XMT-GPU: A PRAM Architecture for Graphics Computation *

Thomas DuBois
University of Maryland
College Park, MD
tdubois@cs.umd.edu

Bryant Lee
Carnegie Mellon University
Pittsburgh, PA
bryantl@cs.umd.edu

Yi Wang
Virginia Polytechnic Institute
Blacksburg, VA
samywang@vt.edu

Marc Olano
University of Maryland, Baltimore County
Baltimore, MD
olano@umbc.edu

Uzi Vishkin
University of Maryland
College Park, MD
vishkin@umiacs.umd.edu

Abstract

The shading processors in graphics hardware are becoming increasingly general-purpose. We test, through simulation and benchmarking, the potential performance impact of replacing these processors with a fully general-purpose parallel processor, without the fixed-function graphics hardware legacy of current graphics processing units (GPUs). The representative general-purpose processor we test against is XMT (for eXplicit Multi-Threading[1]), a PRAM-like single-chip parallel architecture. Performance is compared for two characteristic shaders running in a fragment-limited GPU benchmark harness and on a cycle-accurate XMT simulator. The general-purpose processor is found to be significantly faster at a compute-only shader, but slower on a memory bound texture shader. Finally we analyze the design tradeoffs that would allow combining the best of both worlds: (i) a competitive XMT texture shader, with (ii) a general-purpose easy-to-program XMT many-core approach that scales up or down to the amount of parallelism provided by the application and is even compatible with serial code.

1. Introduction

Procedural Shading is the ability for users to insert custom code into a graphics system to change the computation of surface color and light interaction. Procedural shading first appeared in the complex CPU-based graphics rendering software for film [11, 33, 18]. Since the advent of shading in graphics hardware [25, 31], GPU designs have become increasingly more programmable, and more general-purpose [24, 25, 29]. The subsequent wide availability of large-scale parallel processing has inspired researchers to perform general-purpose computation on GPUs (GPGPU) [6, 7, 8, 16, 22]. However GPUs retain certain limitations from their graphics roots that make them difficult to program effectively.

The parallel random access machine (PRAM) model for high performance parallel programming is a natural generalization of the serial model where memory accesses take a uniform amount of time across all threads and further synchronization, such as barriers, is unnecessary. It is an easy model for parallel reasoning and programming [13]. As such, PRAM came to be the dominant model for parallelism in the theory community, and at least three major standards texts on serial algorithms and data structures include a major chapter on PRAM algorithms [2, 12, 26]. Some limited empirical validation regarding the relative PRAM ease-of-programming was reported by Hochstein, Basili, Vishkin and Gilbert [21]. A recent innovation of the XMT project, demonstrated with high school students and through a university course offered to Freshmen that are not majors in computer science, is that these people can comprehend PRAM high-level parallel algorithmic thinking and successfully turn it into PRAM programming.

The continued increase of silicon capacity has finally made possible building a PRAM-like chip which could replace a serial CPU. It seems evident that as long as single core clock rates remain stable, some such on-chip parallel system will replace the standard general-purpose CPU, and that system must be easy to program as well as highly efficient. XMT (for eXplicit Multi-Threading) is a possible candidate to be this general-purpose on-chip parallel CPU.

*Supported in part by NSF ITR Award CNS-0426683, NSF Award CNS-0626964, NSF grant CCF-0325393, and NSF STTR Award IIP-0339489.

[1]The home page for the XMT project is www.umiacs.umd.edu/users/vishkin/XMT. This work was partially supported by NSF grants CCR-0325393 and STTR-0339489.

Currently an XMT simulator and FPGA implementation are available for performance testing [41, 42]. For these reasons we chose XMT as a representative general-purpose parallel chip for our benchmarks.

If an architecture can perform graphics tasks competitively while conforming to the PRAM model, it will offer benefits beyond ease of programming to developers. The PRAM model allows for arbitrary memory reads and writes as well as dependencies between threads. In addition, within a PRAM algorithm it is easy to spawn differing numbers of threads for different purposes during the algorithm — per-pixel, but also per-texture sample on an object, or per-sample on an illuminance texture. These operations do not fit the GPU stream model well, generally resulting in multiple rendering passes to overcome the limitations of streaming [19, 15, 44].

In this paper we explore the possibility that this recently available PRAM-like architecture may have a place in graphics hardware. We examine replacing the most general-purpose component of a modern GPU, the shading processors, with a true general-purpose parallel processing unit, specifically an XMT-like subsystem. Clearly, this hybrid GPU would have some of the programming advantages of PRAM, including flexible, load balanced spawning of parallel threads, and thread-to-thread or pixel-to-pixel communication. What is not clear is how it would perform on typical GPU tasks. Therefore, we performed tests on two simple characteristic shaders, one performing MIP-mapped texture lookup, and one computing a brick shader without texture access. GPU versions of these were run on several GPUs, and XMT versions were run on the XMT simulator under several configurations.

We show that XMT can perform computational shading operations faster than a GPU. However it is not competitive enough with texture shading to replace a modern GPU. This observation supports either a general-purpose system with additional graphics specific hardware or the idea of a hybrid XMT/GPU system where the GPU does the work for which it is optimized, and the XMT processor handles all other serial and parallel operations. This system would have better performance and make it easier to program custom tasks. A GPU would not have to support custom shading or other secondary parallel application at all if its co-processor does a better job of it. The first approach of adding additional texture decompression, caching and filtering hardware to XMT is similar to the approach taken by Intel's Larrabee [37], while the second approach is similar to that taken in the PlayStation 3, coupling a Cell Processor and GPU [10].

2. Related Work

GPUs have historically been designed to conform to the stream model of parallel computation [29], or an extension thereof where each kernel operates on multiple data at once. Even with recent advances, this model and its limitations are still present in the graphics APIs for GPUs. The stream model requires programmers to decompose an algorithm into a series of kernels with explicit dependencies between kernels, no dependencies within a kernel, and limited local data storage [34, 32]. The primary advantage of this model is that data access is highly predictable, which reduces hardware complexity.

The concurrent read concurrent write (CRCW) PRAM model, which XMT closely follows, has not been as well used for graphics. In fact, we are unaware of any recent studies of vertex or fragment processing under the PRAM model. The difficulty of building a machine that looks to a programmer like a PRAM has contributed to this situation.

This has not prevented other researchers from exploring alternate architectures for graphics. Recent works include Humphreys et al.'s *Chromium* networks of PCs [23], Chen et al.'s reconfigurable architecture based on the *Raw* multicore processor [9], and Whitted and Kajiya's proposal for a fully-procedural SIMD graphics processor [43].

Within the stream GPU architectures, the kernel processors themselves have become increasingly general multithreaded parallel cores. The first programmable pixel processing was *register combiners*, which defined a configurable ALU stage consisting of two 4-element SIMD adders feeding into a single 4-element SIMD multiplier [28]. These stages were "programmed" by explicitly setting the inputs to each adder and multiplier for up to eight combiner stages in a pipeline. This was succeeded by deep pipelines of ALUs programmed in a special-purpose branch-free assembly language, with a direct mapping from instructions to pipeline stages [14]. Recent GPUs schedule threads to several more general programmable cores for the kernel processors, with variations including the NVIDIA GeForce 6 architecture's shallow pipeline of 4- element computational units [24], and the NVIDIA GeForce 8800's cluster of 16 single-element computational units, with much more general thread scheduling [29].

These general computational units increasingly make up the bulk of the GPU, as compared to a CPU, where cache dominates. In addition to its general computational units, the GPU includes special-purpose hardware assisting the general computational units in texture filtering, clipping, rasterization, and other graphics-specific tasks.

General-purpose programming languages are appearing for the newest of these systems, including CUDA and Brook+ [30, 1]. While the graphics APIs for these new chips still appear to follow a streaming model, CUDA exposes non-streaming details of the GPU architecture, including the partitioning of threads into *blocks* and SIMD-like *warps*, as well as the ability for fast shared data access within a warp and slower shared access within a block.

Though the CUDA model is more general-purpose, adapting even a classic parallel programming primitive to it is complex enough to merit independent publication [36]. We show that XMT could be the next step, and that a similar or better computational efficiency can be achieved with XMT using the easier to program PRAM model.

3. XMT System

Parallel programming can be an intimidating task. Implementing an efficient algorithm on a real system can involve details such as explicit management of memory, communications, and threads. Furthermore, an implementation that works well on a two or four processor system may not be useful at all on a thousand processor system.

XMT [40, 27, 41, 42, 39] was designed as a system to improve overall parallel productivity. One obvious measure of productivity is single task completion time, and this is a primary goal of XMT. However XMT strives to optimize other factors as well. It provides a methodology for converting PRAM algorithms to programs in XMT-C — an extension of C with a small number of special instructions. The XMT-C compiler then applies techniques such as data prefetching and broadcasting to further increase performance while maintaining some natural memory consistency guarantees. Such a program should run on an XMT architecture of any size with performance modeled by the work-depth model of PRAM efficiency. This model measures a program by its total number of operations executed (*work*) and its runtime (*depth*) assuming unlimited hardware. For an in-depth look at XMT performance modeling, see [39].

3.1. Architecture

The XMT architecture consists of a single master thread control unit (MTCU), multiple clusters of TCUs, multiple memory modules, an interconnection network between them, and additional support for some XMT specific operations. Figure 1 provides a block diagram of the system.

The system alternates between serial mode, with only the MTCU running, and parallel mode, with each of the cluster TCUs running. The MTCU is a modern uniprocessor with its own cache. When the MTCU runs a `spawn` instruction, the system switches to parallel mode until all TCUs encounter a `join`, as in Figure 2. Each cluster consists of several TCUs, functional units shared between them, and a single memory port. There is a small read only cache/prefetch buffer for each cluster, however all writes are committed directly to memory.

The memory system consists of a set of independent memory modules that partition the address space. There are one or two times as many memory modules as clusters. Each module has a single FIFO port to the high speed, highly parallel interconnection network[3]. Using hashing,

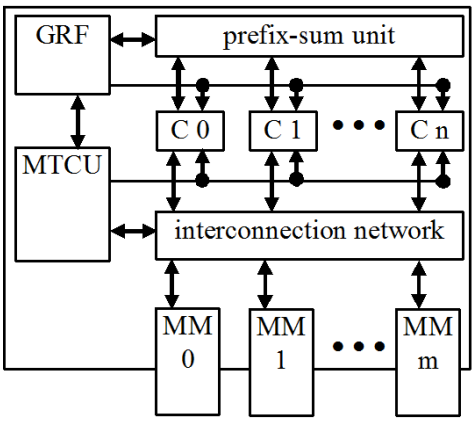

Figure 1. The XMT architecture. Blocks are a master thread control unit (MTCU), global register file (GRF), memory modules (MM), and clusters of thread control units (C).

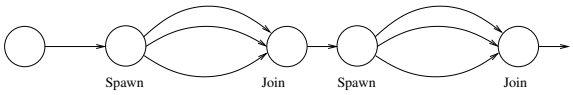

Figure 2. Multithreaded execution flow on XMT.

the address space is spread out among the modules so that nearby or evenly spaced addresses do not necessarily map to the same module. Thus there are no likely access patterns which will cause low contention on average but high contention at a small number of modules. Reducing memory contention can make a significant difference in memory access latency. The remaining latency can be partially or totally hidden through the use of prefetching reads, non-blocking writes, and the small read only cache within each cluster.

3.2. Programming with XMT-C

XMT-C is largely C with the additional instructions `spawn`/`join` and `ps`. From the programmer's perspective a `spawn` block acts as an asynchronous parallel `for` loop. In its simplest usage a `spawn(low,high)` can replace a serial `for` loop from `low` to `high` as long as there are no data dependencies between loop iterations. Each iteration through the original loop becomes one virtual thread with the thread ID, accessed through the reserved character `$`, replacing the loop variable. Virtual threads are mapped to TCUs as they become available. Once all virtual threads complete, the `spawn` ends and the MTCU resumes serial execution. Figure 2 shows the flow between serial and parallel mode using `spawn`.

There are two ways to guarantee an ordering between instructions in different virtual threads. The first

method is to end one spawn and start another as in Figure 2. The second involves the use of the prefix sum instruction, `ps(local,psReg)` (prefix-sum to memory `psm(local,variable)` is similar). When `local=1` this instruction acts as an atomic fetch and increment on the accumulator `psReg` with the previous value stored in `local`. Within a `spawn` block the `psReg` global register can only be accessed through a `ps` statement. `ps` is often used to generate values over a range that are unique across all virtual threads.

For example, the underlying mechanism that assigns virtual thread IDs to TCUs is a `ps` with `local=1`. When one virtual thread completes, its TCU uses `ps` to get its next ID. We spawn virtual threads for each fragment and use this ID to locate the per-fragment data. For threads that generate a variable amount of data (as in a geometry shader [5]), a `psm` with `local` values equal to the number of outputs will give each thread a unique address to store its outputs in a packed result buffer.

Other work has established XMT's ability to handle general algorithms, even those with nontrivial parallelism. Vishkin et al. [39] give a detailed explanation and runtime analysis of several applications written for XMT-C. They include but are not limited to BFS, quicksort, sample sort, dense matrix multiplication, and graph connectivity.

4. Simulated Architecture

We simulate an architecture consisting of XMT-based fragment processing, with all other portions of the GPU pipeline unchanged. In this model all fragment processing is managed by XMT, giving XMT full control over the fragment-level parallelism. One potential benefit is that the XMT/fragment programmer can use multiple spawns and joins within a fragment program. This capability is not used to its fullest extent in our testing since the GPU cannot do it and thus it is not compatible with our head-to-head testing paradigm. However, generally it could allow a programmer to express multiple types of parallelism and multiple data to thread remappings within a single fragment program.

To simulate this architecture, we modified a version of the Mesa OpenGL Library to record span information during rendering to a file, and to accept processed spans from a file to complete the pipeline. Since Mesa operates on spans and our XMT fragment program assumes a fragment buffer, we include two additional processing stages to transform this span data to and from fragment buffer form (Figure 3). The converted fragment data is placed in simulated memory as input to an XMT-C program running on an XMT simulator.

The simulator executes a program one clock cycle at a time, modeling the cycle accurate behavior of individual hardware elements. It takes into account factors including

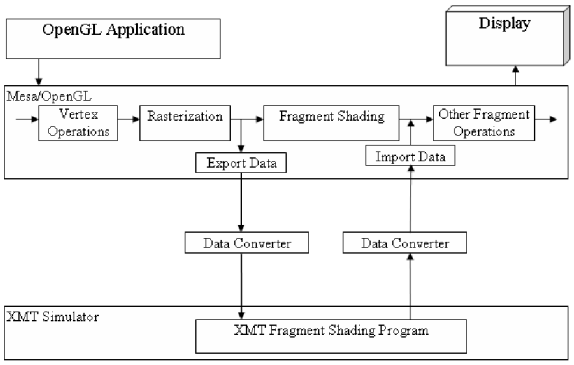

Figure 3. This diagram shows the process by which the fragment information passes through the XMT simulator.

but not limited to functional unit contention and latency, interconnection network contention and latency at each node of the network, L1 cache behavior, and prefetch buffers.

We gathered statistics for three different sets of XMT implementations. All of them use a base system with 64 clusters of 16 TCUs, for a total of 1024 TCUs, each running at 1GHz. The simulator gives clock cycles in terms of the interconnection network, which is twice the rate of the TCUs.

We had significant freedom in choosing the scale of our base system. To provide the fairest comparison, we wanted a system that would closely match the GeForce 8800. However published specifications are not sufficiently detailed for us to determine with certainty how closely a target XMT and the commercial GPU match. For example, we know from the technical brief that the NVidia GeForce 8800 has 128 scalar processors, but not which instructions are supported by those processors, their level of pipelining, or the latency of arithmetic operations. One important and easily comparable detail they give is that the GeForce 8800 has "roughly 520 gigaflops of raw shader horsepower."[29] Our base design closely matches the GeForce in this regard with roughly 512 raw gigaflops in 8 floating point units per cluster.

The following shaders are hand-written in XMT-C. The first two match the operations and algorithm of the GPU versions, and the third matches those operations modulo some approximations.

XMT version 1: The first set of tests was a direct conversion from the Mesa code. The only XMT specific optimization involved was that `for` loops over all fragments were replaced by `spawn` blocks to exploit their parallelism.

XMT version 2: The next set of tests adapts to work around limitations of the current XMT-C compiler. While work is ongoing to automate these features, the compiler does not make use of memory prefetching or non-blocking

stores. For version 2, instructions for these features were added manually. Additionally, the compiler treats code within a spawn block as it would treat a function; it does not do optimizations such as moving idempotent reads and calculations out of the iterated section. As a result, each TCU in version 1 may reload constants once per fragment, when they could be loaded once per TCU regardless of how many fragments that TCU processes. If we cluster the operations such that we spawn exactly one thread per TCU and use `ps` to iterate over the fragments, then the compiler is able to identify and load constants once per TCU. This blocking factor is on par with the blocking factors of some current GPUs.

XMT version 3: The third version applies to texture shading only. For this version we add additional graphics-specific instructions to the simulator, compiler, and XMT-C code. These include equivalents to the GPU instructions `FLR` (floor), `FRC` (fractional part), and `LRP` (linear interpolation). These all are simple operations that a floating point ALU could handle in one step, and are known to be common operations within shading computations. Yang and Lee further discuss the benefits such ISA extensions [45].

In addition to these new instructions, XMT version 3 also includes an estimate of the performance improvement possible by operating on four channels at once within each TCU, as GPU ALUs do natively. This is an improvement for many graphics programs, though recent GPUs have avoided this optimization to achieve greater ALU utilization. We do not simulate this change to XMT explicitly, but approximate it by loading and operating on only one of the four channels. This accurately captures the impact of common 3-vector and 4-vector operations. It underestimates the performance gains possible by packing less-related computations into a four-channel ALU, as is common in GPU programming.

5. Results

We compare the fragment shading performance of the three modeled XMT systems and three GPUs from different generations and manufacturers. To compare fragment performance to fragment performance, we carefully constructed a test program to be fragment limited. Our test bench consists of an OpenGL program rendering a simple screen-filling pyramid, covering several MIP levels (Figure 4). It consists of approximately 800,000 fragments to be shaded.

Since test shaders needed to be re-implemented on the GPU and in multiple XMT versions, we chose two short characteristic shaders. The first is a standard MIP-mapped texture mapping of a 256×256 texture, to test memory access. Since the XMT system does not have the texture filtering or texture caching hardware of the GPU, we expect this to be a memory-bandwidth challenge for it. The GPU version uses native texturing. The second fragment program is a classic brick shader, to test computational speed. The GPU version is implemented as a fragment shader. Neither shader requires more than a single stream pass, and therefore they do not test any of the fined grained communication of XMT or the newer GPUs. If anything this focus on classical graphics tasks biases our results in favor of the GPUs.

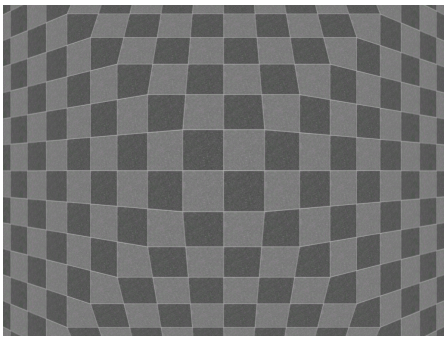

Figure 4. The 768×1024 pixel scene used in the experiments. This is the texture mapped version of the scene.

Because the XMT programs only include the fragment processing stage, for the best comparison we needed to verify that this stage is the bottleneck on the GPUs. Following the methodology for locating bottlenecks in the pipeline outlined by Rege and Brewer [35] and Hart [20], we ran a series of tests designed to add work to a limited section of the pipeline while keeping work elsewhere constant. If the FPS rate does not change much, that stage can be eliminated as a potential bottleneck. We found that shading was the bottleneck on both the ATI x700 Pro and the GeForce 7900 GTX. However, the results were inconclusive on the GeForce 8800 GTX. Unfortunately we know of no way to isolate a bottleneck in the 8800's unified architecture.

5.1. Benchmark Results

Our tests show that the GPUs are faster at texture shading, while XMT has faster brick shading performance and the significantly better programmability of a PRAM. The results from each of the configurations is given in Table 1. The most heavily optimized versions of the XMT shaders - XMT version 2 for the brick shader - ran at 3222 FPS and - XMT version 3 for the texture shader - ran at 487 FPS. This demonstrates an improvement over all GPUs tested for the brick shader, but a substantial loss for the texture shader (.26× vs the ATI and .15× vs the GeForce). At first it is counterintuitive that XMT would be faster with one operation while the GPUs are much faster with the other. We now suggest some reasons for this inversion, and what it says about using XMT for graphics.

Configuration	Texture	Brick
ATI x700	1846	354
NVidia GeForce 7900	8632	1917
NVidia GeForce 8800	3179	2760
XMT version 1	197	1423
XMT version 2	275	3222
XMT version 3	487	

Table 1. Frames per second results for all shaders. The best XMT version does better at brick shading than any GPU tested, however it performs worse than the GPUs on texture shading. Also the 8800 sacrifices texture shading performance for more general performance compared to the 7900.

	Texture Shader		Brick Shader	
#TCUs / Cluster	FPS	memory usage	FPS	memory usage
16	487	.17	3222	.16
32	782	.26	5150	.23
64	1221	.39	7643	.31
128	1763	.59	8491	.51
192	2035	.74	8436	.56
256	2148	.85	8335	.56

Table 2. Frames per second and memory utilization of XMT with varying computational capacity.

When looking at the XMT-C shader programs in isolation, these results seem reasonable since the texture shading program is much more computationally and memory intensive than the brick shading program. The brick shader does a few floating point operations to determine whether the fragment should be brick or mortar colored and then writes the appropriate color values. The texture shader has to figure out the correct level, read values from multiple levels of the texture, interpolate those values, and write out the result. The XMT architecture is not optimized for either shader over the other, so its results reflect the actual difficulty of the programs.

Therefore the asymmetry can best be explained on the GPU side. Specifically the GPU hardware is heavily optimized for streaming texture shading, with specialized cache and functional units for texture management. The natural conclusion here is that the GPUs do better than a general-purpose architecture at tasks for which they are optimized, and the general-purpose system does better on other tasks. We have only shown that this statement applies to brick shading, but given XMT's performance on a wide variety of tasks shown by Vishkin et al. [39] and Gu and Vishkin [17] it is likely that these results would extend to other application such as physics processing.

The comparison between the two GeForce chips shows that the 8800 sacrifices texture shading performance in favor of more general performance. This fact is indicative of the wider trend toward more general-purpose performance on GPUs. This trend is described by Thompson, Hahn, and Oskin [38], and is more recently evidenced by comparing the GeForce 6 series architecture [24] with the GeForce 8800 architecture [29].

5.2. XMT Scalability Analysis

Finally we explore the computational scalability of the XMT architecture on fragment shading. We ran a series of tests using the most heavily optimized versions of both shaders on a subset of the scene. For these tests we kept the memory, interconnection network, and number of clusters constant with 64 clusters and 128 memory modules. Across the tests we varied the number of TCUs in each cluster from 16 to 256. The 16 TCUs per cluster system is the same one used in the previous section. For each test we recorded the total time to completion, how long on average each instruction takes during the processing of a fragment, and what fraction of memory access instructions are satisfied by the cluster cache (and therefore do not require the interconnection network).

One way to implement the equivalent of 256 TCUs per cluster without requiring much more area than the 16 TCU base system is to bring back an idea from the Cray MTA, where each of 16 TCUs effectively emulates 16 "virtual TCUs" by context switching among them in a round robin fashion. Another technique would be to augment the ISA with SIMD graphics instructions, that way each TCU could operate on 16 fragments at once.

The results from these tests are shown in Table 2. It is evident that the frame rate changes nearly linearly with memory utilization. The frame rate deviates somewhat from this pattern when under heavy load, due primarily to functional unit contention. This observation agrees with the intuitive idea that each fragment requires a fixed number of memory accesses, and that throughput should be linearly proportional to the rate at which these happen. Thus we estimate the maximum texture shading performance of a fixed memory system and algorithm at approximately 2800 frames per second. A similar calculation bounds the maximum brick shading rate at above 20000 FPS. Algorithmic changes that could improve on this bound include compressing textures, clustering fragments, or other methods to decrease memory accesses. These upper bounds assume ideal memory and computation utilization, and are thus somewhat higher than the observed values near 2148 and 8335 respectively.

To better understand how computation and memory capacity affect throughput we examine instruction level aver-

age per fragment behavior. For the average fragment, the program runs a fixed sequence of instructions that take T cycles. Those cycles can be divided into memory accesses and computation. Let C be the average number of computation cycles per fragment, and M be the number of memory accesses that are not satisfied locally by the prefetch buffers. The time for each memory access can be separated into unhidden memory latency L plus cluster port queuing delay Q.

We give two distinct lower bounds on T. Any iteration must take an average of $L + Q$ cycles for each of M memory accesses and C cycles of computation. Therefore $T \geq M(L + Q) + C$. Additionally, there are P TCUs that must share a single cluster memory port for their M accesses. If the memory port processes one of them every a cycles, it will take $a * M * P$ cycles for all of them to be satisfied. Therefore we have a bound $T \geq a * M * P$.

If $C+M(L+Q) > a*M*P$ we call the system computationally limited, if $C + M(L + Q) < a * M * P$ we call it memory limited, a system where they are equal is balanced. When the system is computationally limited, as in our base system, $T \geq C + M(L + Q)$. In such a system, decreasing computation time or average latency will have a substantial affect on performance and increasing parallelism will linearly increase performance, though increasing memory bandwidth will have only a marginal effect. When a system is memory limited, the throughput is given by $\frac{1}{a*M}$ which does not depend on computational resources or latency at all. In such a configuration, performance can only be improved by making the memory system faster or decreasing the amount of memory accesses per iteration.

As P increases in a TCU, a system pivots from being computationally limited to bandwidth limited when $C + M(L+Q) = M*a*P$, or alternatively $\frac{C}{M}+L+Q = a*P$. This corresponds to processors generating packets for the memory network at exactly the rate at which network can process them. If L and Q were static with respect to the network utilization, it would be a simple matter to determine at exactly what level of per cluster parallelism the memory system saturates, however they are not static. The unhidden network latency L increases for two reasons. First, as the network utilization goes up, round trips take longer. Balkan, Qu and Vishkin examine this effect in detail[4]. They show that for our 64 terminal configuration, one way time at low or moderate utilization is 16.9 and increases to 42.7 at full utilization. Secondly, as a higher percentage of each processor's cycles are spent waiting for memory accesses, the fraction of the latency that is hidden decreases to 0. The average cluster memory port queuing delay Q starts at ≈ 0 when the memory bandwidth greatly exceeds the rate of requests, and approaches P minus a constant. This is because the ratio of time spent in the queue comes to dominate each loop iteration and any new request must be added behind all $\approx P$ previous requests in the queue. Since average memory response time only increases with utilization, we observe a range of values for P during which the system pivots from computationally limited to memory limited.

For the texture shader our experiments show that with 8 TCUs per cluster, $L_{\text{low}}+Q_{\text{low}} \approx 25$, which corresponds to very little latency hiding and some queuing delay. Also $C \approx 1860$, $M = 50*$prefetch buffer miss rate ≈ 16, and $a \approx 1.9$ for higher numbers of TCUs. Therefore we expect the range at which the system pivots to start with $\frac{1860}{16} + 25 = 1.9 * P$ or $P \approx 74$. The high end of the range we expect to be near $\frac{1860}{16} + 25 + 2 * (42.7 - 16.9) + (Q_P - Q_{\text{low}}) = 1.9 * P$. With $Q_P \approx P$ and Q_{low} small this gives $P \approx 214$. The experimental results shown in Table 2 agree with this pivot range for P. The throughput increases rapidly before $P = 64$, very slowly after $P = 192$, and the rate changes gradually in between.

We perform a similar analysis for the brick shader with $C = 526$, $M = 7 *$ prefetch buffer miss rate ≈ 3.5, and $L_{\text{low}} + Q_{\text{low}} = 8$ (corresponding to low utilization and effective latency hiding). These parameters give a pivot value of P between $\left(\frac{526}{3.5} + 8\right) * \frac{1}{1.9} \approx 79$ and $\left(\frac{526}{3.5} + 8 + 2 * (42.7 - 16.9) - Q_{\text{low}}\right) * \frac{1}{.9} \approx 233$. Our model does not account for measured brick shading memory utilization leveling off at .56. So to test our model we compare the results for the two shaders on 16 to 64 TCUs, where memory utilization is still increasing. Those results suggest that the brick shader would reach the pivot range with P slightly higher than the texture shader, which agrees with the model's brick shader range of $79 - 233$.

6. Architectural Extensions

We have demonstrated the ability of the XMT system to perform texture shading at a rate about $\frac{1}{6}$ of the fastest GPU we tested, and custom shading at a rate faster than any of the GPUs we tested. We see two main directions in GPU development that these results suggest.

6.1. Customized XMT

It should be noted that the graphics specific instructions added to the simulator in XMT version 3 are only a few of many potential optimizations to make. In addition, just as current GPUs consist of somewhat general-purpose computational cores supplemented by special-purpose graphics subsystems, we could add the same subsystems to an XMT-based GPU. These would include texture filtering, per-cluster read only texture caches so texture reads would not require a round trip to the memory system, and hardware support for compressed textures. Given the faster performance of XMT on the computational benchmarks, this would help to balance the texture advantage of the current GPUs, and could result in a new GPU with competitive

performance, but with a much more general programming model.

6.2. Combination Systems

An alternative way to combine ideas and strengths from the two architectures is to combine a GPU with an XMT chip on the same system. Currently a system will pair a GPU with a serial CPU, and all work is divided between them such that inherently serial computation is performed by the CPU, and parallel computation by the GPU. On this system, the less stream based a parallel application is, the more unnatural it is to run it on a GPU. By instead pairing a GPU with XMT we can achieve the best of both worlds. Graphics applications can run predominantly on the GPU while all other applications, both serial and parallel, run on the XMT chip.

This type of setup offers several advantages over the hybrid approach above. With one general-purpose chip and one specialized graphics chip neither has to compromise any of its functionality to more closely match the other model. Future GPU development can focus more on graphics and streaming without losing anything in an attempt to be more general than needed. XMT development can continue to focus on running PRAM algorithms quickly without losing anything to specialization.

7. Conclusion

In order to better support applications whose demands lay outside of the traditional graphics pipeline, graphics processor architectures have become more similar to general-purpose parallel architectures. At the same time, technological developments have put the prospect of a true general-purpose multiprocessor, one that closely matches the PRAM model for parallel programming, within reach.

We have shown that such a general-purpose on-chip multiprocessor can be competitive with current GPU hardware in some graphics applications, while maintaining all of the benefits that come from being designed from the ground up as a general-purpose architecture.

This suggests that development and implementation of general-purpose CPUs will affect the future of GPU development. GPU designs can borrow both programming models and architectural ideas from the general-purpose parallel community in order to support a wider range of algorithms. Alternatively they can drift further away from general-purpose, recognizing that their co-processor can aptly handle more complicated forms of parallelism.

A possible consequence is that in the short term GPUs will continue to broaden their outreach. However, once CPU vendors become as aggressive about the incorporation of parallelism towards faster completion of single tasks (e.g., using many-core architecture) as GPU vendors already are, these new CPUs will start competing with GPUs on applications for which the CPUs will be good enough. This could cause, in turn, an eventual retreat of GPUs to co-processors engineered to handle the more graphics-specific forms of parallelism that will be reserved for GPUs.

References

[1] AMD. AMD delivers first stream processor with double precision floating point technology. Press Release, November 2007.

[2] S. Baase. *Computer Algorithms: Introduction to Design and Analysis. Second edition.* Addison-Wesley, 1988.

[3] A. Balkan, M. Horak, G. Qu, and U. Vishkin. Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing. In *Hot Interconnects 2007*, Stanford University, CA, August 2007.

[4] A. Balkan, G. Qu, and U. Vishkin. Mesh-of-Trees and Alternative Interconnection Networks for Single-Chip Parallel Processing. In *17th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2006)*, pages 73–80, Steamboat Springs, Colorado, September 11-13 2006. Best Paper Award.

[5] D. Blythe. The direct3d 10 system. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH 2006)*, 25(3):724–734, 2006.

[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[8] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Proceedings of the 10th Panhellenic Conference in Informatics (PCI 2005)*, 2005.

[9] J. Chen, M. I. Gordon, W. Thies, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *Graphics Hardware 2005*. ACM SIGGRAPH / Eurographics, July 2005.

[10] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007.

[11] R. L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, 1984.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, first edition, 1990.

[13] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1999.

[14] C. Donham, A. Minkin, B. Nordquist, E. Hutchins, M. Tian, and G. S. III. Method and system for scalable dataflow-based programmabel processing of graphics data. US Patent 6,980,209 (Filed June 14, 2002), December 2005.

[15] R. Geiss. *GPU Gems 3*, chapter Chapter 1: Generating Complex Procedural Terrains Using the GPU. Addison Wesley Professional, 2008.

[16] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast database operations using graphics processors. In *Proceedings of ACM SIGMOD 2004*, 2004.

[17] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocesor. *Journal of Embedded Computing, Special Issue: Issues in Embedded Single-Chip Multicore Architectures, 2, 2*, pages 181–190, 2006.

[18] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1990. ACM Press.

[19] T. Harada, S. Koshizuka, and Y. Kawaguchi. Sliced data structure for particle-based simulations on gpus. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 55–62, New York, NY, USA, 2007. ACM.

[20] E. Hart. Opengl performance tuning. Game Developers Conference, 2004.

[21] L. Hochstein, V. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 2008.

[22] D. Horn, M. Houston, and P. Hanrahan. ClawHMMer: A streaming HMMer-search implementation. In *Proceedings of Supercomputing 2005*, 2005.

[23] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702, New York, NY, USA, 2002. ACM Press.

[24] E. Kilgariff and R. Fernando. The GeForce 6 series GPU architecture. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*, chapter 26. Addison-Wesley Professional, 2005.

[25] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, New York, NY, USA, 2001. ACM Press.

[26] U. Manber. *Introduction to algorithms: a creative approach*. Addison-Wesley, 1989.

[27] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. In *Invited Special Issue for SPAA01: TOCS 36,5*, pages 521–552. Springer-Verlag, 2003.

[28] NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.

[29] NVIDIA. NVIDIA GeForce 8800 GPU architecture overview. Technical Brief TB-02797-001_v01, NVIDIA, November 2006.

[30] NVIDIA Corporation. Nvidia cuda compute unified device architecture : Programming guide. Whitepaper, NVIDIA Corporation, 2007.

[31] M. Olano and A. Lastra. A shading language on graphics hardware: the PixelFlow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168. ACM Press, 1998.

[32] J. D. Owens. *COMPUTER GRAPHICS ON A STREAM ARCHITECTURE*. PhD thesis, Stanford University, 2002.

[33] K. Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[34] T. J. Purcell. *RAY TRACING ON A STREAM PROCESSOR*. PhD thesis, Stanford University, 2004.

[35] A. Rege and C. Brewer. Practical performance analysis and tuning. Game Developers Conference, 2004.

[36] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/ EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[37] J. Stokes. Clearing up the confusion over Intel's Larrabee, part II. Ars Technica, June 4 2007. http://arstechnica.com/.

[38] C. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 306– 317, 2002.

[39] U. Vishkin, G. Caragea, and B. Lee. Models for advancing pram and other algorithms into parallel programs for a pram-on-chip platform, 2008. Handbook on Parallel Computing: Models, Algorithms, and Applications, Editors: S. Rajasekaran and J. Reif, CRC Press.

[40] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism (Extended Abstract). In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[41] X. Wen and U. Vishkin. Pram-on-chip: First commitment to silicon. *Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 9-11 2007.

[42] X. Wen and U. Vishkin. Fpga-based prototype of a pram-on-chip processor. *Proc. ACM International Conference on Computing Frontiers, Ischia, Italy*, May 5-7 2008.

[43] T. Whitted and J. Kajiya. Fully procedural graphics. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[44] C. Wyman. Hierarchical caustic maps. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 163–171, New York, NY, USA, 2008. ACM.

[45] X. Yang and R. B. Lee. PLX FP: An efficient floating-point instruction set for 3d graphics. *IEEE International Conference on Multimedia and Expo (ICME)*, 2004.