study (Section 4.3), the current GPU method would render the scene to a texture and use a complex pixel shader to perform image filtering. We simply reconfigure Raw to act as a extremely parallel image processor.

In our case studies (Section 4) we compare the performance of a flexible pipeline against a fixed-allocation reference pipeline. The reference pipeline models the same design tradeoff as made in GPUs in fixing the ratio of fragment to vertex units. We demonstrate that a flexible pipeline better balances the load across the available resources and achieves up to a 157% increase in throughput.

3.1. Pipeline Implementation

The reference pipeline is implemented using StreamIt and emulates most of the functionality of a programmable GPU. It is manually laid out on Raw (Figure 1). The pipeline stages

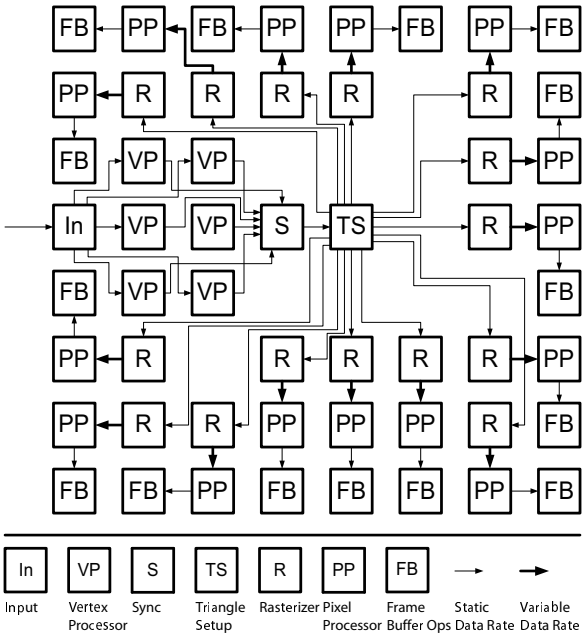

Figure 1: *Reference pipeline layout on an* 8×8 *Raw configuration. Squares represent Raw tiles. Data arrives from an I/O port off the edge of the chip, and data is written to off-chip memory by tiles assigned to frame buffer operations. Unallocated tiles have been removed to clarify the routing.*

include Input, Programmable Vertex Processing, Triangle Setup, Rasterization, Programmable Pixel Shading (including texture mapping), and Reconfigurable Raster Operations that write to the frame buffer. Some tiles are empty to improve routing of data items.

The pipeline is a sort-middle architecture [MCEF94]. The input stage is connected to off-chip memory through an I/O port. Six tiles are assigned to programmable vertex processing, and they are synchronized through one synchronization tile. The synchronizer consumes output of the vertex shaders using a round-robin strategy and pushes the data to the triangle setup tile. We use homogeneous rasterization to avoid the overhead of clipping operations [OG97]. The triangle setup stage computes the vertex matrix and its inverse, the screenspace bounding box, triangle facing, and the parameter vectors needed for interpolation. It distributes data to the 15 pixel pipelines. The pixel pipelines are screen locked and interleaved. Each pipeline is assigned to every 15th column. The pixel pipelines each consist of three tiles, a rasterizer that outputs the visible fragments of the triangle, a programmable pixel processor, and a frame buffer operations tile, which communicates with off-chip memory through an I/O port to perform Z-buffering, blending and stencil buffer operations. In StreamIt, filters are independent and have independent address spaces. Hence, for efficient random access to textures, texture memory is replicated to all pixel

processors. In contrast, the frame buffer is not replicated; it is distributed across multiple memory banks. Since the pixel pipelines are screen locked, frame buffer accesses are independent.

We implement data-level parallelism using StreamIt's split-joins at the vertex and pixel level. Our reference pipeline corresponds to the design strategy of current graphics hardware: at design time, a hypothesis is made on the relative usage of the various stages of the pipeline, and a fixed resource allocation is decided to optimize for the situation. As mentioned above, split-joins are used to provide 6 vertex and 15 pixel pipelines.

In contrast, our reconfigurable pipeline builds on the same filters as the reference pipeline, but the programmer varies the topology depending on the rendering pass of the application. It leverages the StreamIt compiler to automatically lay out the stream graph on Raw. The main parameters are the width of the split-joins at the vertex and pixel stages. Where the computation permits, the depth of the pipeline for a certain stage (such as triangle setup or pixel shading) can also be changed. In some cases, the programmer can also omit some of the filters when they are not needed. Together, flexible resource allocation and dead-code elimination greatly improve performance.

While the automatic assignment of filters to tiles by the compiler provides great flexibility, the layout is not necessarily optimal. Automatic layout can act as a good first approximation so the programmer can iterate on pipeline configurations without having to manually configure the tiles, after which the programmer is free to tweak the layout. In our benchmarks, we use the automatic layouts without modification.

3.2. Switching Between Configurations

We only consider *static load balancing*, where it is assumed that the programmer has *a priori* knowledge of the upcoming frame, rendering pass, or even part of a frame. This scenario is realistic for a wide range of applications where the programmer has already profiled the application. Static load balancing is achieved via user-specified "context switches" between pre-compiled stream graphs when the load is expected to change. Switching between configurations involves flushing the pipeline and having each tile jump to the code for its new task. These context switches may or may not incur some cost. Consider the case of a multi-pass rendering algorithm such as shadow volume rendering. Each pass places the load on a different part of the pipeline and we would like to switch configurations between them. In this case, a pipeline flush occurs between passes anyway, so the overhead of the branch (and possible instruction cache miss) is negligible. The other case is a configuration switch *within* a frame. An example of this would be rendering a scene with a detailed character (vertex limited) over a background composed of large triangles (fragment limited). Normally, a flush would not occur; the programmer must profile the application and decide if the overhead from the flush is greater than the performance increase of the new configuration.

3.3. Variable Data Rates in StreamIt

Variable data rates are essential for graphics rendering. Because the number of pixels corresponding to a given triangle depends on the positions of the vertices for that triangle, the input/output ratio of a rasterizer filter cannot be fixed at compile time. This contrasts with traditional applications of stream-based programming such as digital signal processing that exhibit a fixed ratio of output to input and can be implemented using synchronous dataflow models. In particular, the original version of StreamIt relies on the static data rate assumption.

We augmented the StreamIt language and compiler to support variable data rates between filters. The language extension is simple, allowing the programmer to specify a data rate as variable. On the compiler side, variable data rates are supported by dividing the stream graph into *static-rate subgraphs*. Each subgraph represents a stream in which child filters have static data rates for internal communication. A variable data rate can appear only between subgraphs. The phases of the StreamIt to Raw mapping are described below.

Partitioning with Variable Data Rates In this paper, we rely on manual partitioning. Because the programmer might have application-specific knowledge about the relative load between different subgraphs, she should write the application to have the appropriate number of filters in each stage. The compiler maps each filter to a single tile. The programmer can easily adjust the number of filters allocated to a given task using StreamIt's parameterized split-joins.

Layout with Variable Data Rates Variable data rates impose two new layout constraints. First, a switch processor must not interleave routing operations for distinct static subgraphs. Because the relative execution rates of subgraphs are unknown at compile time, it is impossible to generate a static schedule that interleaves operations from two subgraphs without risking deadlock. Second, there is a constraint on the links between subgraphs: variable-rate communication channels that are running in parallel and have downstream synchronization must not cross on the chip. Even when such channels are mapped to the dynamic network, deadlock can result if such channels share a junction, since a high-traffic channel can block another. However, this constraint is only needed in the general case; our benchmarks do not contain synchronization between variable-rate streams. In our implementation, these constraints are incorporated into the cost function in the form of large penalties for illegal layouts.

Communication Scheduling with Variable Data Rates Communication scheduling requires a simple extension:

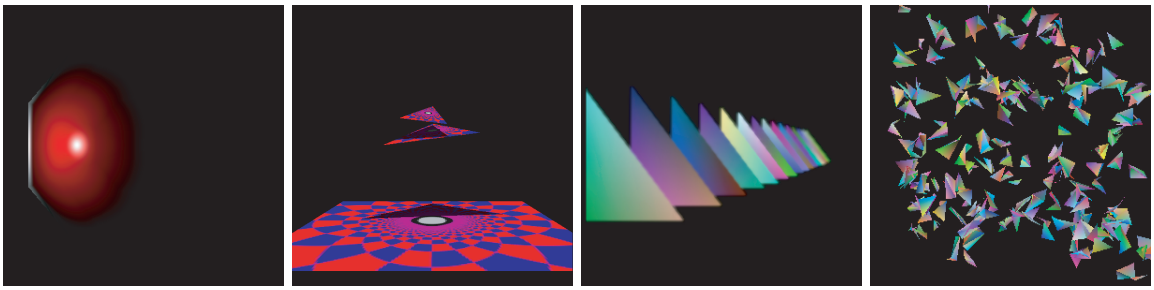

Figure 2: *Output Images. Case studies 1 through 4, left to right. Original Resolution:* 600×600.

channels with variable data rates are mapped to Raw's general dynamic network (rather than the static network, which requires a fixed communication pattern). Within each subgraph, the static network is still used. Our implementation avoids the cost of constructing the dynamic network header for every packet; instead, we construct the header once at initialization time. Even though the rate of communication is variable, the endpoints of each communication channel are determined at compile time.

4. Case Studies

We study a number of rendering scenarios to demonstrate the load imbalance present on a fixed hardware allocation. We also show how the imbalance can be alleviated and performance improved by reallocating resources appropriately.

In the following case studies, we list performance numbers in terms of triangles per second and percent utilization. The screen resolution is fixed at 600×600 pixels. See Figure 2 for output images. Pipeline stage utilization is computed as the number of instructions completed by all tiles assigned to that stage divided by the number of instruction slots for all the tiles of the stage. Note that this metric for processor utilization is unlikely to reach 100% in any scenario, even in highly parallel computations such as image filtering (Section 4.3). While each tile is fully pipelined, it is unlikely to achieve 1 instruction per clock cycle. Floating point operations incur a 4-cycle latency, local memory accesses have a 3-cycle latency, and there are likely to be data hazards in the computation. Also, Raw's in-order, single-issue compute processor uses static branch prediction with a mis-prediction penalty of 3 cycles.

4.1. Case Study 1: Phong Shading

Consider the case of rendering a coarsely tessellated polyhedron composed of large triangles with per pixel Phong shading. In the vertex shader, the vertex's world space position and normal are bound as texture coordinates. The rasterizer interpolates the texture coordinates across each triangle and the pixel shader computes the lighting direction and the diffuse and specular contributions. Most of the load is expected to be on the fragment processor.

Case Study	Instructions Removed	Throughput Improvement
Case 1: Phong Shading	-1%	55%
Case 2: Shadow Volumes	46%	62%
pass 1	62%	126%
pass 2	63%	126%
pass 3	24%	13%
Case 4: Particle System	13%	157%

Table 1: *Fraction of dynamic instructions removed and throughput improvement when using a flexible resource allocation rather than a fixed resource allocation.*

Reference Pipeline As expected, the reference pipeline suffers from an extreme load imbalance. The fragment processor is at 68% utilization, the rasterizer is at 17%, while the other units are virtually idle ($< 1\%$) (Figure 5). Throughput is 4060 triangles per second.

Flexible Pipeline We tried several different allocations for this scenario. We varied the ratio between vertex and fragment processors as well as the depth of the pixel pipelines. We discovered that the largest gain in performance came when we pipelined the pixel shader onto two tiles; the layout is shown in Figure 3.

In this allocation, the first pixel processor is at 74% utilization, and the second at 60%. The rasterization stage's utilization increases to 31%. The load-balance has improved significantly. This allocation achieves a throughput of 6280 triangles per second, a 55% increase over the fixed allocation (Table 1).

4.2. Case Study 2: Multi-Pass Rendering—Shadow Volumes

To demonstrate the utility of a flexible pipeline, we benchmarked shadow volume rendering, a popular technique for generating real-time hard shadows. In this algorithm, the load shifts significantly over the three passes. In the first pass, the depth buffer is initialized with the depth values of the scene geometry. In our scene, the triangles are relatively large and the computation is rasterization bound. In the second pass, the shadow volume itself is rendered. This incurs a significant load on the rasterizer which has to rasterize large

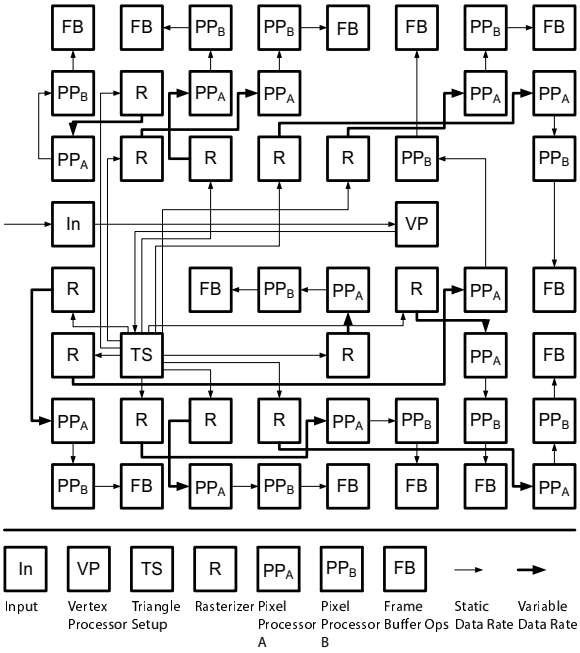

Figure 3: *Compiler generated allocation for case study #1. It has 1 vertex processor and 12 pixel pipelines, with 2 fragment processors per pixel pipeline.*

shadow volume polygons, and the frame buffer operations, which must perform a depth test and update the stencil buffer. In the final pass, the fragment shader is used to texture map the geometry and the computation is fragment-limited.

Reference Pipeline In the first pass, as expected, the rasterization stage is the bottleneck at 69% utilization. It takes approximately 55 floating point operations for the software rasterizer to output each fragment. The pixels are output in screen-aligned order and memory access is very regular for the large triangles. The frame buffer updates only achieve a 7% utilization. The other units in the pipeline are virtually idle, with the exception of the pixel shader use at 6%; it simply forwards rasterized fragments to the frame buffer operations unit. Throughput is 933 triangles / second.

On the second pass, the results are virtually identical, with a slight increase in utilization at the frame buffer operations stage where the Z-Fail algorithm updates the stencil buffer based on triangle orientation. Throughput is 752 triangles / second.

In the final pass, the pixel processor retrieves four samples from memory per fragment to perform texturing with bilinear filtering. This causes a number of cache misses and also stalls the rasterizer upstream. Rasterizer utilization drops to 50% and the pixel shader utilization is 42% due to the number of cache misses. Throughput is 651 triangles per second.

Flexible Pipeline The first two passes are rasterization bound, so the allocation is changed to use only 1 tile for vertex processing. Neither pass requires pixel shading, so the stage is removed completely and the tiles are reallocated to increase the number of pixel pipelines to 20 (Figure 4). Since the input vertices do not contain any attributes other than position, we can safely remove interpolation and parameter vector calculation for the other attributes from the rasterization and triangle setup stages. As shown in Table 1, a total of 46% fewer dynamic instructions are fired compared to the reference pipeline.

We achieve more than a 100% increase in throughput over the reference pipeline for both the first and second passes. In the first pass, throughput improves 126% to 2110 triangles / second, and in the second pass, throughput also improves 126% to 1700 triangles / second. It is interesting to note that although load balancing between stages improves (Figure 5), the utilization in the rasterization stage actually decreases from 71% down to 49%.

The final pass is fragment processing limited due to the expensive texture memory accesses, so we pipeline that stage. We use only 1 tile for vertex processing, and allocate 12 pixel pipelines, with two tiles (pipelined) for the pixel shader: the first tile retrieves the samples, the second performs the interpolation. Using this configuration, throughput increases 13% to 736 triangles / second. Utilization of the rasterizer, first pixel shader, and second pixel shader are 34%, 31%, and 32%, respectively.

4.3. Case Study 3: Image Processing—Poisson Depth-of-Field

Image processing requires a quite different pipeline architecture than 3D rendering. Since we are using a general purpose architecture, we do not need to map the computation onto a traditional graphics pipeline. Consider the Poisson-disc fake depth-of-field algorithm by ATI [Sch04]. In a GPU implementation, the final pass of the algorithm would require rendering a large screen-aligned quadrilateral and performing the filtering in the pixel shader. The operation is extremely fragment bound since the scene contains only 2 triangles and the pixel shader must perform many texture accesses per output pixel.

In the flexible pipeline, each tile is allocated as an image filtering unit. We express the tile configuration using a 62-way StreamIt split-join. We use 62 tiles because the input requires one tile and incorporation of the split-join requires one tile. The color and depth buffers are split into 62 blocks. At 600×600 resolution, the blocks fit in the data cache of a tile. The tiles achieve a 38% utilization and a throughput of 122 frames per second. Due to the memory-intensive nature of the operation, 100% utilization is not reached—each cache hit incurs a 3-cycle latency.

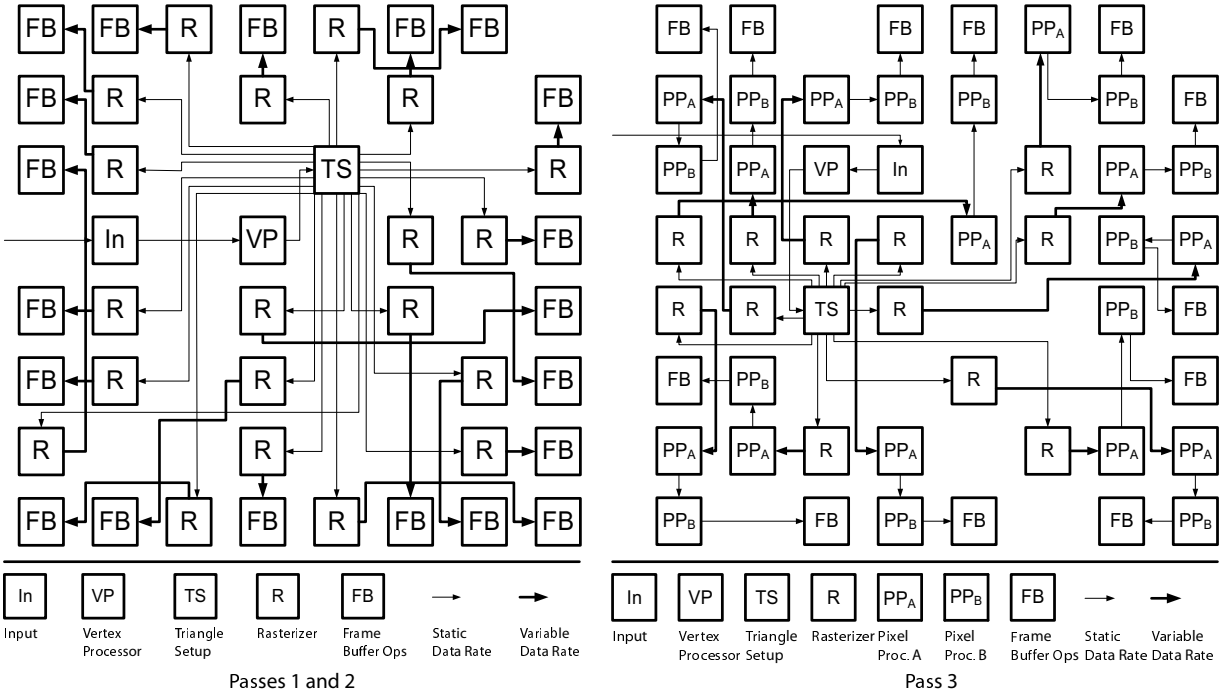

Figure 4: *Compiler generated layout for case study #2.*

4.4. Case Study 4: Particle System

In our fourth experiment, we consider automatic tessellation and procedural deformation of geometric primitives. We modify the vertex shaders to receive a triangle as input and output 4 complete triangles. Each vertex is given a small random perturbation. Since the input triangles require no shading and occupy only a small area on the screen, we expect this scene to be vertex-limited on the reference pipeline.

Reference Pipeline It turns out, however, that the bottleneck lies in the triangle setup stage. Triangle setup has a 49% utilization, the rasterizer is at 22%, and the other units are stalled ($< 4\%$) (Figure 5). In retrospect, this is unsurprising; our sort-middle architecture requires output vertices to be synchronized and contains only one triangle setup stage. Since the triangles are small, setup takes a proportionally large amount of computation relative to rasterization.

Flexible Pipeline Noticing that triangle setup is a bottleneck, we pipeline it by dividing the work onto two tiles and forwarding the necessary data. We also adjust the pipeline to remove unnecessary computation, such as texture coordinate interpolation in the rasterizer and computation of parameter vectors in triangle-setup. As shown in Table 1, 13% of the dynamic instructions are removed and the pipelined version obtains a performance increase of 157% over the reference pipeline. Even though we originally misjudged where the bottleneck would be, this case still illustrates the benefit of a flexible architecture: we can improve performance by transferring a tile from an idle stage to a busy one.

4.5. Discussion

In the above experiments, we have compared triangle throughput achieved by a fixed and a flexible resource allocation under several rendering scenarios. We have shown that flexible resource allocation can increase throughput up to 157%. We believe these results are indicative of the speedups that could be obtained by designing more flexible GPU architectures. However, the absolute performance obtained by Raw is orders of magnitude lower than current GPUs. As technology improves, this can be addressed by integrating more tiles on the Raw chip: Raw has a homogeneous and scalable design that is free of global communication structures. Combining the specialized computational resources of GPUs (e.g., the rasterizers) with the flexible communications infrastructure of Raw seems to be a promising research challenge.

5. Conclusions and Future Work

We have presented a graphics hardware architecture based on a multicore processor, where load balancing is achieved at compile-time, by reconfiguring the resource allocation. Both the 3D rendering pipeline and shaders are expressed in the same stream-based language, allowing for full programmability and load-balancing. Although our prototype

cannot compete with state-of-the-art GPUs, we believe it is an important first step in addressing the load-balancing challenge in graphics architecture.

We are working on alleviating the current limitations of our approach. We are studying the replacement of certain computation tiles by specialized rasterizers since this is the stage of the graphics pipeline that benefits most from specialization. We are also studying the memory hierarchy for optimal graphics performance, in particular the prefetching of textures. With prefetching, texture mapping performance can be greatly improved. Dynamic load balancing is the most exciting avenue of future work. A first intermediate step might exploit the statistics from the previous frame to refine resource allocation or switch between different pre-compiled versions of the pipeline. In the future, we expect that graphics hardware will be introspective and will be able to switch resource allocation within a frame or rendering pass depending on the relative load of computation units and on the occupancy of its buffers. Achieving the proper granularity for such changes and the appropriate state maintenance are the biggest challenges.

6. Acknowledgements

We thank Mike Doggett and Eric Chan for all their help on this project. We also thank David Wentzlaff, Patrick Griffin, Rodric Rabbah, and Jasper Lin for helpful discussions and feedback. John Owens was involved in the formative stages of this work. We thank the Raw group (esp. Jonathan Eastep) for support with the Raw simulator. Michael Gordon and Bill Thies thank their advisor, Saman Amarasinghe, for his support and guidance. Raw and StreamIt are supported by DARPA grant PCA F29601-03-2-0065, NSF award EIA-0071841, and the MIT Oxygen Alliance. In addition, StreamIt is supported by DARPA grant HPCA/PERCS W0133890 and NSF award CNS-0305453.

References

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph. 23*, 3 (2004). 3

[Bly05] BLYTHE D.: Windows Graphics Overview. *WINHEC 2005* (2005). 3

[EIH00] ELDRIDGE M., IGEHY H., HANRAHAN P.: Pomegranate: A Fully Scalable Graphics Architecture. In *SIGGRAPH* (2000). 2

[EMP*97] EYLES J., MOLNAR S., POULTON J., GREER T., LASTRA A., ENGLAND N., WESTOVER L.: PixelFlow: The Realization. In *SIGGRAPH / Eurographics Workshop on Graphics hardware* (1997). 2

[GTK*02] GORDON M., THIES W., KARCZMAREK M., LIN J., MELI A. S., LEGER C., LAMB A. A., WONG J., HOFFMAN H., MAZE D. Z., AMARASINGHE S.: A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS* (2002). 2, 4

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *SIGGRAPH* (2002). 2

[ISH98] IGEHY H., STOLL G., HANRAHAN P.: The Design of a Parallel Graphics Interface. In *SIGGRAPH* (1998). 2

[LKM01] LINDHOLM E., KILGARD M. J., MORETON H.: A User-Programmable Vertex Engine. In *SIGGRAPH* (2001). 3

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994). 5

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph. 22*, 3 (2003). 3

[MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader Metaprogramming. In *Graphics Hardware* (2002). 3

[MTP*04] MCCOOL M., TOIT S. D., POPA T., CHAN B., MOULE K.: Shader Algebra. In *SIGGRAPH* (2004). 3

[NK96] NISHIMURA S., KUNII T. L.: VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers. In *SIGGRAPH* (1996). 2

[ODK*00] OWENS J. D., DALLY W. J., KAPASI U. J., RIXNER S., MATTSON P., MOWERY B.: Polygon Rendering on a Stream Architecture. In *Graphics Hardware* (2000). 3

[OG97] OLANO M., GREER T.: Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Graphics Hardware* (1997). 5

[PH89] POTMESIL M., HOFFERT E. M.: The Pixel Machine: A Parallel Image Computer. *SIGGRAPH* (1989). 2

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A Real-time Procedural Shading System for Programmable Graphics Hardware. In *SIGGRAPH* (2001). 3

[Sch04] SCHEUERMANN T.: Advanced Depth of Field. *GDC 2004* (2004). 8

[TKA02] THIES W., KARCZMAREK M., AMARASINGHE S.: StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction* (2002). 4

[TKM*02] TAYLOR M. B., KIM J., MILLER J., WENTZLAFF D., GHODRAT F., GREENWALD B., HOFFMANN H., JOHNSON P., LEE J.-W., LEE W., MA A., SARAF A., SENESKI M., SHNIDMAN N., STRUMPEN V., FRANK M., AMARASINGHE S., AGARWAL A.: The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro* (2002). 2, 3

[TLAA03] TAYLOR M. B., LEE W., AMARASINGHE S., AGARWAL A.: Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *HPCA* (2003). 3

[TLM*04] TAYLOR M. B., LEE W., MILLER J., WENTZLAFF D., BRATT I., GREENWALD B., HOFFMANN H., JOHNSON P., KIM J., PSOTA J., SARAF A., SHNIDMAN N., STRUMPEN V., FRANK M., AMARASINGHE S., AGARWAL A.: Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA* (2004). 3

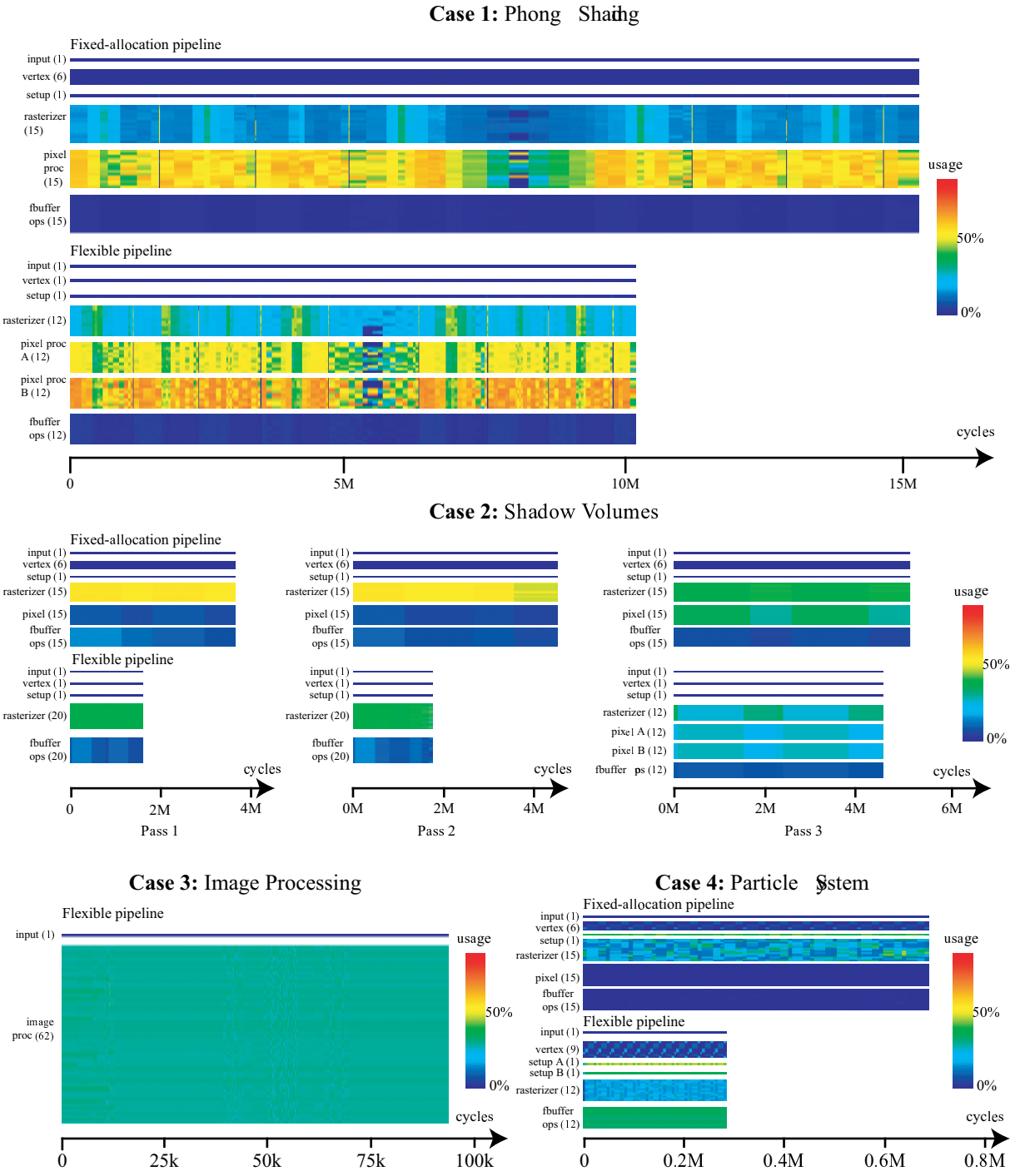

Figure 5: *Steady-state utilization graph.*