NyuziRaster: Optimizing Rasterizer Performance and Energy in the Nyuzi Open Source GPU

Jeff Bush (jeffbush001@gmail.com), Mohammad A. Khasawneh (mkhasaw1@binghamton.edu),
Khaled Z. Mahmoud (kmahmou1@binghamton.edu), Timothy N. Miller (millerti@binghamton.edu)

***Abstract*—The Intel Larrabee GPU was not a traditional GPU. Designers reasoned that many functions typically implemented in dedicated hardware could instead be done efficiently in software using a wide vector ALU. Among these functions was rasterization. To compensate for the absence of a dedicated rasteriser, Intel engineers designed a recursive rasterizer algorithm that efficiently utilizes Larrabee's compute resources. However, researchers recently have shown software rasterization to have substantial overhead for some workloads.**

This paper develops and evaluates a range of hardware rasterization accelerators for Nyuzi, a complete and fully synthesizable open source GPU inspired by Larrabee. An open source GPU is of great value to researchers wishing to conduct experiments on GPUs for graphics and high performance computing applications. Recently, Nyuzi's developers showed rasterization to extend benchmark runtime 10 to 30 percent. This makes rasterization a prime target for optimization, which we address along two orthogonal dimensions.

The first dimension trades off circuit area against throughput. Accelerators that take more cycles to compute pixel masks can be shared among fewer Nyuzi threads. From this we compute scalability of each alternative design. The second dimension trades off hardware versus software, decomposing rasterization into phases and identifying the software overhead of each phase. This evaluation suggests how these tradeoffs apply to other GPU architectures and would benefit GPU designers if considered in design decisions. We observe that the original Larrabee rasterizer algorithm is inefficient for small triangles and recommend replacing it with a bounding box scanning algorithm, which is much faster in both hardware and software.

I. Introduction

In 2007, Intel announced the Larrabee processor [18]. Larrabee was an experiment in doing graphics in a more CPU-like architecture. It was ultimately more successful in high performance computing, and it has appeared in that arena under the name Xeon Phi. Nevertheless, the Larrabee architecture is an interesting alternative to the traditional SIMT (Single Instruction Multiple Thread) GPU architecture used by AMD and Nvidia, and it has its own advantages.

Nyuzi is a new open source GPU architecture, described as "Larrabee-inspired," sharing some architectural choices, but shedding the baggage of the x86 instruction set. Nyuzi is fully synthesizable, running at about 60MHz in a low-end Altera Cyclone IV FPGA. Along with a fast functional simulator, the Nyuzi RTL can be simulated with Verilator [19], yielding an excellent cycle-precise GPU simulator. The test environment also includes a reference DRAM controller and video output for simulating a full system-on-chip. The RTL is parameterized and easily modified, and we have taken advantage of this in our research.

This paper considers the problem of triangle rasterization, which is the process of converting a geometric representation of triangles into raster coordinates. Larrabee lacks dedicated hardware for rasterization. Its designers reasoned that software can choose among specialized and optimized alternative code paths in ways that a rigid hardware implementation cannot. They point out that software can skip over unnecessary steps, while a hardware pipeline must always perform all phases of execution (even if some stages do no work), wasting time and energy. Instead of using hardware, Intel software engineers developed a clever recursive vectorized software algorithm [2]. However, in a recent paper [9], it was shown that this algorithm adds between 10 and 30 percent to the runtime of various benchmarks, making it an interesting target for optimization. The rasterizer in [9], however, was not synthesizable. In this work, we develop a range of synthesizable and non-synthesizable rasterization hardware accelerators, considering different tradeoffs.

The first dimension we consider is that of scalability. We design a progression of rasterizers, where all rasterization functions are implemented in hardware. With each design, gate count is reduced, with the consequence of requiring more cycles to compute pixel masks and hence fewer triangles being rasterized concurrently. Longer mask computation time results in a reduction in the number of concurrent threads one rasterizer can support. From this, we can observe how gate count for each design will increase as the number of supported threads is increased and how design tradeoffs affect the maximum number of threads that can be supported.

The second dimension we consider is software/hardware trade-off. Rasterization can be divided into sets of calculations for setup, iteration, mask computation, reject/accept,

This work was supported in part by the National Science Foundation under grant CCF-1115564.

and clipping. We consider hardware and software implementations of some of these phases and identify their overheads.

All GPUs, including Nyuzi, do rasterization in roughly the same way, dividing triangles into square or rectangular patches of pixels (e.g. 2x2, 4x4, etc.), which are then processed in parallel by shading hardware. Although Larrabee and Nyuzi are unusual in their choice to perform rasterization in software, our work can provide insight into aspects of hardware/software tradeoffs and scalability of shared rasterization hardware in large GPUs with many shading cores. Compared to the rasterizer algorithm published by Intel [2], a Nyuzi core with our best hardware rasterizer yields 26% higher throughput and uses at least 20% less energy.

II. Background on Nyuzi

Like Larrabee, Nyuzi takes a parallel tile-based approach to rendering. Mainstream GPUs, when performing fragment shading, will rasterize triangles to fragments, which are collected into batches (called warps or wavefronts in different architectures) and distributed by hardware to parallel shader cores. Nyuzi divides computation at the tile level such that the target rendering surface is divided nominally into 64x64 tiles. Each tile is dynamically assigned to a render thread, which rasterizes and shades all triangles and fragments in the tile before moving on to the next tile in its list.

Compared to mainstream GPUs, Nyuzi is much more CPU-like, borrowing features from Larrabee and Sun Niagara. Nyuzi executes its own RISC instruction set [6], and has a microarchitecture optimized for multithreading, multi-cycle floating point instructions, and single-cycle integer instructions. Nyuzi makes a number of interesting and unique microarchitectural choices, which its developer has expounded on in on-line documentation [7].

As a software-only architecture, Nyuzi has some performance challenges that differ from those of familiar GPUs that accelerate more functions in dedicated hardware. However, there are many key similarities that make Nyuzi a good platform to conduct experimentation on GPU architecture in general. Both architectural approaches (Larrabee-style SIMD vs. AMD- or Nvidia-style SIMT) have to deal with flow control divergence and masking at the triangle edge, which leads to vector lanes and/or hardware threads being disabled. Although there are some differences in the way these architectures map fragments to threads for graphics workloads, these differences are largely superficial and hidden by the compiler. We have compiled some algorithms for both Nyuzi and AMD Southern Islands (Radeon HD 7000 series), and the structure of the machine code for both is almost identical.

Nyuzi is also like many mobile GPUs in that it performs tile-based rendering. One major difference is that while many mobile GPUs render one tile at a time (processing fragments and triangles in parallel but tiles serially), Nyuzi processes multiple tiles in parallel (processing multiple fragments in parallel as vectors but triangles serially within each thread).

Despite some differences from mainstream GPUs, Nyuzi's greatest advantage is that it is the only fully-functional, fully-synthesizable, and complete open source GPU[1]. GPGPUsim [1] is very good for conducting cycle-accurate simulations of Nvidia GPUs, but it cannot be synthesized to gates and used in "real time." MIAOW [5] is a synthesizable partial implementation of the AMD Southern Islands ISA, but it focuses on high performance computing and lacks video and memory controllers, among other graphics components. Not only is Nyuzi itself open source, but it also comes with a functional simulator, synthesizable RTL (which we simulate using Verilator [19]), a complete LLVM compiler toolchain, libc, OS emulation (libos), and rendering library (librender). All of our experiments in this paper are conducted at the RTL level using Verilator.

III. Nyuzi Microarchitecture and ISA

Nyuzi's microarchitecture is shown in Figure 1 [7]. Instruction fetch is split over two pipeline stages, one for instruction cache (Icache) tag compare, and the other for Icache data lookup. For a Nyuzi core configured for four threads, there are four program counters (PCs), and the fetch select logic iterates over the PCs in round-robin fashion, skipping those for blocked threads. An Icache miss will result in a bubble in the front-end. The next stage is instruction decode; this is done before the instruction queues to determine instruction latencies and identify write-back conflicts, which will be used by the thread select logic to choose the next instruction to issue. Following decode is a set of queues, one for each thread. These queues help avoid stalls in the execution back-end due to Icache misses. The thread select logic implements a scoreboard and chooses an instruction from a non-empty queue whose operands are available and whose write-back will not result in a resource conflict. This is followed by the register file. Next are three parallel pipelines: A single stage integer ALU, a 5-stage floating point ALU, and a 2-stage load/store unit. All of these ALUs support both scalar and 16-wide vector instructions. This is followed by a shared write-back path whose use is scheduled by the thread select logic. In earlier revisions, Nyuzi supported result forwarding, but this was found to be a critical path and removed; to make up for the performance loss, a scoreboard was implemented to allow independent instructions in the same thread to overlap their execution. Nyuzi supports precise exceptions and has fully coherent caches.

Nyuzi implements a 3-operand register-to-register RISC instruction set architecture (ISA) [6], which supports mixed

[1]The Verilog source code for Nyuzi used to prepare the results in this paper can be found at `https://github.com/jbush001/NyuziProcessor` in the `ispass-2016` branch.

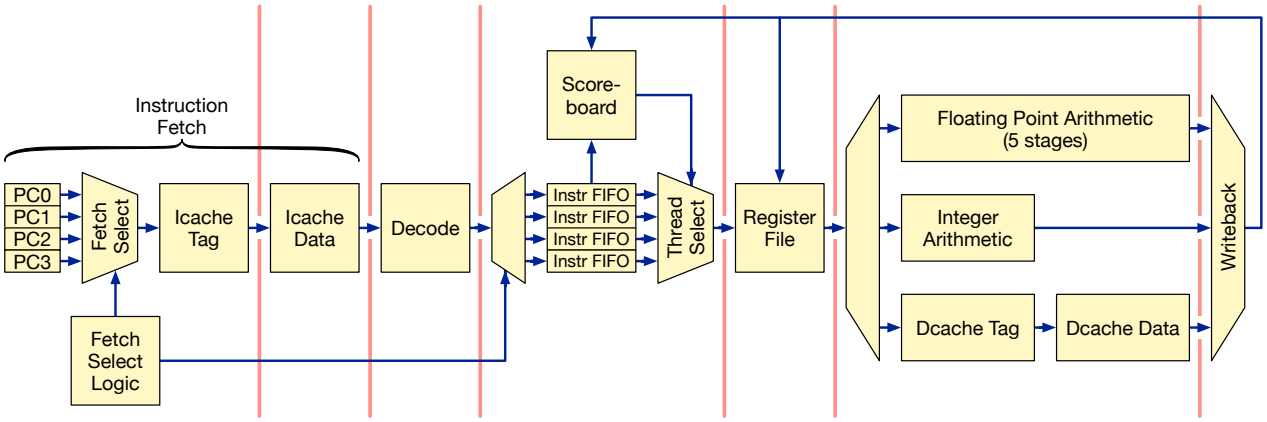

Fig. 1. Nyuzi processor core microarchitecture and execution pipeline

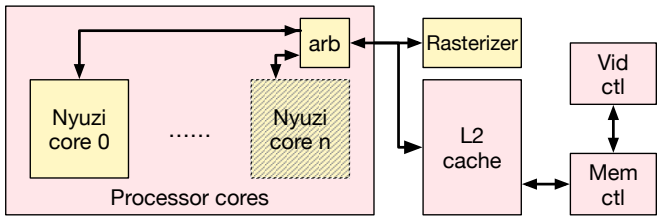

Fig. 2. Nyuzi system-on-chip, showing integration of the Rasterizer coprocessor.

vector and scalar operands, where scalar operands are sent down all vector lanes in the ALU. Additionally, vector instructions support masks, allowing destination registers to be selectively written on a per-lane basis. Loads and stores support vector scatter/gather, and supported word sizes are 32, 16, and 8 bits.

A diagram of a Nyuzi system-on-chip is shown in Figure 2. One or more Nyuzi cores (each with their own L1 caches) are bundled together in a core group. Cores in the core group communicate with the memory system and I/O systems through arbiters. The memory system is composed of the large shared L2 cache and memory (DRAM) controller. The video hardware also connects to the memory controller. The Rasterizer coprocessor is connected as an I/O device to the processor cores through the I/O arbiter in the core group.

IV. Experimental Design

For this paper, we construct, simulate, and synthesize 11 different hardware rasterizer implementations. A rasterizer is integrated into Nyuzi as a coprocessor that receives triangle and tile coordinates from software and returns a stream of 4x4 pixel masks to software. Six of the designs represent logic area vs. scalability tradeoffs; smaller designs require more cycles to perform setup and compute pixel masks and therefore cannot scale to support as many concurrent triangles

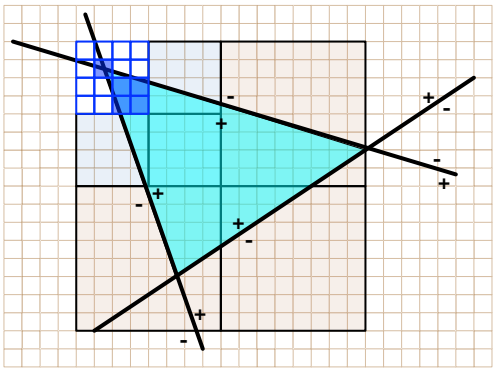

Fig. 3. Triangles are rasterized by intersecting edges with 4x4 patches of pixels. Pixels are considered "inside" if all three line edge formulas ($Ax - By + C$) yield zero or greater.

being rasterized. Another five designs represent hardware vs. software tradeoffs; phases of setup and patch computation are moved from hardware into software, and we learn the relative overhead of performing all, some, or none of the rasterization in hardware. Rasterization is divided into the following phases:

- Setup – Compute edge parameters from triangle vertices.
- Iteration – Traversing a tile or the intersection of a triangle bounding box and the tile.
- Mask computation – Intersection of 4x4 patches with triangle edges.
- Rejection – Elimination of empty patches (when all mask bits zero).
- Clipping – Rejection of patches outside of a window.

A. Algorithms

Nyuzi's default rasterizer is based on the algorithm described by Abrash at Intel [2]. Each triangle edge is described in terms of the formula $Ax - By + C = 0$, where A, B, and C are computed from a pair of vertex coordinates, according to

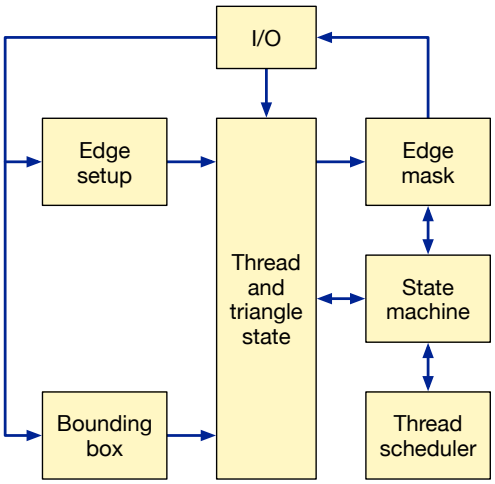

Fig. 4. Block diagram of iterative (scanning) hardware rasterizer implementations.

Equations 1, 2, and 3. All calculations are done with integers in these implementations. An arbitrary pixel is considered to be "inside" of the triangle if $Ax - By + C \geq 0$. This is illustrated in Figure 3. 64x64 tiles are broken into 16x16 subtiles, and it is determined which ones lie entirely outside of the triangle (trivial reject), which lie entirely inside the triangle (trivial accept), and those that must be decomposed further. This recurses down to the level of 4x4 patches, which are represented by 16-bit vector masks applied to vector instructions that perform shading. So that abutting triangles do not double-paint pixels, a bias of 1 is subtracted from edges that are "top" or "left" [14].

$$A = Y_2 - Y_1 \tag{1}$$

$$B = X_2 - X_1 \tag{2}$$

$$C = BY_1 - AX_1 \tag{3}$$

This recursive algorithm is very efficient for triangles with large areas that are trivially accepted or rejected. However, the recursion also introduces too much overhead for very small triangles. We therefore developed purely iterative scanning algorithms. Either the whole tile or the intersection of the bounding box and the tile is scanned, and 16-bit masks are computed for each 4x4 patch in the region. This has the disadvantage of expensive calculations for every patch in the region but almost always has much less work to do for small triangles, compared to the recursive algorithm.

B. Hardware rasterizer architecture

The general architecture of the hardware rasterizers is shown in Figure 4. To initiate rasterization, processor cores perform memory-mapped I/O writes to the triangle state memory, specifying triangle corner coordinates. The last coordinate write automatically instructs the thread scheduler and state machine to schedule setup for the new triangle. The thread scheduler iterates over all active triangles, performing one processing step per cycle. Setup involves computing triangle edge parameters (A, B, and C) and the triangle bounding box. For each patch inside the bounding box, the state machine takes one cycle to intersect the current patch with each edge (a total of three cycles), one cycle to check the validity of the combined mask, and one cycle to advance to the next patch in the bounding box. For empty patches, the state machine advances to the next patch automatically; for non-empty patches, the state machine suspends that thread and waits for software to retrieve the patch.

C. Scalability

Below, we present the evolutionary history of our rasterizer designs. Each of the designs in this subsection trades off logic area for scalability. As we share rasterizer hardware among multiple threads, make the design synthesizable, and optimize for area, these design decisions impact the number of cycles required to perform each rasterization phase. We simulate Nyuzi with only four threads, so in fact, all of these designs improve performance over software about the same. What changes from one design to the next is scalability in terms of the theoretical number of Nyuzi threads that could utilize the rasterizer hardware simultaneously.

What we expect to learn from this is how shared rasterizers scale in GPUs in general. In mainstream GPUs, there may be multiple rasterizers, but each one is capable of rasterizing only a single triangle at a time, shipping patches of the same triangle off to different fragment shaders. But at what point does a rasterizer's compute bandwidth saturate, requiring the addition of another rasterizer in order keep up with the flow of triangles? From these experiments, we can make projections as to the saturation point of any rasterizer that is shared, either serially or concurrently.

L1: The recursive rasterizer algorithm (as described by Abrash at Intel [2]), implemented entirely in software. This implementation serves as the comparison baseline for all other experiments. Note that although Abrash referred to this algorithm as the "Larrabee rasterizer," the algorithm ultimately used for Larrabee is much more complex, with many special cases for improved performance; our L1 algorithm does not include these enhancements.

R1: The recursive rasterizer algorithm implemented in hardware as a pushdown finite state machine. Setup takes one cycle, while each mask computation takes at least three cycles due to transitions between levels of recursion. This represents an upper bound on the performance of this rasterization strategy and is not necessarily synthesizable. We initially specified enhancements to this approach (other R# designs), but chose not to implement them due to the clear superiority

TABLE I
TEST AND BENCHMARK WORKLOADS USED TO EVALUATE IMPROVED RASTERIZERS. TRIANGLE COUNT INCLUDES TRIANGLES MULTIPLE TIMES WHEN THEY OVERLAP MULTIPLE TILES. PATCH AND PIXEL NUMBERS COUNT PIXELS THAT ARE CULLED BY DEPTH OCCLUSION OR PAINTED MULTIPLE TIMES.

Conformance Tests		Triangles	4x4 Patches	Pixels
blend	One triangle is drawn on top of another with alpha blending.	83	14232	221184
clip	Frustum clipping is applied to a textured scene.	142	21151	327696
depth	Two triangles intersect in Z space, making only part of each visible.	82	14232	221184
fill	The whole drawing surface is filled white by two triangles.	97	20561	323207
mipmap	Texture mapping is applied to a receding plane, sampling different levels of detail.	35	4732	73152
texture	Texture-mapped cube.	47	3616	51397
triangle	A single triangle is drawn, with colors interpolated between vertices.	47	7955	124128
Benchmarks				
teapot	The standard teapot benchmark.	1072	4769	31317
sponza	Dabrovik's Sponza Atrium.	16144	76892	552444
quake	A scene from the Quake game.	12803	75805	684806

of H2 below, which is more compact and more scalable. (H2 outperforms R1 by more than 7×.)

H1: For each triangle, an entire 64x64 tile is scanned in 4x4 patches from top to bottom. This is meant to approach an upper bound on patch throughput, so setup takes one cycle, and each mask takes one cycle to compute. When a blank patch is computed, the state machine automatically advances to the next patch. This approach will take 257 cycles to reject all patches in a tile that does not intersect the programmed triangle. As an upper bound, this is not meant to be synthesizable, and each rasterizer supports only one concurrent triangle, so multiple instances are required to support multiple Nyuzi threads.

H2: Based on H1, but scanning is confined to the intersection of the tile and the bounding box of the triangle. H2 supercedes H1 as the upper bound on throughput and scalability. H2 was an algorithmic proof-of-concept that relied too heavily on unsynthesizable Verilog behavioral code.

H3: Based on H2 but rewritten to be synthesizable. Setup takes 8 cycles (2 to compute the bounding box, 6 to compute edge parameters), and each patch takes 5 cycles to compute (3 to compute masks against each edge, 1 for rejection, 1 to advance coordinates).

H4: Based on H3 but modified to share one set of compute resources among multiple concurrent thread contexts. Contexts for concurrent triangles are stored in RAM blocks. This design, when synthesized, requires a longer clock period than a Nyuzi core.

H5: Based on H4 but optimized for speed so that the clock period is the same as a Nyuzi core. Setup takes 14 cycles, and each patch requires 5 cycles. See Table II for states.

H6: Based on H3, but modified with a resource lock. Unlike H4 and H5, which are shared by multiple threads concurrently, H6 maintains only a single context and is shared by multiple threads *serially*. To use H6, a thread will lock the rasterizer, program it with a triangle, download all of the patches to an array, unlock the rasterizer, and then shade all of the patches. This design is intended to be more similar to how other GPUs share a rasterizer among multiple triangles.

TABLE II
STATES FOR THE H5 HARDWARE RASTERIZER IMPLEMENTATION. FOR EACH TRIANGLE, THE STATE MACHINE STARTS IN 31, WHICH IS THE IDLE STATE.

0-3	Compute A, B, C for triangle edge 1
4-7	Compute A, B, C for triangle edge 2
8-11	Compute A, B, C for triangle edge 3
12	Compute mask for edge 1
13	Compute mask for edge 2
14	Compute mask for edge 3
15	On blank mask, go to 16; else wait for dequeue then go to 16
16	Advance coordinates; if done go to 31, else 12
29	Compute bounding box top and bottom
30	Compute bounding box left and right, then go to 0
31	Idle; go to 29 on start

D. Hardware/Software Tradeoffs

Here, we conduct a set of experiments to identify the software overheads of important phases in performing rasterization. How is performance affected if each phase is done sequentially in software or in parallel in a coprocessor? Experimental results should serve as explanations for the performance benefits we observe from hardware acceleration.

When moving some functions into software, we will compare to pure hardware rasterizer implementations H2 and H5. For software/hardware tradeoffs, we also think of these as **S1** and **S2**, respectively. These are also compared against **L1**, described above [2].

S3: Based on S2/H5, but setup is moved into software.

S4: Based on S3, but blank patch rejection is moved into software, rather than having hardware automatically advance to the next non-empty patch. All else is done in hardware. This is just to demonstrate the time savings of having a trivial amount of hardware to do auto-reject.

S5: Based on S3, but software rejects patches that fall outside of a clipping rectangle.

S6: The scanning algorithm implemented by H2 through H5 is instead implemented entirely in software.

V. Experimental Methodology

There are 10 workloads we use to evaluate our hardware and software rasterizers, as shown in Table I. Seven of them are merely conformance tests, while the remaining three are "real" graphics benchmarks with numerous triangles of various sizes. The benchmarks all benefit across the board about the same from hardware acceleration, while the conformance tests are mostly unaffected. We therefore do not report experimental results on the conformance tests, although some are informative to some experiments.

To synthesize Nyuzi for FPGA, we used Quartus, targeting an Altera Cyclone IV E (EP4CE115F29C7). On this low-end device, the Nyuzi processor core and other components (e.g. L2 cache, video, DRAM controller) ran at a maximum frequency of about 60MHz and used 92,186 logic elements (59,154 combinatorial only, 6,419 register only, 26,613 both), about 81% of the device's total capacity. For ASIC synthesis, we used Synopsys Design Compiler [11] and the Nangate 45nm transistor technology model [15]. In this technology, we find that the Nyuzi processor core can achieve about 570MHz.

RTL-level simulations of the Nyuzi Verilog code [8] and our modifications were performed using Verilator [19]. Since Verilator does not support undefined values, all memory elements are initialized to random numbers, and this results in a small degree or nondeterminism in runtimes. We ran teapot 400 times and computed a coefficient of variation (σ/μ) in runtime of 0.16%.

The circuit area of a Nyuzi core is dominated by RAM blocks. The Nangate library does not support SRAM blocks, so Synopsys must synthesize RAMs out of flipflops and multiplexers. In our experimental results, we synthesized Nyuzi without RAMs and added RAM area and power numbers from CACTI [20] instead.

For power analysis with Synopsys, we used the default switching activity. According to the user's guide, "The default value of `power_default_static_probability` variable is 0.5 and the default value of `power_default_toggle_rate` variable is 0.1."

VI. Experimental Results

In this section, we present results on scalability, software/hardware tradeoffs, performance, energy, and circuit synthesis.

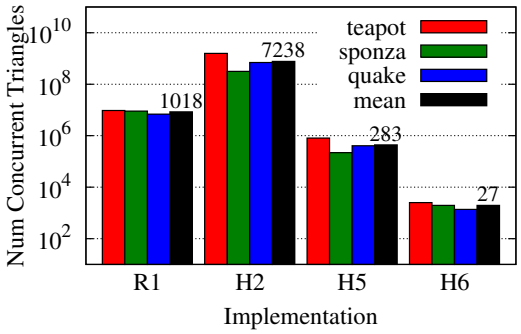

Fig. 5. Scalability projections for each hardware implementation, along with averages, indicating the number of triangles each implementation should be able to rasterize concurrently.

A. Scalability

Figure 5 shows scalability projections for selected hardware designs. These numbers estimate the maximum number of triangles that could be rasterized concurrently, if we were to scale Nyuzi up to execute that many rendering threads. These are computed from Equation 4. Given the triangle submission rate ($Cycles_{trian}$), the number of triangles (N_{trian}), the number of 4x4 patches (N_{patch}), the number of triangle setup cycles ($Cycles_{setup} = 14$ for H5), and the number of patch computation cycles ($Cycles_{patch} = 5$ for H5), scalability can also be calculated from Equation 5.

$$Scalability = \frac{Cycles_{scene}}{Cycles_{rasterizer-busy}} \quad (4)$$

$$Scalability = \frac{N_{trian}\,Cycles_{trian}}{N_{trian}\,Cycles_{setup} + N_{patch}\,Cycles_{patch}} \quad (5)$$

We first implemented R1 and H1. R1 could scale to hundreds of threads. H1 scaled even better for the conformance tests but much more poorly for the benchmarks. This is because H1 would scan an entire tile for every triangle, wasting a lot of time for very small triangles. Thus, we developed H2, which confined scanning to the bounding box of each triangle. As we can infer, the utilization of H2 is very low, suggesting that a single rasterizer could be shared among many threads in a much larger GPU.

For really big triangles, the recursive algorithm (R1) can be a bit faster than the sweeping algorithm (H1). The fill test paints two screen-sized triangles, one covering the lower left, and the other the upper right. This means the bounding box of each is the whole screen, although a triangle is not included in any non-intersecting tiles. Where tiles are completely filled, whole tiles can be trivially accepted, which is handled by a fast iterative loop. Where tiles are partially filled, the recursive algorithm can skip a lot of the tile hierarchically. On the other hand, the sweeping algorithm has to iterate through most or all of a partially-filled tile in this case. The conclusion, which applies as well to other GPUs as it does to

Nyuzi, is that the sweeping algorithm is a better choice for a hardware accelerator, because it can rasterize more triangles in a given amount of time for real graphics workloads.

H2 was a non-synthesizable prototype, so we wrote a synthesizable version called H3. Setup was designed to take 7 cycles, requiring only two 32-bit multiplier blocks. Mask computation was spread over three cycles, one per triangle edge, plus two more for reject/accept and coordinate increment. Note that H2 and H3 require one dedicated rasterizer for each Nyuzi thread.

H2 and H3 would spend most of their time idle and underutilized. To support time-sharing, we developed H4, which can switch between multiple triangles whose context are stored in RAMs. Since we ran only four Nyuzi threads, H4 numbers are almost exactly 1/4 that of H3. A single H4 rasterizer can process hundreds of triangles concurrently, with a mean of 356 across all tests and benchmarks.

As can be seen in Table IV, H4's critical path was too slow. Adding H4 to Nyuzi would have required a lower clock speed, slowing down all other computations. We thus developed H5, which we optimized to have the same clock speed on the Altera FPGA as a Nyuzi core. This required increasing the setup time from 7 cycles to 14 cycles, but this did not affect scalability significantly, with a mean projection of 283 concurrent triangles.

Finally, we developed H6 (equivalent to one instance of H3 plus a small amount of logic for the lock) in an attempt to make the Nyuzi hardware rasterizer be shared similarly to how one is shared on a regular GPU. Instead of multiple threads sharing it concurrently, they share it serially by locking the resource, processing the whole triangle, unlocking the resource, and then shading pixels. Unfortunately, two problems made this perform very poorly. One is that Nyuzi's uncached peripheral bus is low bandwidth. However, we found other rasterizers to be largely insensitive to the bus throughput. Even if bus throughput were improved, the second and more substantial problem is that threads competing for the rasterizer waste excessive time spinning on the lock. Other rasterizers perform better due to greater concurrency in both rasterization and other rendering functions. The result for H6 is slower rendering and poor scalability. For the benchmarks, H6 averages only 23% faster than L1, while it averages 3.4% worse for the conformance tests.

B. Software/hardware tradeoffs

Figure 6 shows total runtimes (in clock cycles) of a 4-thread Nyuzi core with different hardware/software tradeoffs for rasterization. For the benchmarks, the recursive algorithm (L1) requires about 20% more runtime compared full hardware implementations (e.g. H5), which suggest that software rasterization can have a substantial amount of overhead.

TABLE III
FOR BENCHMARKS, AVERAGE NUMBER OF CYCLES PER TRIANGLE, 4X4 PATCH, AND PIXEL.

		teapot	sponza	quake	geo.mean	ratio
cycles/ triangle	L1	10099	10955	9993	10340	1.000
	H5	7968	8742	7921	8202	0.793
	S6	8245	9334	8329	8622	0.834
cycles/ 4x4 patch	L1	2270	2300	1688	2066	1.000
	H5	1791	1835	1338	1638	0.793
	S6	1853	1960	1407	1722	0.834
cycles/ pixel	L1	345.7	320.1	186.8	274.5	1.000
	H5	272.7	255.5	148.1	217.7	0.793
	S6	282.2	272.8	155.7	228.9	0.834

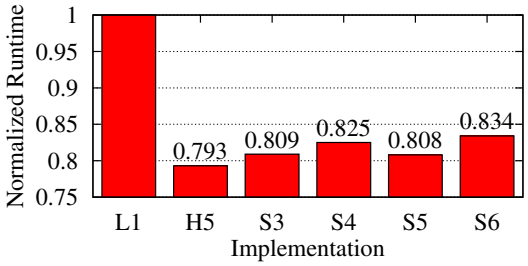

Fig. 6. Relative runtimes of hardware/software tradeoff implementations, averaged over benchmarks, normalized to baseline recursive algorithm (L1).

Moving setup from hardware (S2/H5) into software (S3) adds about 2% additional runtime, for all benchmarks. (There was no measurable effect on the conformance tests.) Additionally moving rejection into software (S4) adds about 2% more. Doing setup and clipping (but not rejection) in software (S5) has essentially the same performance as S3.

Moving just about everything into software did not explain the 20% performance gap between L1 and the full hardware implementations. To understand this more clearly, we implemented S6, which is our scanning algorithm, but entirely in software. For benchmarks, L1 averages 20% more runtime than S6, while S6 requires 5.1% more runtime than S2/H5. We wondered why Larrabee did not use a scanning algorithm like ours, so we contacted the original developers of the

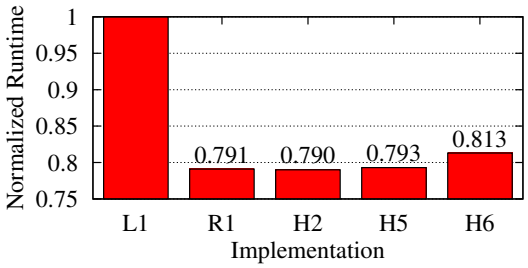

Fig. 7. Relative runtimes using hardware implementations, averaged over benchmarks, normalized to baseline recursive algorithm (L1).

TABLE IV
AREA, DELAY, AND POWER FOR ASIC (NANGATE 45NM) AND ALTERA CYCLONE FPGA: NYUZI CORE VS. VARIOUS RASTERIZER DESIGNS.

	Nangate 45nm ASIC			Altera FPGA		
	Area (μm^2)	Delay	Power	LE's	FF's	Delay
Nyuzi core	1800523	1.75ns	1.71 W			
H3 (total)	9967	1.92ns	3.15mW	1867	451	16.7ns
H4 (total)	20945	2.09ns	7.80mW	3158	1817	19.1ns
edge mask	1163			772		
edge setup	2609			269		
bound box	160			90		
H5 (total)	20441	1.75ns	7.21mW	3463	1889	15.6ns
edge mask	1163			773		
edge setup	1625			398		
bound box	170			90		
H5 8 threads	34579	1.76ns	13.0mW			
H5 16 thr	62076	1.87ns	27.7mW			
H5 32 thr	117945	1.99ns	61.6mW			
H5 64 thr	228565	2.04ns	128mW			

TABLE V
45NM ASIC SYNTHESIS RESULTS FOR NYUZI PROCESSOR CORE (FROM SYNOPSYS/NANGATE) AND RAM COMPONENTS (FROM CACTI).

	unit μm^2	unit Watts	units	total μm^2	total Watts
Core w/o RAMs	294681	0.1020	1	294681	0.1020
L1 LRU	271	0.0005	2	543	0.0010
L1 I&D caches	605663	0.6747	2	1211326	1.3494
Instruction FIFO	16037	0.0276	4	64147	0.1105
Register file	13519	0.0088	17	229826	0.1504
Total				**1800523**	**1.7133**

Larrabee rasterizer. Tom Forsyth told us that, in fact, they had numerous special cases for small triangles. For instance, sliver triangles that fit into Nx1, 1xN, Nx2, and 2xN regions of 4x4 patches were handled specially. (We expected S6 to be slow because of all the patches it would have to reject so many blank patches, but S4 shows that software rejection does not have high overhead.)

C. Performance

Figure 7 shows performance results from running tests and benchmarks on various hardware and software rasterizer implementations. Compared to L1 (the recursive algorithm described in [2]), most of the full hardware implementations (H1 through H5) reduce execution time by about 21% (26% faster). L1 is very efficient for large triangles, so hardware acceleration does not benefit any of the conformance tests.

On 45nm ASIC, Nyuzi can run at over 570MHz, which would mean a peak baseline (L1) throughput of 2,076,503 pixels/sec per core (on average for the benchmarks). Switching from L1 to S6, that increases to 2,490,170 pixels/sec per core (20% faster than L1). Switching to H5, that increases to 2,618,282 pixels/sec per core (26% faster than L1, 5.1% faster than S6). For the conformance tests, average throughput is 8,934,169 pixels/sec per core for L1; S6 and H5 improve performance by less than 1%. These numbers are inferred from Table III (and corresponding bars in Figure 6), which presents submission rates for triangles and corresponding rates for patches and pixels for each of the conformance tests and benchmarks.

The conformance tests have a much higher throughput in part because they make more efficient use of the vector ALU. On average, conformance tests paint 15.4 pixels/patch, while the benchmarks only average 7.6 pixels/patch, which can be inferred from Table I.

D. Synthesis Results

Table IV shows synthesis results for a number of our hardware rasterizer designs, with area, delay, and power estimates for both ASIC and FPGA. We implemented the rasterizers with submodules for computing bounding box, performing setup, and computing masks. Those turn out to take up a very small amount of the area, which is dominated instead by the RAMs (triangle context) and the state machine logic. To put rasterizer circuit area and power into context, Table V breaks down the Nyuzi core into logic and RAM components. RAM component metrics are provided by CACTI 5.3 [20].

Although sharing the rasterizer (H4) requires a lot more area, H4 is still much smaller than four copies of H3. When optimizing for clock frequency (H5), we actually reduced the number of multipliers to 1, resulting in a reduction in logic area but a slight increase in memory area. Going beyond four threads, memory area increases about linearly, while logic area increases sublinearly.

From section VI-B, we saw that H5 is only about 5.1% faster than the best software algorithm. It is important to note that this gap will widen substantially as other phases of Nyuzi rendering (e.g. texturing) are optimized in performance or even accelerated themselves in dedicated hardware. Compared to a Nyuzi core, the proposed rasterizer is tiny, requiring 1/88 as much area and 1/237 as much power in 45nm, making the rasterizer a cost-effective addition.

At the bottom of Table IV, we consider the impact of scaling H5 to larger numbers of contexts. Nangate [15] does not include optimized SRAM blocks, so RAM areas are inflated; however, these synthesis results still provide an indication of how area, performance, and power scale with the number of supported contexts. The H5 designs with more threads use a lot more power, because Synopsys is attempting (but failing) to meet the timing constraint (which we did not adjust) by using logic gates with wider transistors and/or lower threshold voltage. Changing the timing constraint would have reduced power, but we consider it unacceptable to allow the rasterizer to become the circuit critical path. To meet timing with larger numbers of contexts, some additional

pipelining will be required, but this will not affect throughput or scalability to a large degree.

E. Energy

As can be seen from Table IV, a single 4-thread Nyuzi core is about 88× larger than an H5 rasterizer and uses about 237× as much power. The reported area includes L1 caches but excludes the L2, which would be shared among multiple cores. Since we have not performed any gate-level simulations, we do not have a good basis for estimating typical switching activity in Nyuzi and the rasterizers; as a result, the default switching activity used by Synopsys overestimates rasterizer power, even more so than for Nyuzi, making the power ratio conservative. For a cycle time of $\tau = 1.75\text{ns}$ (571MHz), a Rasterizer power consumption of $p = 7.21\text{mW}$, and n Nyuzi cores, Equation 6 applies to the software-only implementations, while Equation 7 applies to an implementation that uses an H5 rasterizer. Because cycle count does not quite scale linearly with the number of cores, these are estimates.

$$E_{\text{sw}} = \mathit{Cycles}_{\text{sw}}/n \cdot \tau \cdot 237np \tag{6}$$

$$E_{\text{hw}} = \mathit{Cycles}_{\text{hw}}/n \cdot \tau \cdot (237n + 1)p \tag{7}$$

With a single core, L1 (recursive software implementation) requires $E_{\text{sw}} = 187.11\text{mJ}$ for the average benchmark runtime, S6 (iterative software implementation) requires $E_{\text{sw}} = 156.02\text{mJ}$ (16.6% less), and H5 (full hardware rasterization) requires $E_{\text{hw}} = 149.04\text{mJ}$ (20.3% less than L1).

Estimating for 10 cores, L1 requires $E_{\text{sw}} = 187.11\text{mJ}$, S6 requires $E_{\text{sw}} = 156.02\text{mJ}$ (16.6% less), and H5 requires $E_{\text{hw}} = 148.48\text{mJ}$ (20.6% less than L1). Although S6 improves energy efficiency substantially, hardware acceleration reduces energy even further, because it has higher performance and comparatively negligible power overhead.

F. Discussion

Key takeaway points from our experiments are as follows. The recursive Larrabee rasterizer algorithm, while mathematically elegant, is not computationally efficient, favoring an iterative approach instead. However, it is important to ensure that scanning for 4x4 patches be confined to the bounding box of a triangle or else a lot of time will be wasted rejecting empty patches. The scanning algorithm in software substantially boosts performance, but a hardware implementation is still more than 5% faster.

At the time Intel developed the Larrabee hardware, they reasoned that hardware rasterization could be replaced by software rasterization in an economical way. Software rasterization instead turned out to be much more challenging than anticipated. To compensate for the seemingly-sensible absence of a tiny hardware component, Intel software engineers had to develop a complex and elaborate software algorithm with numerous special cases.

Considering that the amount of logic area is tiny compared to even a single Nyuzi core, the benefits substantially outweigh the costs. Any rasterizer will spend very few cycles on any given triangle, compared to all other rendering computation, allowing one rasterizer to support hundreds of rendering threads. As we add processor threads, the relative area cost of the rasterizer hardware diminishes to almost nothing, and the energy benefits increase, since a single rasterizer is shared among multiple processor cores. The main benefits to using the hardware rasterizer come from the ability to compute a critical rendering phase in dedicated hardware in parallel to other computation. Hardware rasterization is usually taken for granted, but our analyses highlight why rasterization should be out of the critical path for all GPU types. There are many other rendering phases that themselves could be sped up by more efficient software and/or hardware acceleration, which we will leave to future work.

VII. Related Work

This section presents related work on rasterization and other open source GPUs.

A. Rasterization

The Nyuzi architecture is inspired by Intel's Larrabee architecture, so by default, its software utilizes the "Larrabee rasterizer" algorithm published by Intel [2]. Rasterization in a GPU is the process of converting a polygon specification (i.e. vertex coordinates) into pixels. Modern GPUs generally expect polygons to be decomposed into triangles, with some limited support for quads, although some work has been done in the past on rasterizing more general polygons [3]. Other rasterization algorithms exist, such as serpentine algorithms [17].

Early graphics accelerators employed a simple row scan algorithm that walked triangle edges vertically, generating horizontal spans, which are then decomposed into pixels [16]. This approach works well for wide triangles, but thin triangles, particularly very vertical slivers, will encounter poor memory bandwidth due to DRAM row misses.

Thus, GPU designers have switched to using tile-based memory organization (not to be confused with tile-based rendering) [13], where DRAM rows are made to correspond to rectangular blocks of pixels. Furthermore, modern GPUs have also switched to patch-based rasterization, where triangles are decomposed into streams of square (e.g. 2x2 or 4x4) blocks of pixels. Square or rectangular patches are important for avoiding warp divergence, because pixels that are closer to each other are more likely to require exactly the same sequence of instructions in a programmable shader.

B. Simulators and open source GPUs

Nyuzi is an updated version of the Nyami open source GPU. A recent paper [9] covered the Nyami/Nyuzi microarchitecture and compiler tools, along with experiments on instruction scheduling, load/store queueing, cache associativity, number of threads per core, and preliminary work on rasterization. As a synthesizable design, Nyuzi is useful as a GPU model and cycle-accurate GPU simulator. To conduct our research, Nyuzi was the best choice, since it allows us to study GPU-related problems at the RTL level and below, and Nyuzi emphasizes graphics applications.

For other kinds of GPU research, there are some alternatives. Simulator projects include GPGPU-sim [1] (a software cycle-accurate simulator of Nvidia GPUs), Guppy [4] (a modified LEON3), ATTILA [12] (a cycle-accurate GPU simulator focusing on graphics), Barra [10] (an array of PowerPC cores), and Multi2sim [21] (an AMD GPU simulator).

A group from University of Wisconsin-Madison developed MIAOW [5]. MIAOW's long-term goal is to emulate a subset of the AMD Southern Islands GPU instruction set, with a focus on high performance computing. Currently, enough logic exists for it to simulate the compute elements of a GPU, and those parts are fully synthesizable. MIAOW has the advantage of being able to run unmodified OpenCL applications compiled for Southern Islands.

VIII. Conclusions

Prior research observed that the recursive software rasterizer algorithm published by Abrash at Intel [2] imposed about a 20% performance penalty compared to performing rasterization in hardware. For this paper, we implemented 11 different hardware rasterizer implementations. We observe that a single rasterizer can scale comfortably to hundreds of Nyuzi threads, which is equivalent to thousands of fragments, while still maintaining the same performance benefit. We developed a very compact hardware rasterizer coprocessor that improves rendering performance by 26% and reduces energy by least a 20%. This, along with other optimizations and coprocessors, has the potential to bring Nyuzi closer to performance parity with other GPUs in terms of clock cycles.

We also conducted hardware/software tradeoff experiments to identify the overhead of different phases of rasterization. This lead to a better understanding of the inefficiencies of the recursive rasterizer as described in the original literature and a new, much more efficient software rasterizer for Nyuzi.

Rasterization is a critical process for any graphics workload, which has the potential to significantly impact energy, execution time, and/or circuit area of any GPU. Our experiments show that although software rasterizers can be made efficient, a hardware implementation will always be faster and more energy-efficient, with diminishingly small circuit area cost as processor core count increases.

References

[1] Tor M. Aamodt and Wilson W.L. Fung. GPGPU-Sim 3.x manual. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.

[2] Michael Abrash. Rasterization on Larrabee. https://software.intel.com/en-us/articles/rasterization-on-larrabee.

[3] BD Acland and Neil H Weste. The edge flag algorithm–a fill method for raster scan displays. *Computers, IEEE Transactions on*, 100(1):41–48, 1981.

[4] 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.

[5] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. Enabling GPGPU low-level hardware explorations with MIAOW–An open source RTL implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimization (ACM TACO)*, 2015.

[6] Jeff Bush. NyuziProcessor instruction set. https://github.com/jbush001/NyuziProcessor/wiki/Instruction-Set, 2015.

[7] Jeff Bush. NyuziProcessor microarchitecture. https://github.com/jbush001/NyuziProcessor/wiki/Microarchitecture, 2015.

[8] Jeff Bush. NyuziProcessor: Source code. https://github.com/jbush001/NyuziProcessor, 2015.

[9] Jeff Bush, Philip Dexter, Timothy N. Miller, and Aaron Carpenter. Nyami: A synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[10] 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 Telecommunication Systems*, pages 351–360, 2010.

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

[12] 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.

[13] Robert W. Frederickson and Andrew C. Goris. Graphics frame buffer with programmable tile size. https://www.google.com/patents/US5056044, 1990.

[14] F. Giesen. Triangle rasterization in practice. https://fgiesen.wordpress.com/2013/02/08/triangle-rasterization-in-practice/, 2013.

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

[16] Marc Olano and Trey Greer. Triangle scan conversion using 2D homogeneous coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 89–95. ACM, 1997.

[17] Juan Pineda. A parallel algorithm for polygon rasterization. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 17–20. ACM, 1988.

[18] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: A many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

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

[20] S Thoziyoor, N Muralimanohar, JH Ahn, and N Jouppi. CACTI 5.3. *HP Laboratories, Palo Alto, CA*, 2008.

[21] 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.