Nyami: A Synthesizable GPU Architectural Model for General-Purpose and Graphics-Specific Workloads

Jeff Bush
San Jose, California
jeffbush001@gmail.com

Philip Dexter†, Timothy N. Miller†, and Aaron Carpenter*
†Dept. of Computer Science
* Dept. of Electrical & Computer Engineering
Binghamton University
{pdexter1, millerti, carpente}@binghamton.edu

Abstract

Graphics processing units (GPUs) continue to grow in popularity for general-purpose, highly parallel, high-throughput systems. This has forced GPU vendors to increase their focus on general purpose workloads, sometimes at the expense of the graphics-specific workloads. Using GPUs for general-purpose computation is a departure from the driving forces behind programmable GPUs that were focused on a narrow subset of graphics rendering operations. Rather than focus on purely graphics-related or general-purpose use, we have designed and modeled an architecture that optimizes for both simultaneously to efficiently handle all GPU workloads.

In this paper, we present Nyami, a co-optimized GPU architecture and simulation model with an open-source implementation written in Verilog. This approach allows us to more easily explore the GPU design space in a synthesizable, cycle-precise, modular environment. An instruction-precise functional simulator is provided for co-simulation and verification. Overall, we assume a GPU may be used as a general-purpose GPU (GPGPU) or a graphics engine and account for this in the architecture's construction and in the options and modules selectable for synthesis and simulation.

To demonstrate Nyami's viability as a GPU research platform, we exploit its flexibility and modularity to explore the impact of a set of architectural decisions. These include sensitivity to cache size and associativity, barrel and switch-on-stall multithreaded instruction scheduling, and software vs. hardware implementations of rasterization. Through these experiments, we gain insight into commonly accepted GPU architecture decisions, adapt the architecture accordingly, and give examples of the intended use as a GPU research tool.

1 Introduction

Historically, high-performance computing (HPC) performance growth has generally followed Moore's law. This trend continues today, except for one major recent discontinuity: the adoption of GPUs. In terms of performance per Watt and performance per cubic meter, GPUs can outperform CPUs by orders of magnitude on many important workloads. The adoption of GPUs into HPC systems has therefore been both a major boost in performance and a shift in how supercomputers are programmed.

Unfortunately, this shift has suffered slow adoption because the GPU programming model is unfamiliar to those who are accustomed to writing software for traditional CPUs. In an attempt to bridge this gap, Intel developed Larrabee (now called Xeon Phi) [25]. Larrabee is architected around small in-order cores with wide vector ALUs to facilitate graphics rendering and multi-threading to hide instruction latencies. The use of small, simple processor cores allows many cores to be packed onto a single die and into a limited power envelope.

Although GPUs were originally designed to render images for visual displays, today they are used frequently for more general-purpose applications. However, they must still efficiently perform what would be considered traditional graphics tasks (i.e. rendering images onto a screen). GPUs optimized for general-purpose computing may downplay graphics-specific optimizations, even going so far as to offload them to software. Ideally, a GPU would have the capability to process both general-purpose and graphics-specific workloads with high performance and efficiency. Additionally, a modular approach can allow system integrators to make static selections among alternative components to optimize more for one paradigm or the other.

These factors make it important for the research and development communities to have effective tools that allow them to contribute new performance and energy-efficiency improvements, particularly as GPUs are traditionally very power-hungry [19]. Unfortunately, most GPU design details are proprietary, and there are limited options for accurate and modular architectural simulations.

With this work, we have attempted to take a fresh look at both graphics and GPGPU applications to develop an architecture, and associated simulation model, that performs well for both types of workload. Like Larrabee [25], we adopt a more traditional programming model, but we avoid the performance and die-area drawbacks by using a more GPU-like RISC ISA and pipeline architecture. As such, a given die area can hold a greater number of processor cores, increasing the aggregate throughput and performance per Watt.

Thus, we present *Nyami*, implemented as synthesizable logic in Verilog. This allows us to conduct design space exploration for GPUs, with very precise simulation of GPU operation at the RTL and gate levels. The Nyami architecture and model allow us to explore the trade-offs inherent in simultaneously designing a GPU for general-purpose and graphics-centric workloads in order to facilitate selecting an optimization target (graphics, GPGPU, or a compromise). The Nyami Verilog model provides a flexible framework for exploring architectural trade-offs that affect GPU performance in general, including changes to the cache hierarchy, pipeline structure, and hardware thread

978-1-4799-1957-4/15/$31.00 ©2015 IEEE

scheduling. Most importantly, Nyami is offered up as a new platform for research, designed to help researchers contribute to high-performance GPU research. Although Nyami is synthesizable, it is still easy to modify, which is important for testing architectural hypotheses and performing design space exploration. (Source code and documentation are available on-line.) Our accompanying software also includes an LLVM-based C++ compiler [16] that targets the Nyami ISA and a functional simulator written in C, allowing us to test the architecture across the various design stack levels, including cycle-precise simulation, instruction emulation, and power/area analysis.

While numerous open-source CPU implementations have been available for many years [2, 15, 20, 24], this is not true for GPUs. There are only two open-source fully-functional GPUs currently active, OpenShader [18], which is still under development, and Nyami. This currently leaves Nyami as the only fully-functional, synthesizable open source GPU implementation available. Nyami is also written to be a research tool, where the implementation in Verilog directly reflects the architecture, with minimal obfuscating performance optimizations. To demonstrate the significance of Nyami's contribution, we will describe the native architecture, as well as a number of design explorations, including hardware thread scheduling techniques, rasterization methods, and cache configuration design space exploration.

The rest of the paper is organized as follows. Section 2 discusses relevant existing work. Section 3 gives an overview of the Nyami simulation model and relevant baseline design choices. Section 4 presents examples of the significance of open-source GPU simulation and co-optimization techniques. Finally, Section 5 concludes.

2 Related Work

GPU Architectures The earliest 3D graphics accelerators implemented a fixed-function pipeline, specialized for graphics. But the progress of 3D games and other demanding graphics workloads have been relentless in their demand for more sophisticated rendering functionality, driving the adoption of "programmable shaders." Initially, only fragment processing was programmable, followed quickly by vertex and geometry processing, as more graphics acceleration was integrated on-chip. Modern GPUs are now based on a massively parallel multithreaded architecture, with numerous simple compute cores (numbering hundreds or thousands) on one chip, and as a result, they are capable of general-purpose computation. In general, GPUs are designed for highly data-parallel computation, which can be done concurrently using arrays of scalar or vector processor cores. Each thread of the computation is largely independent of the surrounding threads, as position, color, and texturing are calculated independently for each pixel.

In the early days of GPGPU research, there were no open application program interfaces (APIs) for general-purpose processing, forcing researchers to use OpenGL to render images that contained the computational results; input data came from textures, and outputs appeared in a rendered image. Since then, CUDA and OpenCL have exposed general-purpose compute functionality for most modern GPU platforms.

Intel's Larrabee architecture [25] is an often-cited modern example of a divergence from the architectural trends of traditional GPU architecture. Instead of numerous scalar processor cores, Larrabee is an array of general-purpose SIMD cores being used as a graphics processor, adapted from the existing x86 CPU architecture. These in-order x86 cores are enhanced with wide vector functional units, with very little special-purpose logic for graphics. While many traditional GPUs use hardware for task control, scheduling, and rasterization, Larrabee does all of this in software under the assumption that software control gives flexibility that confers performance advantages. Specifically, rasterization and graphics rendering is done in software. In Section 4, we will explore this option, as well.

Aspects of modern GPU architecture are described in other sections below. Top vendors of discrete high-performance GPUs include Nvidia and AMD (ATI). AMD and Intel also produce GPUs that integrate on the same die as their CPUs. There is also a sizable market for embedded GPUs, which are licensed as IP blocks to be integrated into systems-on-chip for portable devices, and vendors include Imagination Technologies (PowerVR), ARM, and Qualcomm. Relatively recent literature on the basics of GPU architecture includes [8, 17, 31, 32].

GPU Simulators GPUs and GPGPUs are an active area of research. However, most GPU architectures are proprietary, and information about internal details is trade secret. This makes it challenging for researchers to evaluate microarchitectural tradeoffs in a simulation environment. There are a few notable simulators that warrant discussion. The Guppy project modified an open source processor, LEON3, to be more GPU-like, and is synthesizable for FPGA [3]; unfortunately it is not widely available to the public. Similar work has been done to create soft GPGPU frameworks in FPGA hardware [4]. The ATILLA project is a cycle-accurate emulator for GPU architectures, written in C++ although with more emphasis on graphics than computation [11]. GPGPU-SIM is a cycle accurate GPGPU simulator also written in C++ that simulates modern architectures and has a very sophisticated execution pipeline model [6]. GPGPU-SIM has also been integrated with the Gem5 CPU simulator [29]. The Barra simulator is a parallelizable GPU/GPGPU simulator, but is based on arrays of PowerPC cores, making its performance characteristics diverge substantially from typical GPU behavior [9]. Multi2sim not only allows the simulation of GPUs but couples an AMD Evergreen GPU with a x86 CPU for heterogeneity [27]. PowerRed, while not a functional simulator, provides RTL-level power estimations for GPUs [22].

Open-Source GPUs Besides Nyami, there is only one other open-source fully functional GPU project in some state of development. OpenShader [18], mentioned above, is designed strictly for synthesis and high clock frequency, with a 10-stage scalar ALU. Currently, only the ALU is functional, while other components are implemented as simulation-only stubs.

There are also two open-source projects that are in early stages of development or have limited capabilities. Theia [28] is

available from OpenCores and is designed primarily for raytracing. Lacking floating point support, Theia currently has limited applicability to typical GPU applications. Silicon Spectrum is in the early stages of an open source community buy-out of their existing graphics accelerator IP (2D and fixed-function 3D) [7].

Overall, each of the discussed GPUs and simulators provides a useful piece of the GPU simulation puzzle including synthesizability, ease-of-use, open-source availability, accuracy, and flexibility. However, each is also significantly lacking in at least one of these categories. Nyami is a significant step towards a more complete simulation framework for modular testing of GPU architectural choices. Nyami also includes video and DRAM controllers, allowing it to be loaded into an FPGA to render graphics in real-time. This allows researchers to run more sophisticated tests than can practically be run in simulation.

3 Overview of Nyami's Architecture and Verilog Model

Nyami's default configuration is described in Table 1. While many of these configuration settings are standard for a many GPU and GPGPU architectures, a few warrant discussion below.

3.1 General Pipeline/Execution

Nyami executes a register-to-register RISC ISA. Arithmetic instruction operands can be scalars or vectors or a combination, and masks can be applied to vectors to select which destination elements are modified. Integer and floating point values are stored in the same register file. Load and store instructions support 8, 16, and 32-bit scalar sizes, and vector loads and stores support scatter/gather and strided accesses. Branch instructions have no delay slot; strand switching occurs to avoid a stall.

Hardware Threading Within the in-order, single-issue core (illustrated in Figure 1), we provide multi-threading capabilities to improve utilization and hide latencies. Each hardware strand has a distinct program counter and set of architectural registers and appears to software as an independent processor. The default is four strands. Simulated sensitivity studies (see Section 4.1.2) show that beyond four strands with the Nyami architecture, the performance begins to saturate, largely because of the native pipeline structure. These strands can be swapped into the pipeline on stalls or in a cycle-based interleaved fashion. Section 4 shows the performance comparison of the switch-on-stall (SoS) and barrel instruction scheduling policies.

The pipeline cannot be stalled, as it would also unnecessarily stall other strands in the pipeline. Thus on a stall, the offending strand must be rolled back, which invalidates all successive instructions for that strand in the pipeline. When the stalled strand is ready again, it is restarted from the point of the stall.

Nyami utilizes a unified scalar/vector pipeline, rather than separate units, and instructions can mix vector and scalar operands (with scalar registers being duplicated to all lanes of the operation). It also uses a very wide vector unit that can process sixteen 32-bit integers or floats in single-cycle and 4-cycle pipelines. All vector instructions support specifying an optional mask register to control which lanes will be updated. This is illustrated in Figure 2. This is especially useful for divergent

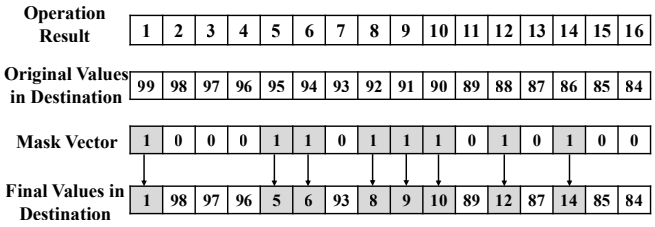

Figure 2. Operation results can be ignored by setting bits in the mask register to 0.

lock-step parallel execution, as found in Single Program Multiple Data (SPMD) architectures.

The Nyami microarchitecture also bears some similarities to Sun Niagara [14], which emphasizes thread-level parallelism and utilizes a simple in-order pipeline. Features inspired by Niagara include how strand scheduling and rollbacks are done, how coherency works, and the interface between L1 and L2 caches.

3.2 Divergence

Current GPUs [8, 17, 31, 32] employ a SIMT (single instruction multiple thread) architecture that allows an instruction cache to be shared across multiple threads and multiple ALUs ("lanes"), as long as those threads execute exactly the same sequence of instructions. When branches make those threads go different ways, this is known as "warp divergence" in Nvidia nomenclature, where the processor must alternate between threads in the same warp, substantially impacting performance. This has two major disadvantages: (1) a substantial amount of circuitry is dedicated to warp divergence and reconvergence; and (2) in ideal conditions, every lane is busy executing the same instruction as every other lane (albeit on different data), but the threads are technically independent, with the potential for divergence. This requires a significant amount of redundant decode and data flow logic across lanes. Nyami optimizes for the common case, making all lanes share a single front-end, putting many of the challenges GPUs face (e.g. divergence and convergence) under software control.

More specifically, Nyami puts the burden of divergence handling on the compiler. When there is potential for divergence among vector lanes (e.g. at the edges of a triangle), the compiler reserves a general-purpose scalar register (using its standard register allocator) to track which vector lanes (work items, in OpenCl nomenclature) are active on that path. Then each divergent path is executed serially, applying the current mask register to all vector instructions. When a reconvergence point is reached, the mask register is updated. Divergence and reconvergence points are computed statically by the compiler, and the compiler can allocate multiple registers to track nested divergent paths. The overhead of updating masks is relatively low, requiring only simple Boolean operations.

3.3 Cache Hierarchy

Each core has private L1 instruction and data caches, all of which are 4-way set associative for the default of 4 strands, to ensure high hit rate for each strand. The cores share the L2 cache. Communication between L1 and L2 uses separate re-

Attribute	Design Specification
Registers	30 general purpose scalar, 32 general purpose vector, 32 control
General Pipeline	In-order, single-issue, unified scalar/vector pipeline; scalar execution is duplicated across lanes; multi-threaded supported (4-way default); no stalling; rollback on mispredict and/or memory access
Execution	RISC, Single Instruction Multiple Data (SIMD),
Memory Access	8, 16, 32 bit scalars; 512-bit vectors
L1 Cache	Private 4-way set associative L1 I-cache and D-cache with miss queue (write-through/no-write-allocate);
L2 Cache	Shared write-back/write-allocate L2 with split transaction protocol
Cache Coherence	L2-based allocation; Directory based write update, processor consistency
Synchronization	Similar to load-linked/store-conditional; can implement spin-locks or lock-free operation

Table 1. The Nyami default architectural configuration is described here. Many parameters are flexible and can be changed to fit the simulation needs. For brevity, only defaults are listed here.

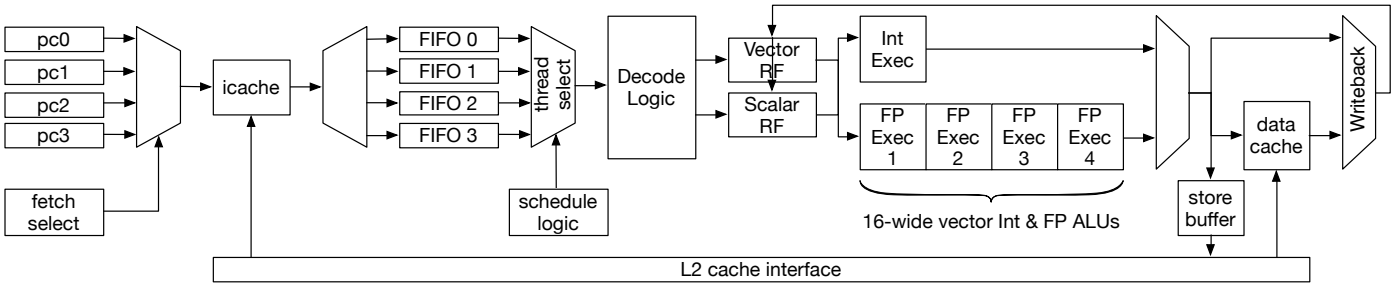

Figure 1. Nyami's native pipeline supports multiple simultaneous hardware threads (strands) and supports both vector and scalar operations, as shown here in a unified pipeline.

quest and response communication networks. L1 requests are sent using dedicated links to an L2 arbiter, which stores the request and processes them in the order of arrival. The response is broadcast on a bus, which each core checks to update its own cached copies. Upon a write from the core's store buffer, the L2 directory is used to specify an update to all cores that have that cache block in the local L1, sent via the broadcast bus. The request/response networks provide high throughput for requests and an efficient mechanism for data response and write updates, each of which is well suited for GPUs, which often have high read-to-write ratios (in our workloads, reads occur about 3.5 time as often as writes). Since L1 is write-through and write-no-allocate, coherence traffic is trivial, and only current sharers need to be tracked in the L2 directory. Additionally, the L2 manages the L1 cache allocation and therefore has global knowledge of the state of all caches. Writes from an L1 must happen in-order, but writes from different strands can occur in any order, providing processor consistency. The L2 is write-back/write-allocate to the memory to minimize off-chip accesses.

The L2 cache can accept a new request every cycle in the best case. Because it has high throughput (up to 64 bytes per cycle), a single core typically cannot saturate the L2 bandwidth. Arbitration logic within cores (among L1I, L1D, and store buffer) and between cores funnels L2 access requests down to a single request interface, allowing the L2 to service memory accesses in a predictable order, which simplifies coherence. As the number of cores grows, contention for L2 access increases, but the pipelined memory system maintains high throughput, and multi-threading allows processor cores to be effective at hiding the latency. Both the L1 and L2 hierarchies support hit-under-miss and can reorder requests, and there are queues at each level of the hierarchy to support large numbers of pending requests. The L2 cache is pipelined to 6 stages; with arbitration, the hit latency for the L2 with no contention is 9 cycles.

In order to (a) demonstrate the flexibility of Nyami simulator's and (b) justify the design choices in creating the default configuration, we present a cache sensitivity study for a suite of GPU benchmarks. Full discussion is in Section 4.

3.4 Pipeline Description

As shown in Figure 1, the instruction pipeline consists of the following stages:

- **Instruction Fetch**: This stage contains the instruction cache and maintains a FIFO of instructions and a program counter for each hardware strand. Each cycle, it selects a strand and issues its PC to all four ways of the instruction cache tag and data SRAMs. These SRAMs have one cycle of latency. Combinational logic after the SRAMs detects a hit or miss, uses a multiplexer to select cache line data from the appropriate way, and enqueues it into the requested strand's FIFO. If a cache miss occurs, instruction fetch for that strand is suspended and the missed address is put into a load miss queue.
- **Strand Select**: This stage selects the next strand to issue an instruction from according to the configured policy (e.g. round-robin). Strands that are not ready are skipped. This can occur for empty instruction FIFOs, when waiting on a data cache miss/full store buffer, or when a long-latency instruction has been issued. For instructions that require mul-

tiple execution cycles (e.g. scatter/gather vector load/store), this will issue the same instruction multiple times with a signal that indicates which lane the operation is for.

- **Decode**: This stage selects the appropriate operand indices from the instruction and issues them to the register file. The register file has one cycle of latency, so the values will be available to the next stage. This also contains a bypass network that forwards results from subsequent stages where appropriate.
- **Execute**: There are two parallel pipelines in the execute stage, both of which support scalar and 16-wide vector operations. A pipeline with one substage handles integer operations and also detects mispredicted branches. The floating point pipeline has four substages. There is a structural hazard at the end of this stage where the different pipeline lengths converge. The strand select stage schedules around this by tracking which instructions have been issued.
- **Memory Access**: In this stage, memory **load** instructions (scalar and vector) will access the data cache tag and data memories. Like in the instruction fetch stage, these are accessed in parallel. As the SRAMs have a cycle of latency, cache hits are detected a cycle later and the appropriate cache line is selected by a multiplexer via combinational logic before being passed to the next stage. If a cache miss occurs, the address goes into a separate load miss queue. This stage also contains the **store** buffer. When a load is executed, the store buffer will be checked to see if the address is already present and, if so, bypass the cache. Memory stores do not access the local data cache. They instead go through the L2 cache, which eventually broadcasts and update to all cores. This maintains proper coherence ordering.
- **Writeback**: This stage sets control signals that update the register file when needed. The mask register (if used) is a set of enable signals that control which vector lanes are updated.

There is a rollback controller that detects rollback conditions from various stages and issues squash signals to appropriate stages, prioritizing when multiple squashes are triggered simultaneously from different stages.

3.5 Compiler

A full C/C++ toolchain has been ported to this architecture, based on the LLVM Compiler Infrastructure [16]. LLVM has robust support in its intermediate code representation for vector types and operations. The backend implementation for this processor exposes specialized vector functionality directly using built-in intrinsics and standard GCC vector extensions. Vectors are first class types, which can be local variables, struct/class members, and function arguments. Standard arithmetic operators can be used on them, which will generate the appropriate vector instructions. For example:

```
// Initialize a vector with an array constant
vec16f_t a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    12, 13, 14, 15, 16 };
vec16f_t b;

// Add two vectors
a = a + b;

// copy a lane of a vector into a scalar variable
float c = a[12];

// Copy a scalar into a vector lane
b[11] = 0;
```

Vector and scalar types can be mixed using an intrinsic that expands a scalar to a vector. As mentioned above, the processor uses a unified pipeline, so scalar and vector operands can be mixed. The compiler is intelligent about generating instructions that mix types automatically where possible. For example:

```
float d;
a = a + __builtin_vp_makevectorf(d);
```

Will generate:

```
add_f v0, v0, s0
```

A flexible intrinsic `__builtin_vp_vector_mix` allows support of predication from C++. It requires three parameters: a bit mask and two vector parameters. Each of the low 16 bits of the mask corresponds to a lane of the vector. A one bit will take the lane from the first vector parameter, and a zero will take it from the second. This intrinsic does not correspond to an instruction but will infer the appropriate masked forms. For example, the following:

```
vec16f_t a, b;
...
a = __builtin_vp_vector_mixf(mask, a + b, a);
```

Generates a single instruction:

```
add_f_mask v0, s0, v0, v1
```

The `add_f_mask` instruction uses a scalar register (e.g. s0) to specify which lanes to take from vector registers (e.g. v0 and v1).

3.6 Simulation Environment and Benchmarks

There is a large body of sophisticated parallel benchmarks in the literature. For the purposes of this paper, we have restricted ourselves to a few benchmarks. The first two are more narrow microbenchmarks, while the remaining 3D tests exercise a more complex, real-world workload. Rendered examples are found in Figure 8.

- **Alpha Blend**: Blends a 64x64 image into a 64x64 destination image (32-bit pixels). This utilizes integer arithmetic to extract and pack color elements and to blend the pixels, as well as memory accesses to read and write the images.
- **Dot Product**: Computes a large number of dot products from a memory array. This stresses floating point arithmetic.
- **Teapot**: Using the 3D engine, renders the standard Utah teapot with a Phong shading model into a 512x512 pixel output buffer. The model has 2304 triangles and 1364 vertices.
- **Torus**: Using the 3D engine, renders a torus shape with 288 triangles and 144 vertices, also Phong shaded into a 512x512 pixel output buffer.

- **Cube**: Using the 3D engine, render a simple cube, mapping a 128x128 texture on each side, using bilinear filtering into a 512x512 pixel output buffer. The model has 12 triangles and 24 vertices and stresses texture mapping.

Nyami features a set of built-in performance counters for evaluating various performance metrics and is also easy to instrument with additional Verilog code for additional data collection. As a whole, the Nyami simulation environment provides a flexible, synthesizable GPGPU-like architecture, with the further advantages of simplicity and precision.

The following configurations are available for testing and analysis:

- A C-based instruction-accurate functional simulator is provided, used as a reference model for co-simulation and verification and as a software prototyping platform. It runs at a fast rate (the equivalent of around 25 MHz when run on modern desktop hardware).
- The design can also be simulated using the Verilator [26] simulation framework, which allows cycle accurate simulation and complete inspectability at the waveform level. The effective clock speed of this simulation running on modern desktop hardware is around 70 kHz.
- The system can also be synthesized to run on FPGA at a speed of 30 MHz. This configuration also features a SDRAM controller and VGA display controller, allowing real time display. The design can be instrumented in software to gather performance data.

Additionally, because the Verilog code is synthesizable, it can also be used to gather power and area statistics for various technology nodes. The multi-platform portability of Nyami makes it an ideal environment for GPU architectural exploration and evaluation.

3.7 3D renderer

A software 3D renderer has been implemented to explore the capabilities of this architecture and provide a more complex benchmark that exercises many parts of the implementation. The engine was implemented almost completely in C++ (utilizing platform specific intrinsics that are exposed by the compiler), and takes advantage extensively of both SIMD and hardware multi-threading. The renderer uses tile based rendering, which is sometimes known as a sort-middle architecture [12]. There are two phases during rendering:

- **Geometry**: During this phase, a vertex shading routine is executed on the vertices. The vertex shader is explicitly parallelized in SIMD to process up to 16 vertices simultaneously. The vertices in the array are statically assigned (by index) to hardware strands. With four strands running on one core, up to 64 vertices are in progress at any time. The computed vertex parameters are written to an output array to be used by the next phase. Strands wait at a barrier when they reach the end of the vertex array.
- **Pixel**: The screen is broken into tiles (the size is configurable in software). Each strand reads the next unrendered tile and renders it completely before moving onto the next. This allows the active tiles to fit in the L2 cache and provides a low-contention way of scaling work as the number of strands/-cores increases. The strands rasterize all triangles that overlap the tile (using a bounding box test). This is described in more detail in Section 4.2. The rasterizer outputs aligned 4x4 pixel block coordinates with coverage masks. These blocks contain 16 pixels, which map exactly to the width of the vector, allowing 16 instances of the pixel shader to be run simultaneously on each strand. The renderer first performs a depth buffer check, followed by perspective correct interpolation of the pixel parameters. It then calls a shader function that computes the color values. With a single core and four strands, there are up to 64 pixels being shaded simultaneously. When a strand has finished rendering the tile, it uses a data flush instruction to explicitly push the contents out of the data cache.

A texture sampler function can be called by the pixel shader. This is implemented completely in software and performs bilinear filtering. Like other parts of the renderer, it is vectorized and computes values for up to 16 texels at a time using vector gather load instructions.

3.8 Pipeline Visualization Tool

Included in the Nyami simulation environment is a visualization tool that displays the state of the running strands over time. Several papers have explored using graphical representations of pipeline state to examine behavior, including [5, 30].

Figure 3 illustrates this functionality with four strands. In the figure, each iteration of a loop can be seen as a repeating pattern of two red stripes followed by several yellow stripes as the pipeline handles multiplications. The visualization tool illustrates memory stalls, active strands, and dependency stalls in an effort to aid in performance analysis.

4 GPU Architectural Design Exploration

We will now demonstrate some of the possibilities presented when using Nyami to answer theoretical questions about GPU architectural choices. Utilizing Nyami as a scientific research tool, we have conducted several experiments, presented below. Each experiment has required a fundamental architectural change, which we were able to implement very quickly. These include instruction scheduling policies, hardware vs. software rasterization, varying cache size and associativity, and the addition of a write-combining load/store queue.

4.1 Scheduling: Barrel vs. Switch-on-Stall

GPUs take advantage of thread-level and data-level parallelism, while also adapting well to increasing memory demand. The memory access latency is the greatest challenge to maximizing throughput, so GPUs are designed to have many simultaneous transactions pending. The memory system performs best when there are numerous outstanding requests to process, particularly when they can be reordered, and the processor cores hide much of the latency by assigning many threads to the same compute hardware, switching among them as memory transactions stall and complete. Ideally, both the memory and compute systems

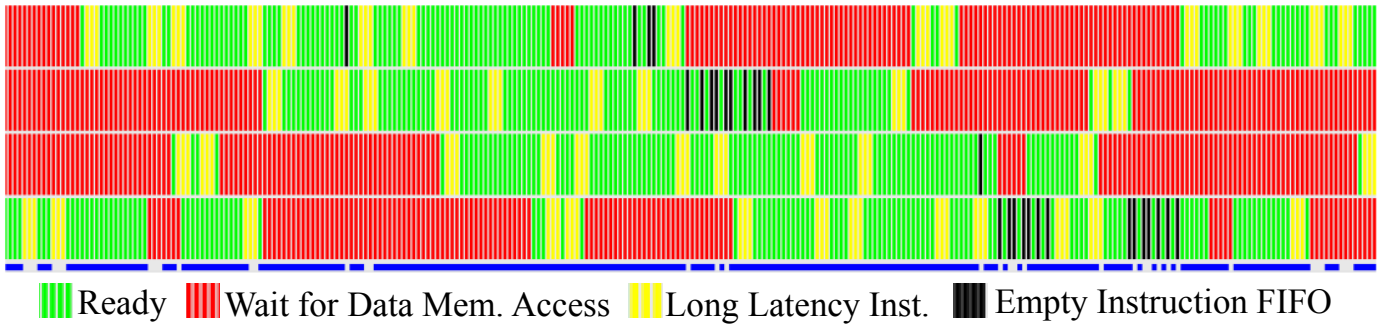

Figure 3. Four strands states over time using the visualization application. Red indicates a strand waiting on a memory access. Green indicates a ready strand. Yellow represents a strand waiting on a long latency instruction. Black shows a strand waiting due to an empty instruction queue.

will always have an abundance of pending work to perform. With enough work available, any time an instruction stream encounters a stall condition (RAW hazard or load dependency), a processor core will typically be able to switch to another thread with instructions ready to execute. This allows computation and memory access to occur in parallel, maximizing the amount of logic that is doing useful work at any one time.

Although there has been some substantial study of the effects of multithreaded instruction scheduling for CPUs, CPUs are optimized to minimize memory latency and maximize single-thread throughput, while GPUs are optimized for high memory latency and maximize overall throughput of all running threads. Therefore, it seems appropriate to utilize Nyami to explore this question for GPUs.

For instruction scheduling, we primarily have two options. Some GPUs, e.g. from Nvidia, opt for coarse-grained multi-threading, which we refer to as *switch-on-stall (SoS)*. With this policy, instructions for a given thread are issued back-to-back until a stall condition is identified, and the next thread is selected. An alternative is to use fine-grained multi-threading. The extreme case is *barrel processing*, where a strand switch occurs on every cycle and two instructions from the same strand are not allowed to simultaneously occupy the same pipeline.

Our first attempt to assess the impact of these two instruction scheduling policies employed artificial workloads. The result was a very small performance difference, mostly due to too many simplifying assumptions. From a memory perspective, the two policies are effectively equivalent: the order that strands are unblocked by the L2 cache has a bigger impact on overall throughput than the order in which they block. It was not until we executed real workloads on Nyami in a cycle-precise manner that any substantive differences were revealed, demonstrating the value of a synthesizable GPU implementation.

Our original hypothesis was that SoS would have superior performance because of its memory access behavior. When executing a graphics workload, it is typical for different threads to execute the same or nearly the same sequence of instructions. We therefore expected that the barrel processor would alternate between phases of compute-intensive instructions and phases where every strand floods the memory system with requests. These floods would cause slowdowns due to high contention access to the memory system. By contrast, SoS spreads out memory accesses, commonly executing several arithmetic instructions before encountering another stalling memory instruction. In out-of-order CPUs, workloads with more spread-out accesses often perform better than those that cluster memory accesses together, and we assumed that this would apply here as well.

This hypothesis was not supported by the evidence. For one thing, although barrel and SoS have different memory behavior at the start of a rendering pass, the memory system's latencies will naturally cause strands in a barrel system to desynchronize, spreading out memory requests in steady-state, where memory access patterns become more similar to those created by the SoS policy. More importantly, SoS performance was hurt most by contention for a limited-capacity load-store queue, which we will explore further in Section 4.1.1.

An obvious advantage of using a barrel policy over SoS is that a barrel processor can be architecturally simpler. If the number of strands is greater than the number of pipeline stages, then a barrel processor needs no logic to track register-to-register data dependencies and requires no branch prediction. In fact, the only time a barrel processor needs to stall is on high-latency cache misses, and this can be detected far enough in advance that there is also no need for hardware to support rollbacks. On the other hand, barrel processors rely on never having two instructions from the same strand in the same pipeline at the same time, and they also rely on there being a minimum number of cycles between issuing instructions from the same strand. As a result, if many strands assigned to a barrel processor are blocked, it may be forced to insert idle cycles to respect those assumptions.

4.1.1 Improving the Load/Store Queue

Nyami was originally designed with a single-entry store queue for each strand. The store queue is necessary to support RAW data flow through memory within a single strand without waiting on coherence with other caches, and this structure is also used for cache management actions such as cache flushes and synchronized accesses. Based on static analysis of benchmark instruction sequences, we observe that memory stores occur at a rate of 1 per 6 instructions, and a dynamic analysis finds that stores occur at an average rate of 1 per 21 cycles (including stall cycles). These would suggest that a single store entry is sufficient. But for some workloads with heavy back-to-back stores

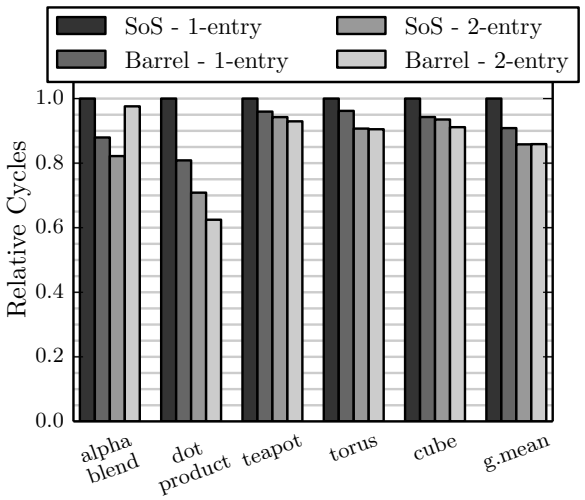

Figure 4. For each benchmark, we see the performance comparison of barrel vs. switch-on-stall strand scheduling, for both a single entry and double entry store queue. The baseline is SoS with 1-entry.

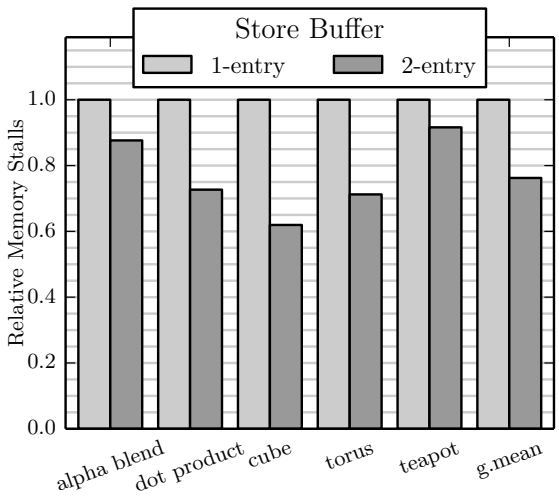

Figure 5. By increasing the store queue to 2-entries, we reduce the data cache and store queue stall cycles, relative to 1-entry store queue.

and scatter stores, writes can occur in bursts, making the single-entry buffer a serious bottleneck. The barrel policy naturally spaces out memory accesses within a strand, while the SoS policy will attempt to issue stores back-to-back, resulting in roll-backs and wasted cycles.

We therefore took advantage of Nyami's architecture-oriented structure to implement a two-entry store buffer, which substantially alleviated the congestion. When queuing a store in an empty buffer, the request is forwarded immediately to the L2 cache. A second entry could also be pipelined behind the first and forwarded to the L2, but we instead chose to hold it back to support write combining, which also benefited performance.

With the single-entry store queue, SoS was slower than barrel by 10%, on average. With the two-entry store queue, store-related stalls were reduced substantially and even eliminated for some benchmarks. SoS sped up by 16.4%, barrel sped up by 6.4%, and the performance gap shrank, making SoS only 0.6% slower than barrel. The remaining performance difference is influenced by many small factors. SoS still has slightly higher contention for the store queue, and barrel also has slightly better cache hit rate. Figure 4 shows the overall performance results of these comparisons, while Figure 5 explains the performance improvements in terms of reduction in stall cycles, where a 2-entry store queue reduces memory stalls by 24% on average. Since a barrel scheduling policy can more efficiently use a simpler and smaller store queue implementation than SoS, barrel has yet another circuit area and power advantage over SoS.

4.1.2 Varying Strands Per Core

To further examine the differences between SoS and barrel, we also varied the number of strands per core. The default number of strands is 4, but 2, 8, and 16 were also tried. The performance comparison of these experiments is shown in Figure 6. With more strands, there is greater potential to hide memory stalls. Adding more strands hides more L1 miss latency for either scheduling policy. Beyond four strands, adding more

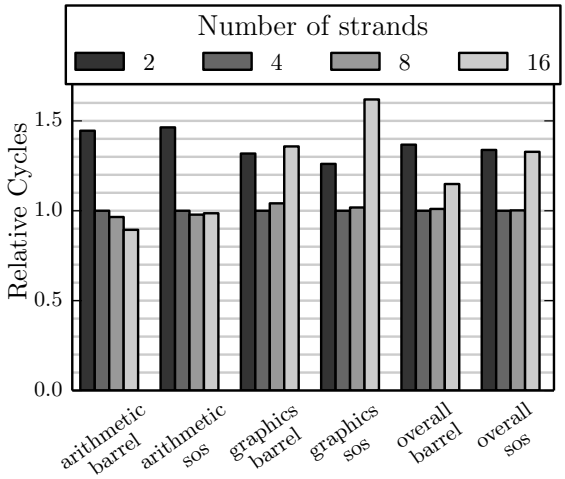

Figure 6. Performance of the Nyami architecture for the different numbers of available strands, against the baseline of 4 strands.

strands has diminishing returns. For 16 strands, the general purpose arithmetic benchmarks continue to show the trend of saturating performance, but the graphical benchmarks have substantially worse performance. This is because the graphical benchmarks have high memory footprints (each strand renders a 64x64 block of pixels and may additionally access texture data) and therefore suffer more cache misses since more simultaneous threads increase the working set.

4.2 Rasterization: Hardware vs. Software

Rasterization is the process of converting polygons (specified by vertexes or vectors) into pixels (raster images). This can be done in hardware or software. While software rasterization is not likely to match the peak performance of a hardware rasterizer, software is a sufficient alternative, particularly as well-designed software can completely avoid unnecessary work in a way that fixed hardware cannot. Here we present a comparison between a Larrabee-like software rasterizer and a performance-ideal hardware rasterizer. Figure 8 shows the rasterized images for the hardware and software methods used.

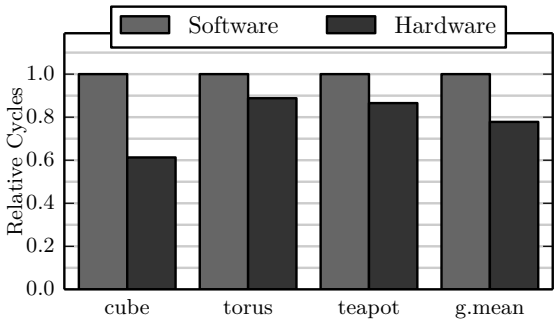

Figure 7. The relative cycles used by using a system with a hardware rasterizer, relative to software rasterizer.

For software rasterization, we employ a recursive descent approach, similar to that used by Larrabee, which identifies which pixels in a square area of the rendering surface intersect with a triangle to be rasterized [1]. At each level of recursion, a surface patch is subdivided into a 4x4 grid of smaller patches. Utilizing half-plane equations, whole patches can be classified as "trivially rejected," "trivially accepted," and "partially overlapped". Trivially rejected patches are passed over without recursion. Trivially accepted patches are completely filled, and partial overlaps are further subdivided. Recursion stops at a 4x4 pixel patch, and a 16-bit mask is computed as an operand to Nyami's vector ALU.

Our hardware rasterizer is designed to be a drop-in replacement for the software rasterizer, requiring minimal modification to our benchmarks and isolating the performance effects exclusively to the critical path in rasterization. This rasterization logic sweeps the bounding box of the triangle to be rendered, calculating 4x4 patches. These patches are returned to software in the form of coordinates and 16-bit masks, where bits represent the inclusion of corresponding pixels in the triangle to be painted. Inclusions are computed by the intersection of three half-planes based on the triangle edge line equations. The hardware rasterizer automatically sweeps past patches that fall entirely outside of the triangle, and this has no impact on software performance, because the sweep is done in parallel to painting dequeued patches.

On average, the hardware rasterizer speeds up rendering performance by 28% (for both barrel and SoS). The graphics benchmarks perform several tasks, including geometry, rasterizing (finding out which portions of the screen to paint), and rendering (painting pixels). The hardware rasterizer virtually eliminates the time overhead of rasterizing, replacing the Larrabee recursive algorithm with hardware that operates in parallel. For the purposes of our experiments, the hardware rasterizer we implemented is nearly ideal in performance and therefore will require more circuit area than a well-optimized design. However, our performance counters show it to have very low utilization such that it could be shared among just under 600 strands, thereby making it a relatively cheap solution even in the worst case.

4.3 Varying Cache Size and Associativity

To illustrate the value of Nyami as an experimentation platform, we present another demonstration of its flexibility. In software-based simulators like MARSSx86 [21] and SESC [23], changing cache parameters is very simple, because the parameters are merely inputs to software that managed dynamically generated data structures. On the other hand, parameterizing hardware can be much more difficult. For instance, a software implementation of a cache tag array will store tag data in 32-bit integer variables, leaving most of the bits unused. In hardware, data structures must be sized much more tightly to the specifications of the cache size and associativity. When these parameters are adjusted in Nyami's configuration, they automatically affect memory word and data bus sizes all across the circuit.

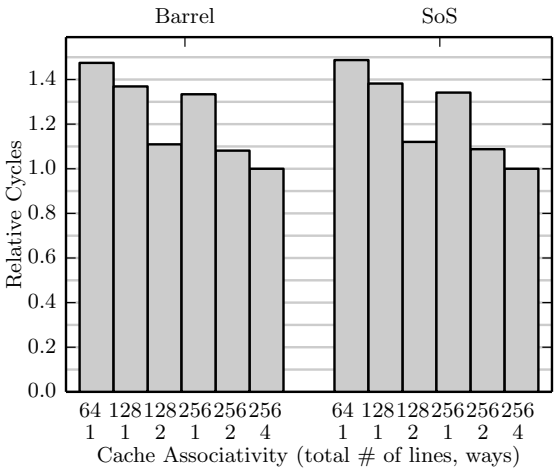

Figure 9. Geometric mean across benchmarks of execution cycles for different cache sizes, relative to baseline (256 lines, 4-way set associative). The top numbers (64, 128, and 256) refer to the total size of the cache (in number of lines) and the bottom numbers (1, 2, and 4) represent the associativity (1-way, 2-way, and 4-way).

When varying cache parameters, the effects on performance and cache hit rate are not at all unexpected, as shown in Figure 9. As expected, smaller caches have lower hit rate, increasing execution time, and increasing associativity has more benefit than increasing capacity.

4.4 Synthesis Report

Using Quartus, we synthesized the baseline configuration of Nyami for the Altera Cyclone IV E (EP4CE115F29C7) FPGA. Including DRAM, video controller, and a single 4-strand processor core, Nyami occupied a total of 92,186 logic elements (59,154 combinatorial only, 6,419 register only, 26,613 both), which is 81% of the device's 114,480 total logic elements. Additionally, 128 of 532 (24%) dedicated multiplier blocks were used. Since Nyami is not optimized for clock frequency, static timing analysis reported about 30 MHz.

Synthesis could also be done using transistor technology models to extract transistor-level area and power analysis. Using Synopsys Design Vision [10] and NanGate 45nm transistor technology model [13], we did a preliminary synthesis as a proof of concept. For brevity, only a few simple components are included in this analysis. The default 32-bit entry, 1K SRAM cache consumes 20.5mW and uses 244,000 μm^2. The simple

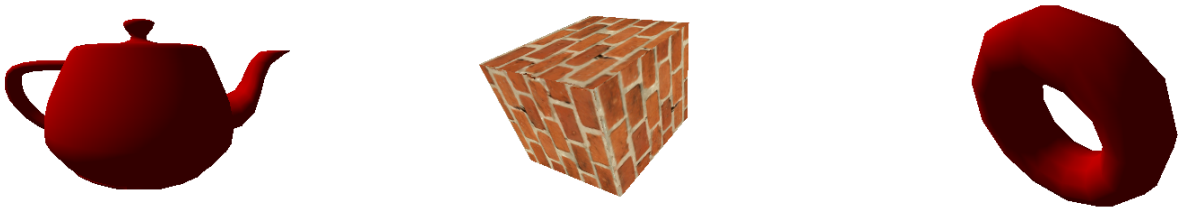

Figure 8. Output of Teapot, Cube, and Torus benchmarks. (To save on toner, original black backgrounds repainted white.)

arbiter requires only 14.4 μW and 52.9 μm^2. A 64-bit, 2-entry synchronous FIFO synthesized to 1303.4 μm^2 and consumes 261.7 μW.

5 Conclusions

In this paper, we have presented Nyami, which is a new GPU and GPGPU architecture, simulation framework, and research tool. Unlike other GPU simulators, Nyami is based on a synthesizable circuit written in Verilog, providing cycle-precise simulation of a multithreaded GPU. Nyami itself explores new GPU architectural choices, borrowing the best design decisions from Larrabee and traditional GPUs, and we employ Nyami to explore the impacts of a set of architectural choices.

Through these experiments, we show the performance implications of using barrel and switch-on-stall instruction scheduling policies, identify and fix bottlenecks in the store queue, and demonstrate the benefits of using a dedicated hardware rasterizer over an equivalent software technique. These demonstrate Nyami's value to GPU researchers, both in terms of modularity for design space exploration and as the only fully-functional, synthesizable open source GPU currently available. Finally, we use Nyami to answer interesting theoretical questions about GPU architectural choices.

Full source code for this is available on github at the following links:

`https://github.com/jbush001/NyuziProcessor`

`https://github.com/jbush001/NyuziToolchain`

The architecture is continuing to evolve and details have most likely changed since publication of this paper. The sources used to prepare the results in this paper are branched as 'ispass-2015' in the above projects.

References

[1] Michael Abrash. Rasterization on larrabee. *Dr. Dobbs Journal*, 2009.

[2] Gaisler Aeroflex. Leon3 User Manual. `http://www.gaisler.com/`.

[3] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W. Wong. Guppy: A GPU-like soft-core processor. In *Int'l Conf. on Field-Programmable Technology*, pages 57–60, 2012.

[4] K. Andryc, M. Merchant, and R. Tessier. Flexgrip: A soft GPGPU for FPGAs. In *Int'l Conf. on Field-Programmable Technology*, 2013.

[5] A. Ariel, W. Fung, A. Turner, and T. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 164–174. IEEE, 2010.

[6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE Int'l Symp. on Perf. Analysis of Systems and Software*, pages 163–174, 2009.

[7] Francis Bruno. Open Source Graphics Processor (GPU) [Kickstarter]. `http://www.kickstarter.com/projects/725991125/open-source-graphics-processor-gpu`.

[8] J. Chen. Gpu technology trends and future requirements. In *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 2009.

[9] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator. In *Int'l Symp. on Modeling, Analysis, & Simulation of Computer and Telecummunication Systems*, pages 351–360, 2010.

[10] Synopsys Design Compiler. http://www.synopsys.com/.

[11] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. In *Int'l Symp. on Performance Analysis of System and Software*, pages 231–241, 2006.

[12] Matthew Eldridge. *Designing graphics architectures around scalability and communication*. PhD thesis, STANFORD UNIVERSITY, 2001.

[13] Nangate Inc. Nangate 45nm open cell libary. http://www.nangate.com, 2009.

[14] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005.

[15] D. Lampret. OpenRISC 1200 IP Core Specification. http://opencores.org.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[18] T. Miller. OpenShader: Open architecture GPU simulator and implementation. `http://sourceforge.net/projects/openshader/`.

[19] European Parliament. Eu commission regulation regarding energy-related products. `http://www.eup-network.de/fileadmin/user_upload/Computers-Draft-Regulation-subject-to-ISC.PDF`.

[20] I. Parulkar, A. Wood, J. Hoe, B. Falsafi, S. Adve, J. Torrellas, and S. Mitra. OpenSPARC: An open platform for hardware reliability experimentation. In *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2008.

[21] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of the 2011 Design Automation Conference*, 2011.

[22] K. Ramani, A. Ibrahim, and D. Shimizu. Powerred: A flexible power modeling framework for power efficiency exploration in GPUs. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2012.

[23] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi P. Sack, K. Strauss, and P. Montesinos. SESC simulator. `http://sesc.sourceforge.net`, January 2005.

[24] C. Santifort. Amber ARM-compatible core. http://opencores.org/project,amber.

[25] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, pages 10–21, 2009.

[26] W. Synder. Verilator. `http://www.veripool.org/`.

[27] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: A simulation framework for CPU-GPU computing. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 335–344, 2012.

[28] Diego Valverde. Theia: ray graphic processing unit. `http://opencores.org/project,theia_gpu`.

[29] H. Wang, V. Sathish, R. Singh, M. Schulte, and N. Kim. Workload and power budget partitioning for single-chip heterogeneous processors. In *IEEE Int'l Conf. on Parallel Archtiecture and Compilation Techniques*, pages 401–410, 2012.

[30] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin. Performance analysis using pipeline visualization. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, 2001.

[31] C. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, 2011.

[32] Y. Zhang, L. Peng, B. Li, J.-H. Peir, and J. Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205–215. IEEE, 2011.