Efficient MIMD Architectures for High-Performance Ray Tracing

D. Kopta, J. Spjut, E. Brunvand, and A. Davis
University of Utah, School of Computing
{*dkopta,spjut,elb,ald*}*@cs.utah.edu*

***Abstract*— Ray tracing efficiently models complex illumination effects to improve visual realism in computer graphics. Typical modern GPUs use wide SIMD processing, and have achieved impressive performance for a variety of graphics processing including ray tracing. However, SIMD efficiency can be reduced due to the divergent branching and memory access patterns that are common in ray tracing codes. This paper explores an alternative approach using MIMD processing cores custom-designed for ray tracing. By relaxing the requirement that instruction paths be synchronized as in SIMD, caches and less frequently used area expensive functional units may be more effectively shared. Heavy resource sharing provides significant area savings while still maintaining a high MIMD issue rate from our numerous light-weight cores. This paper explores the design space of this architecture and compares performance to the best reported results for a GPU ray tracer and a parallel ray tracer using general purpose cores. We show an overall performance that is six to ten times higher in a similar die area.**

I. Introduction

Application specific architectures have proven to be particularly successful for accelerating computer graphics. Modern graphics processing units (GPUs) can process huge numbers of primitive objects (usually triangles) to render complex scenes. However, significant improvements to the visual quality of the rendered scenes may require a fundamental change in the graphics pipeline. Global optical effects that are the key to enhanced realism are not well supported by current rasterization-based GPUs. Ray Tracing [1]–[3] is a different algorithm that is very well suited to highly realistic rendering.

A number of researchers have leveraged SIMD processing to enhance the speed of ray tracing (e.g. [4]–[8]), but ray tracing's divergent branching and memory access patterns suggest an alternate approach may be beneficial. In this paper we consider a MIMD execution model to expose design options that are well-suited to the ray tracing workload. This is a fundamental shift from the current commercial GPU-based approach. We perform a detailed design space exploration using a custom cycle-accurate simulator and a detailed ray tracing application on a variety of standard benchmark scenes. We show that by having multiple independent thread processors customized for ray tracing, overall performance measured in millions of rays per second (MRPS) can be extremely high. We also show that the lack of ray synchrony can be exploited to reduce the complexity of the MIMD ray processor cores through sharing expensive but infrequently used functional units and multi-banked caches. The key is that this lack of synchrony results in reduced stalls from the shared resources and allows a high instruction issue rate to be sustained across multiple cores. Our architecture achieves performance per area (MRPS/mm^2) that is 6-10x higher on average than the best reported results for ray tracing on a commercial (SIMD) GPU or on a MIMD architecture that uses general-purpose cores as building blocks.

II. Background and Motivation

Interactive computer graphics today is dominated by extensions to Catmull's original Z-buffer rasterization algorithm [9]. These highly specialized graphics processors stream all image primitives through the rasterization pipeline using wide SIMD techniques and multiple parallel SIMD pipelines to boost performance. This type of parallelism is possible because all primitives in the scene can be processed independently. However this basic principle becomes a bottleneck for highly realistic images since it limits shading operations to per-triangle or per-pixel computations and does not allow direct computation of global effects such as shadows, transparency, reflections, refractions, or indirect illumination. Known techniques approximate these effects, but they can take significant effort and combining them is non-trivial.

Ray tracing is the major algorithmic alternative to rasterization. Ray tracing is relatively simple in principle: At each pixel a *primary ray* is sent from the viewer's eye point through the screen into the virtual set of objects and returns information about the closest object hit by that ray. The pixel is then colored (shaded) based on material properties, lighting and perhaps using texture lookups or a procedurally computed texture. Most modern ray tracers use a hierarchical acceleration structure to prune the scene prior to ray intersection tests. This results both in divergent branching behavior for that traversal and in non-coherent memory access to the scene database. From each point hit by a primary ray *secondary rays* can be traced to recursively determine shadows, reflections, refractions, soft shadows, diffuse inter-reflections, caustics, depth of field, and other global optical effects. Ray tracing naturally allows all of these effects to be combined cleanly. Even systems based on rasterization as the primary visibility method can benefit from the addition of ray tracing to provide these global effects.

Software ray tracers often collect groups of rays into packets to amortize the cost of cache misses and to use SIMD extensions that are available on the underlying architecture (e.g. [4]–[8]. The primary drawback to using wide SIMD for

ray tracing relates to *ray coherence.* If the rays are processed together in SIMD bundles, each ray in the bundle must follow exactly the same code path in order to keep the SIMD packet intact. Primary rays can be quite coherent, but this quickly breaks down for secondary rays which rarely have the same trajectory. The loss of ray coherence reduces consistency in control-flow branching which can limit SIMD efficiency [6], [10], [11].

Hardware acceleration for ray tracing can be very broadly categorized into SIMD approaches [11]–[17] and MIMD approaches [18]–[22]. In some sense the endpoints of this continuum of approaches are well represented by Aila et al.'s detailed exploration of ray tracing on existing SIMD GPU hardware [11] which represents the best GPU ray tracing performance reported to date, and Govindaraju et al.'s exploration of a MIMD approach called Copernicus using a tiled architecture of general-purpose cores [18]. The Aila paper describes a detailed analysis of how a ray tracer interacts with the NVIDIA GT200 architecture [23], specifically looking in detail at the efficiency of the 32-way SIMD used in that architecture, and how primary and secondary rays behave differently. The Govindaraju paper looks at how a more general purpose core, based on the Intel Core2, could be used in a tiled MIMD architecture to support ray tracing using their Razor ray tracing application [24]. Although these two approaches to improving ray tracing performance are extremely different, they end up with remarkably similar performance when scaled for performance/area (see Table IV). Mahesri et al proposed a custom MIMD architecture similar to ours, but for a broader range of benchmarks, and with fewer shared resources [25]. This requires them to consider more advanced architectural features such as thread synchronization and communication, out-of-order execution, and branch prediction. Given the unpredictable and "embarassingly parallel" nature of ray tracing, these features are either not required, or may be an inefficient use of area. We believe that simplifying the architecture, sharing functional units, and providing more customization specifically for ray tracing can dramatically increase performance/area. We therefore explore a MIMD architectural approach, but with lightweight thread processors that aggressively share resources. We compare with other designs using, as much as possible, the same benchmark scenes and the same shading computations.

III. Architectural Exploration Procedure

The main architectural challenge in the design of a ray tracing processor is to provide support for the many independent ray-threads that must be computed for each frame. Estimates for scenes with moderately high quality global lighting range from 4M to 40M rays/image [11], [18], [19]. At even a modest real-time frame rate of 30Hz. This means that performance in the range of 120 to 1200 MRPS will be desirable. Our approach is to optimize single-ray MIMD performance. This single-ray programming model loses some primary ray performance. However, it makes up for this by handling secondary rays nearly as efficiently as primary rays, which SIMD style ray tracers struggle with. In addition to providing high performance, this approach can also ease application development by reducing the need to orchestrate coherent ray bundles.

We analyze our architectural options using four standard ray tracing benchmark scenes, shown in Figure 1, that provide a representative range of performance characteristics, and were also reported in [11]. Our design space exploration is based on 128x128 resolution images with one primary ray and one shadow ray per pixel. This choice reduces simulation complexity to permit analysis of an increased number of architectural options. The low resolution will have the effect of reducing primary ray coherence, but with the beneficial side-effect of steering our exploration towards a configuration that is tailored to the important incoherent rays. However our final results are based on the same images, the same image sizes, the same mixture of rays, and the same shading computations as reported for the SIMD GPU [11]. Our overall figure of merit is performance per area, reported as MRPS/mm^2, and is compared with other designs for which area is either known or estimable.

Our overall architecture is similar to Copernicus [18] in that it consists of a MIMD collection of processors. However, it actually has more in common with the GT200 [23] GPU architecture in the sense that it consists of a number of small, optimized, in-order cores collected into a processing cluster that shares resources. Those processing clusters (Streaming Multiprocessors (SMs) for GT200, and Thread Multiprocessors (TMs) in our case) are then tiled on the chip with appropriate connections to chip-wide resources. The main difference is that our individual cores can each be executing a different thread rather than being tied together in wide SIMD "warps."

The lack of synchrony between ray threads reduces resource sharing conflicts between the cores and reduces the area and complexity of each core. With a shared multi-banked Icache, the cores quickly reach a point where they are each accessing a different bank. Shared functional unit conflicts can be similarly reduced. Given the appropriate mix of shared resources and low-latency Dcache accesses, we can sustain a high instruction issue rate without relying on latency hiding via thread context switching. This results in a different ratio of registers to functional resources for the cores in our TMs. The GPU approach involves sharing a number of thread states per core, only one of which can attempt to issue on each cycle. Our TMs contain one thread state per core, each of which can potentially issue an instruction to a private per-core FU or one of the shared FUs. We believe this single thread-state approach is a more efficient use of register resources.

We rely on asynchrony to sustain a high issue rate to our heavily shared resources, which enables simpler cores with reduced area, breaking the common wisdom that the SIMD approach is more area efficient than the MIMD model for ray tracing. It should be noted that this asynchrony is fine-grained and occurs only at the instruction level. Threads do not get significantly out of sync on the workload as a whole,

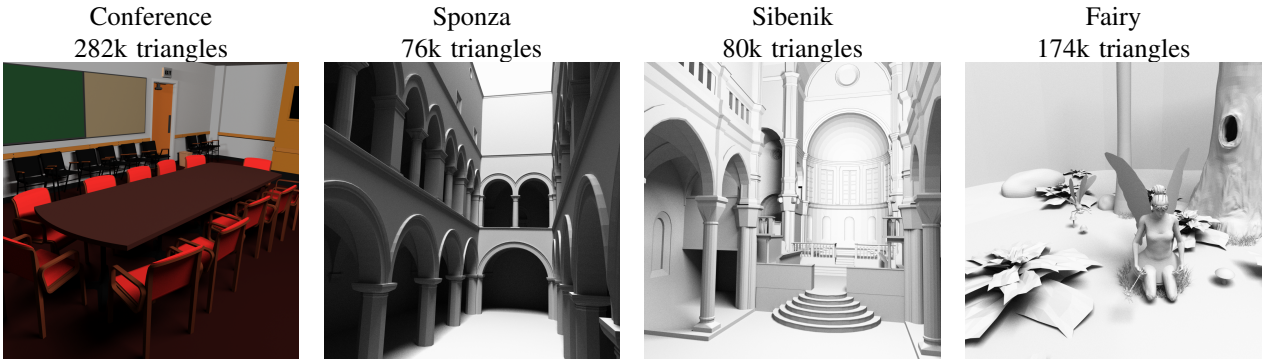


Fig. 1. Test scenes used to evaluate performance

thus maintaining coherent access to the scene data structure, and results in high cache hit rates.

Our exploration procedure defines an unrealistic, exhaustively-provisioned MIMD multiprocessor as a starting point. This serves as an upper bound on raw performance, but requires an unreasonable amount of chip area. We then explore various multi-banked Dcaches and sharing Icaches using Cacti v6.5 to provide area and speed estimates for the various configurations [26]. Next, we consider sharing large functional units which are not heavily used in order to reduce area with a minimal performance impact. Finally we explore a chip-wide configuration that uses shared L2 caches for a number of TMs.

Our simulation infrastructure includes a custom cycle-accurate simulator that runs our ray tracing test application. This application can be run as a simple ray tracer with ambient occlusion, or as a path tracer which enables more detailed global illumination effects using Monte-Carlo sampled Lambertian shading [3] which generates many more secondary rays. Our ray tracer supports fully programmable shading and texturing and uses a bounding volume hierarchy acceleration structure. In this work we use the same shading techniques as in [11], which do not include image-based texturing. Our test application was written in C++ and compiled with a custom compiler based on LLVM [27].

A. Thread Multiprocessor (TM) Design

Our baseline TM configuration is designed to provide an upper bound on the thread issue rate. Because we have more available details of their implementation our primary comparison is against the NVIDIA GTX285 [11] of the GT200 architecture family. The GT200 architecture operates on 32-thread SIMD "warps." The "SIMD efficiency" metric is defined in [11] to be the percentage of SIMD threads that perform computations. Note that some of these threads perform speculative branch decisions which may perform useless work, but this work is always counted as efficient. In our architecture the equivalent metric is thread issue rate. This is the average number of independent cores that can issue an instruction on each cycle. These instructions always perform useful work. The goal is to have thread issue rates as high or higher than the SIMD efficiency reported on highly optimized SIMD code. This implies an equal or greater level of parallelism, but with more flexibility and due to our unique architecture, less area.

We start with 32 cores in a TM to be comparable to the 32 thread warp in a GT200 SM. Each core processor has 128 registers, issues in order, and employs no branch prediction. To discover the maximum possible performance achievable, each initial core will contain all of the resources that it can possibly consume. In this configuration, the data caches are overly large (enough capacity to entirely fit the dataset for two of our test scenes, and unrealistically large for the others), with one bank per core. There is one functional unit (FU) of each type available for every core. Our ray tracing code footprint is relatively small, which is typical for most advanced interactive ray tracers (ignoring custom artistic material shaders) [2], [3] and is similar in size to the ray tracer evaluated in [11]. Hence the Icache configurations are relatively small and therefore fast enough to service two requests per cycle at 1GHz according to Cacti v6.5 [26], so 16 instruction caches are sufficient to service the 32 cores. This configuration provides an unrealistic best-case issue rate for a 32-core TM.

Table I shows the area of each functional component in a 65nm process, and the total area for a 32 core TM, sharing the multi-banked Dcache and the 16 single-banked Icaches. Memory area estimates are from Cacti v6.5[1]. Memory latency is also based on Cacti v6.5: 1 cycle to L1, 3 cycles to L2, and 300 cycles to main memory. FU area estimates are based on synthesized versions of the circuits using Synopsys DesignWare/Design Compiler and a commercial 65nm CMOS cell library. These functional unit area estimates are conservative as a custom-designed functional unit would certainly have smaller area. All cells are optimized by Design Compiler to run at 1GHz and multi-cycle cells are fully pipelined. The average core issue rate is 89% meaning that an average of 28.5 cores are able to issue on every cycle. The raw performance of this configuration is very good, but the area is huge. The next step is to reduce core resources to save area without sacrificing performance. With reduced

[1]We note that Cacti v6.5 has been specifically enhanced to provide more accurate size estimates than previous versions for relatively small caches of the type we are proposing.

TABLE I

FEATURE AREAS AND PERFORMANCE FOR THE BASELINE OVER-PROVISIONED 1GHZ 32-CORE TM CONFIGURATION. IN THIS CONFIGURATION EACH CORE HAS A COPY OF EVERY FUNCTIONAL UNIT.

Unit	Area (mm^2)	Cycles	Total Area (mm^2)
4MB Dcache (32 banks)		1	33.5
4KB Icaches	0.07	1	1.12
128x32 RF	0.019	1	0.61
FP InvSqrt	0.11	16	3.61
Int Multiply	0.012	1	0.37
FP Multiply	0.01	2	0.33
FP Add/Sub	0.003	2	0.11
Int Add/Sub	0.00066	1	0.021
FP Min/Max	0.00072	1	0.023
Total			39.69
Avg thread issue	MRPS/core		MRPS/mm^2
89%	5.6		0.14

area the MRPS/mm^2 increases and provides an opportunity to tile more TMs on a chip.

IV. EXPLORING CONSTRAINED RESOURCE CONFIGURATIONS

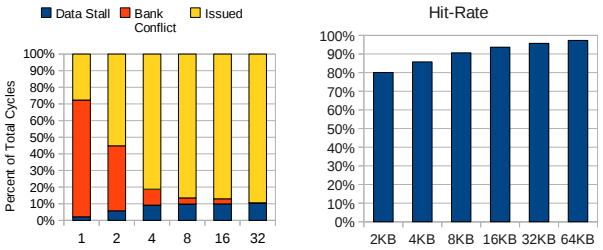
We now consider constraining caches and functional units to evaluate the design points with respect to MRPS/mm^2. Cache configurations are considered before shared functional units, and then revisited for the final multi-TM chip configuration. All performance numbers in our design space exploration are averages from the four scenes in Figure 1.

A. Caches

Our baseline architecture shares one or more instruction caches among multiple cores. Each of these Icaches is divided into one or more banks, and each bank has a read port shared between the cores. Our ˜1000-instruction ray tracer program fits entirely into 4KB instruction caches and provides a 100% hit-rate while being double pumped at 1 GHz. This is virtually the same size as the ray tracer evaluated in [11].

Our data cache model provides write-around functionality to avoid dirtying the cache with data that will never be read. The only writes the ray tracer issues are to the write-only frame buffer; this is typical behavior of common ray tracers. Our compiler stores all temporary data in registers, and does not use a call stack. Stack traversal is handled with a special set of registers designated for stack nodes. Because of the lack of writes to the cache, we achieve relatively high hit-rates even with small caches, as seen in Figure 2. The data cache is also banked similarly to the instruction cache. Data cache lines are 8 4-byte words wide.

We explore L1 Dcache capacities from 2KB to 64KB and banks ranging from 1 to 32, both in power of 2 steps. Similarly, numbers and banks of Icaches range from 1 to 16. First the interaction between instruction and data caches needs to be considered. Instruction starvation will limit



(a) Issue rate for varying banks in a 2KB data cache

(b) Dcache hit%, 8-banks and varying capacities

Fig. 2. L1 data cache performance for a single TM with over-provisioned functional units and instruction cache.

instruction issue and reduce data cache pressure. Conversely, perfect instruction caches will maximize data cache pressure and require larger capacity and increased banking. Neither end-point will be optimal in terms of MRPS/mm^2. This interdependence forces us to explore the entire space of data and instruction cache configurations together.

Other resources, such as the FUs, will also have an influence on cache performance, but the exponential size of the entire design space is intractable. Since we have yet to discover an accurate pruning model, we have chosen to evaluate certain resource types in order. It is possible that this approach misses the optimal configuration, but our results indicate that our solution is adequate. After finding a "best" TM configuration, we revisit Dcaches and their behavior when connected to a chip-wide L2 Dcache shared among multiple TMs. For single-TM simulations we pick a reasonable L2 cache size of 256KB. Since only one TM is accessing the L2 this results in unrealistically high L2 hit-rates, and diminishes the effect that the L1 hit-rate has on performance. We rectify this inaccuracy in section IV-C, but for now this simplified processor, with caches designed to be as small as possible without having a severe impact on performance, provides a baseline for examining other resources, such as the functional units.

B. Shared Functional Units

The next step is to consider sharing lightly used and area-expensive FUs for multiple cores in a TM. The goal is area reduction without a commensurate decrease in performance. Table I shows area estimates for each of our functional units. The integer multiply, floating-point (FP) multiply, FP add/subtract, and FP inverse-square-root units dominate the others in terms of area, thus sharing these units will have the greatest effect on reducing total TM area. In order to maintain a reasonably sized exploration space, these are the only units considered as candidates for sharing. The other units are too small to have a significant effect on the performance per area metric.

We ran many thousands of simulations and varied the number of integer multiply, FP multiply, FP add/subtract and FP inverse-square-root units from 1 to 32 in powers of 2 steps. Given N shared functional units, each unit is only connected to $32/N$ cores in order to avoid complicated

TABLE II

OPTIMAL TM CONFIGURATIONS IN TERMS OF MRPS/MM^2.

INT MUL	FP MUL	FP ADD	FP INV	MRPS/ core	Area (mm^2)	MRPS/ mm^2
2	8	8	1	4.2	1.62	2.6
2	4	8	1	4.1	1.58	2.6
2	4	4	1	4.0	1.57	2.6
4	8	8	1	4.2	1.65	2.6

TABLE III

GTX285 SM VS. MIMD TM RESOURCE COMPARISON. AREA ESTIMATES ARE NORMALIZED TO OUR ESTIMATED FU SIZES FROM TABLE I, AND NOT FROM ACTUAL GTX285 MEASUREMENTS.

	GTX285 SM (8 cores)	MIMD TM (32 cores)
Registers	16384	4096
FPAdds	8	8
FPMuls	8	8
INTAdds	8	32
INTMuls	8	2
Spec op	2	1
Register Area (mm^2)	2.43	0.61
Compute Area (mm^2)	0.43	0.26

connection logic and area that would arise from full connectivity. Scheduling conflicts to shared resources are resolved in a round-robin fashion. Figure 3 shows that the number of FUs can be reduced without drastically lowering the issue rate, and table II shows the top four configurations that were found in this phase of the design exploration. All of the top configurations use the cache setup found in section IV-A: two instruction caches, each with 16 banks, and a 4KB L1 data cache with 8 banks and approximately 8% of cycles as data stalls for both our core-wide and chip-wide simulations.

Area is drastically reduced from the original over-provisioned baseline but performance remains relatively unchanged. Note that the per-core area is quite a bit smaller than the area we estimate for a GTX285 core. Table III compares raw compute and register resources for our TM compared to a GTX285 SM. This is primarily due to our more aggressive resource sharing, and our smaller register file since we do not need to support multithreading in the same way as the GT200. While many threads on the GT200 are context switched out of activity and do not attempt to issue, every single thread in our 32 core TM attempts to issue on each cycle, thereby remaining active. Our design space included experiments where additional thread contexts were added to the TMs, allowing context switching from a stalled thread. These experiments resulted in 3-4% higher issue rate, but required much greater register area for the additional thread contexts.

C. Chip Level Organization

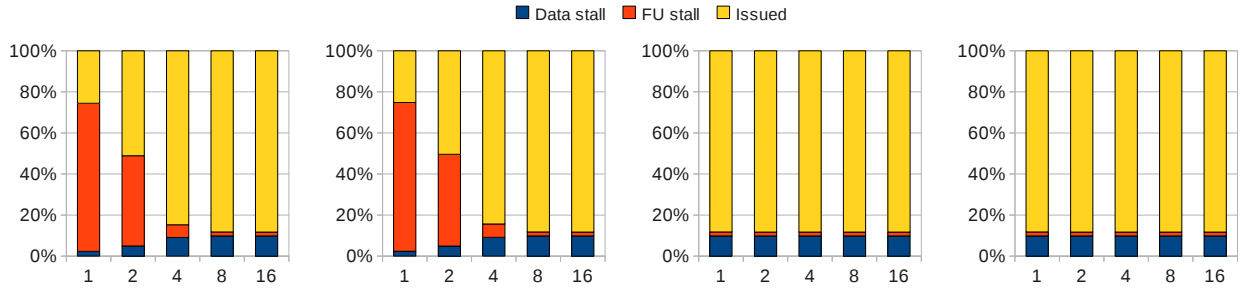
Given the TM configurations found in Section IV-B that have the minimal set of resources required to maintain high performance, we now explore the impact of tiling many of these TMs on a chip. Our chip-wide design connects one or more TMs to an L2 Dcache, with one or more L2 caches on the chip. Up to this point, all of our simulations have been single-TM simulations which do not realistically model L1 to L2 memory traffic. With many TMs, each with an individual L1 cache and a shared L2 cache, bank conflicts will increase and the hit-rate will decrease. This will require a bigger, more highly banked L2 cache. Hit-rate in the L1 will also affect the level of traffic between the two levels of caches so we must explore a new set of L1 and L2 cache configurations with a varying number of TMs connected to the L2.

Once many TMs are connected to a single L2, relatively low L1 hit-rates of 80-86% reported in some of the candidate configurations for a TM will likely put too much pressure on the L2. Figure 4(b) shows the total percentage of cycles stalled due to L2 bank conflicts for a range of L1 hit-rates. The 80-86% hit-rate, reported for some initial TM configurations, results in roughly one third of cycles stalling due to L2 bank conflicts. Even small changes in L1 hit-rate from 85% to 90% will have an effect on reducing L1 to L2 bandwidth due to the high number of cores sharing an L2. We therefore explore a new set of data caches that result in a higher L1 hit-rate.

We assume up to four L2 caches can fit on a chip with a reasonable interface to main memory. Our target area is under 200mm^2, so 80 TMs (2560 cores) will fit even at 2.5mm^2 each. Section IV-B shows a TM area of 1.6mm^2 is possible, and the difference provides room for additional exploration. The 80 TMs are evenly spread over the multiple L2 caches. With up to four L2 caches per chip, this results in 80, 40, 27, or 20 TMs per L2. Figure 4(c) shows the percentage of cycles stalled due to L2 bank conflicts for a varying number of TMs connected to each L2. Even with a 64KB L1 cache with 95% hit-rate, any more than 20 TMs per L2 results in $>$10% L2 bank conflict stalls. We therefore chose to arrange the proposed chip with four L2 caches serving 20 TMs each.

Figure 5 shows how individual TMs of 32 threads might be tiled in conjunction with their L2 caches. The result of the design space exploration is a set of architectural configurations that all fit in under 200mm^2 and maintain high performance. A selection of these are shown in Table IV and are what we use to compare to the best known GPU ray tracer in Section IV-D. Note that the GTX285 has close to half the die area devoted to texturing hardware, and none of the benchmarks reported in [11] or in our own studies use image-based texturing. Thus it may not be fair to include texture hardware area in the MRPS/mm^2 metric. On the other hand, the results reported for the GTX285 do use the texture memory to hold scene data for the ray tracer so although it is not used for texturing, that memory (which is a large portion of the hardware) is participating in the benchmarks.

Optimizing power is not a primary goal of this exploration, and because we endeavor to keep as many units busy as possible we expect power to be relatively high. Using energy and power estimates from Cacti v6.5 and Synopsys DesignWare, we calculated a rough estimate of our chip's



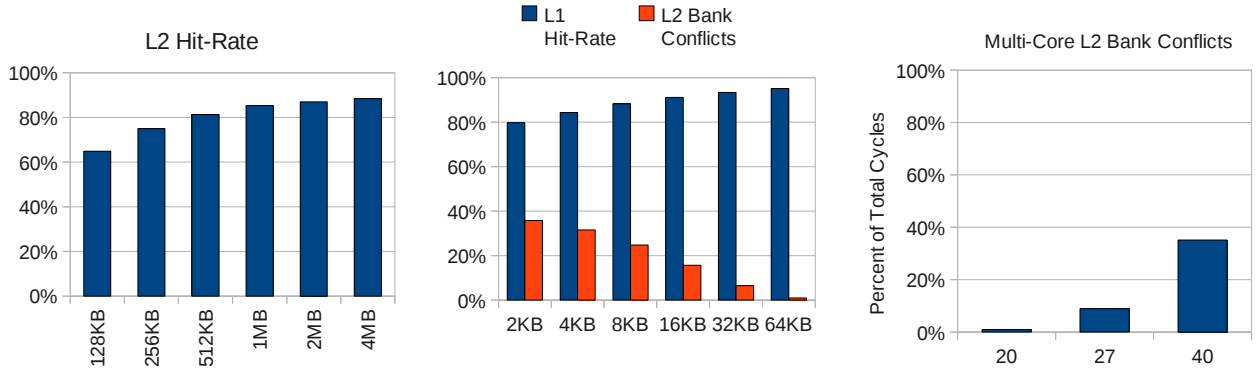
(a) FP Add/Sub (13% of Issued Insts)

(b) FP Multiply (13% of Issued Insts)

(c) FP Inverse Square Root (0.4% of Issued Insts)

(d) Int Multiply (0.3% of Issued Insts)

Fig. 3. Effect of shared functional units on issue rate shown as a percentage of total cycles

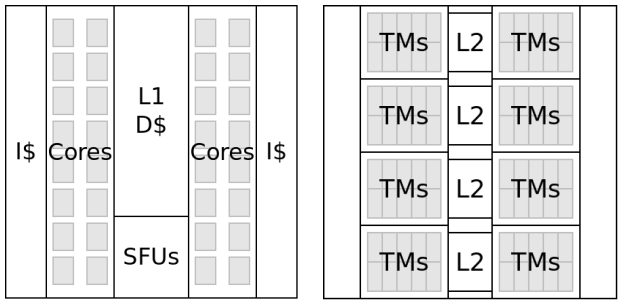


(a) Hit-rate for varying L2 capacities with 20 TMs connected to each L2

(b) Percentage of cycles not issued due to L2 bank conflicts for varying L1 capacities (and thus hit-rates) for 20 TMs

(c) L2 bank conflicts for a varying number of TMs connected to each L2. Each TM has a 64KB L1 cache with 95% hit-rate

Fig. 4. L2 performance for 16 banks and TMs with the top configuration reported in Table II.



(a) TM Layout of 32 cores and shared resources

(b) Chip with multiple TMs sharing L2 caches

Fig. 5. Potential TM and multi-TM chip floor plans

total power consumption on the benchmark scenes. Given the top chip configuration reported in Table IV, and activity factors reported by our simulator, we roughly estimate a chip power consumption of 83 watts which we believe is in the range of power densities for commercial GPUs.

D. Results

To evaluate the results of our design space exploration we chose two candidate architectures from the top performers: one with small area (147mm^2) and the other with larger area (175mm^2) but higher raw performance (as seen in Table IV). We ran detailed simulations of these configurations using the same three scenes as in [11] and using the same mix of primary and secondary rays. Due to the widely differing scenes and shading computations used in [11] and [18], a direct comparison between both architectures is not feasible. We chose to compare against [11] because it represents the best reported performance to date for a ray tracer running on a GPU, and their ray tracing application is more similar to ours. We do however give a high level indication of the range of performance for our MIMD architecture, GTX285, and Copernicus in table IV. In order to show a meaningful area comparison, we used the area of a GTX280, which uses a 65nm process, and other than clock frequency, is equivalent to the GTX285. Copernicus area is scaled up from 22nm to 65nm. Assuming that their envisioned 240mm^2 chip is 15.5mm on each side, a straightforward scaling from 22nm to 65nm would be a factor of three increase on each side, but due to certain process features not scaling linearly, we use a more realistic factor of two per side, giving a total equivalent area of 961mm^2 at 65nm. We then scaled clock frequency from their assumed 4GHz down to the actual 2.33GHz of

TABLE IV

A SELECTION OF OUR TOP CHIP CONFIGURATIONS AND PERFORMANCE COMPARED TO AN NVIDIA GTX285 AND COPERNICUS. COPERNICUS AREA AND PERFORMANCE ARE SCALED TO 65NM AND 2.33 GHZ TO MATCH THE XEON E5345, WHICH WAS THEIR STARTING POINT. EACH OF OUR MIMD THREAD MULTIPROCESSORS (TM) HAS 2 INTEGER MULTIPLY, 8 FP MULTIPLY, 8 FP ADD, 1 FP INVSQRT UNIT, AND 2 16-BANKED ICACHES.

L1 Size	L1 Banks	L2 Size	L2 Banks	L1 Hitrate	L2 Hitrate	Per Cache Bandwidth (GB/s) L1−>reg	L2−>L1	main−>L2	Thread Issue	Area (mm^2)	MRPS	MRPS/ mm^2
32KB	4	256KB	16	93%	75%	42	56	13	70%	147	322	2.2
32KB	4	512KB	16	93%	81%	43	57	10	71%	156	325	2.1
32KB	8	256KB	16	93%	75%	43	57	14	72%	159	330	2.1
32KB	8	512KB	16	93%	81%	43	57	10	72%	168	335	2.0
64KB	4	512KB	16	95%	79%	45	43	10	76%	175	341	1.9
GTX285 (area is from 65nm GTX280 version for better comparison)									75%	576	111	0.2
GTX285 SIMD core area only — no texture unit (area is estimated from die photo)									75%	~300	111	0.37
Copernicus at 22nm, 4GHz, 115 Core2-style cores in 16 tiles									98%	240	43	0.18
Copernicus at 22nm, 4GHz, with their envisioned 10x SW improvement									98%	240	430	1.8
Copernicus with 10x SW improvement, scaled to 65nm, 2.33GHz									98%	961	250	0.26

the 65nm Clovertown core on which their original scaling was based. The 10x scaling due to algorithmic improvements in the Razor software used in the Copernicus system is theoretically envisioned in their paper [18].

The final results and comparisons to GTX285 are shown in Table V. It is interesting to note that although GTX285 and Copernicus take vastly different approaches to accelerating ray tracing, when scaled for performance/area they are quite similar. It is also interesting to note that although our two candidate configurations perform differently in terms of raw performance, when scaled for MRPS/mm^2 they offer similar performance, especially for secondary rays.

When our raw speed is compared to the GTX285 our configurations are between 2.3x and 5.6x faster for primary rays (average of 3.5x for the three scenes and two MIMD configurations) and 2.3x to 9.8x faster for secondary rays (5.6x average). This supports our view that a MIMD approach with appropriate caching scales better for secondary rays than SIMD. We can also see that our thread issue rates do not change dramatically for primary vs. secondary rays, especially for the larger of the two configurations. When scaled for MRPS/mm^2 our configurations are between 8.0x and 19.3x faster for primary rays (12.4x average), and 8.9x to 32.3x faster for secondary rays (20x average). Even if we assume that the GTX285 texturing unit is not participating in the ray tracing, and thus use a 2x smaller area estimate for that processor, these speed-ups are still approximately 6x-10x on average. The fact that our MIMD approach is better in terms of performance per area than the SIMD approach is non-intuitive at first glance. This is mostly because we keep our cores very small due to aggressive resource sharing and by not including extra register resources for multithreading (see Table III).

We believe that MRPS and MRPS/mm^2 are fair units of measurement for ray tracing hardware because they are relatively independent of the resolutions at which the scenes are rendered. To put these MRPS numbers into perspective, if an interesting image is assumed to take between 4-10m rays to render (see Section III), then our MIMD approach would render between 13 (10M rays / 131 MRPS) and 100 (4M rays / 402 MRPS) frames per second (fps) depending on the mix and complexity of the rays. A scene requiring 8M rays (which is a relatively complex scene) at 300 MRPS would achieve 37.5fps.

V. CONCLUSION

Current custom and GPU architectures used for ray tracing have achieved impressive raw performance at a high level of visual realism. However, we believe that wide SIMD GPUs and general purpose MIMD cores are over-provisioned for the specific ray tracing workload and can be an inefficient use of die area for a ray tracing processor. We have proposed an architecture that makes better use of available area, allowing for a greater amount of useful hardware and ultimately higher performance. Our numerous light-weight cores use shared resources to take advantage of the divergent nature of ray tracing code paths and still sustain a high issue rate.

Our architectural exploration has identified a range of solutions that demonstrate speed-ups from 2.3x to 9.8x in raw performance and 8x-32x faster (6x-10x on average with generous area scaling for the GPU) in performance per area over the best reported GPU-based ray tracer. We compare primarily against the GTX285 ray tracing performance because specific data about the architecture and ray tracer is readily available, and it represents some of the best published numbers for ray tracing. We also provide a high-level comparison to the MIMD Copernicus architecture. The new Fermi (GF100) architecture from NVIDIA has caches that would bring its performance closer to our MIMD performance. The initial white paper [28] claims performance gains of 4x with twice the compute resources. Our proposed designs are relatively small at 147-175mm^2. This is encouraging because these designs could potentially be used as a co-processor for accelerating ray performance on existing or future GPUs or CPUs.

TABLE V

COMPARING OUR PERFORMANCE ON TWO DIFFERENT CORE CONFIGURATIONS TO THE GTX285 FOR THREE BENCHMARK SCENES [11]. PRIMARY RAY TESTS CONSISTED OF 1 PRIMARY AND 1 SHADOW RAY PER PIXEL. DIFFUSE RAY TESTS CONSISTED OF 1 PRIMARY AND 32 SECONDARY GLOBAL ILLUMINATION RAYS PER PIXEL.

		Conference (282k triangles)		Fairy (174k triangles)		Sibenik (80k triangles)	
MIMD	Ray Type	MIMD Issue Rate	MIMD MRPS	MIMD Issue Rate	MIMD MRPS	MIMD Issue Rate	MIMD MRPS
147mm^2	Primary	74%	376	70%	369	76%	274
	Diffuse	53%	286	57%	330	37%	107
175mm^2	Primary	77%	387	73%	421	79%	285
	Diffuse	67%	355	70%	402	46%	131
SIMD	Ray Type	GTX SIMD eff.	GTX MRPS	GTX SIMD eff.	GTX MRPS	GTX SIMD eff.	GTX MRPS
GTX285	Primary	74%	142	76%	75	77%	117
	Diffuse	46%	61	46%	41	49%	47

MIMD MRPS/mm^2 ranges from 2.56 (Conference, primary rays) to 0.73 (Sibenik, diffuse rays) for both configs
SIMD MRPS/mm^2 ranges from 0.25 (Conference, primary rays) to 0.07 (Fairy, diffuse rays)
SIMD (no texture area) MRPS/mm^2 ranges from 0.47 (Conference, primary) to 0.14 (Fairy, diffuse)

We have used a relatively full-featured, but traditional ray tracer. Future work will involve adapting the application to better fit the architecture. We envision leveraging the high performance of the system to perform bounding volume hierarchy (BVH) updates on the GPU for dynamic scenes rather than relying on BVH re-builds on the CPU as is currently done, and adapting the ray tracer to handle run-time procedurally generated geometry to name just two areas of interest. There are also other applications used in real time situations, including video games, that could benefit from this type of architecture.

REFERENCES

[1] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[2] A. Glassner, Ed., *An introduction to ray tracing*. London: Academic Press, 1989.

[3] P. Shirley and R. K. Morley, *Realistic Ray Tracing*. Natick, MA: A. K. Peters, 2003.

[4] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing," *Computer Graphics Forum (EUROGRAPHICS '01)*, vol. 20, no. 3, pp. 153–164, September 2001.

[5] K. Dmitriev, V. Havran, and H.-P. Seidel, "Faster ray tracing with SIMD shaft culling," Max-Planck-Institut für Informatik, Tech. Rep. MPI-I-2004-4-006, December 2004.

[6] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, no. 3, pp. 1176–1185, July 2005.

[7] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -," in *Interactive Ray Tracing IRT08*, Aug. 2008, pp. 49–57.

[8] J. Bigler, A. Stephens, and S. Parker, "Design for parallel interactive ray tracing systems," *Interactive Ray Tracing IRT06*, pp. 187–196, Sept. 2006.

[9] E. E. Catmull, "A subdivision algorithm for computer display of curved surfaces." Ph.D. dissertation, University of Utah, 1974.

[10] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *SIGGRAPH '06*. New York, NY, USA: ACM, 2006, pp. 485–493.

[11] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *HPG 2009*, 2009, pp. 145–149.

[12] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR – a hardware architecture for realtime ray-tracing," in *EUROGRAPHICS Workshop on Graphics Hardware*, September 2002.

[13] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip," in *Graphics Hardware Conference*, August 2004, pp. 95–106.

[14] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, no. 3, July 2005.

[15] S. Woop, E. Brunvand, and P. Slusallak, "Estimating performance of a ray tracing ASIC design," in *IRT06*, Sept. 2006.

[16] C. Gribble and K. Ramani, "Coherent ray tracing via stream filtering," in *Interactive Ray Tracing IRT08*, Aug. 2008, pp. 59–66.

[17] K. Ramani, C. P. Gribble, and A. Davis, "Streamray: a stream filtering architecture for coherent ray tracing," in *ASPLOS '09*. New York, NY, USA: ACM, 2009, pp. 325–336.

[18] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark, "Toward a multicore architecture for real-time ray-tracing," in *MICRO '08*, Washington, DC, USA, 2008, pp. 176–187.

[19] L. Seiler, D. Carmean, E. Sprangle *et al.*, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, August 2008.

[20] J. Spjut, D. Kopta, S. Boulos, S. Kellis, and E. Brunvand, "TRaX: A multi-threaded architecture for real-time ray tracing," in *6th IEEE Symposium on Application Specific Processors (SASP)*, June 2008.

[21] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A multicore hardware architecture for real-time ray tracing," *IEEE Transactions on Computer-Aided Design*, 2009.

[22] D. Kopta, J. Spjut, E. Brunvand, and S. Parker, "Comparing incoherent ray performance of TRaX vs. Manta," in *Interactive Ray Tracing IRT08*, August 2008, p. 183.

[23] NVIDIA, "GeForce GTX 200 GPU architectural overview," NVIDIA Corporation, Tech. Rep. TB-04044-001_v01, May 2008.

[24] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark, "Razor: An architecture for dynamic multiresolution ray tracing," Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-07-52, January 2007.

[25] A. Mahesri, D. Johnson, N. Crago, and S. Patel, "Tradeoffs in designing accelerator architectures for visual computing," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 8-12 2008, pp. 164 –175.

[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO '07*, 2007, pp. 3–14.

[27] Chris Lattner and Vikram Adve, "The LLVM Instruction Set and Compilation Strategy," CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2002-2292, Aug 2002.

[28] NVIDIA, "NVIDIA GF100: World's fastest GPU delivering great gaming performance with true geometric realism," NVIDIA Corporation, Tech. Rep., 2009, http://www.nvidia.com/object/GTX_400_architecture.html.