# Chapter 2. Graphics Hardware

*Marc Olano*

# Ignoring Hardware Differences

Marc Olano
University of Maryland, Baltimore County

One of the great promises of real-time shading is the potential to have a single shading program that can run across a wide range of graphics hardware. While we don't yet have a single cross-platform shading language to satisfy everyone, there is ample evidence that it *is* possible. In this chapter, we discuss what is necessary to create a cross-platform shading language, how shading languages allow us to ignore hardware differences, what range of hardware can reasonably be represented by a single shading language, and what evidence exists now that it will really work.

## *The key to cross-platform shading*

The key to a cross-platform shading language is to work with a common model of shading hardware rather than specifics of the hardware itself. The model is a mental picture of what's going on that shader-writers use to make sense of the code they write. The further you get into hardware specifics, the less general your model becomes.

Designing a good model for shading is the balance of three competing goals. The model should be simple enough for novice users to understand. It should be a good model of the problem domain, accurately describing what the shader is trying to do rather than exactly how to do it. This will allow the shading language compiler to map the shader onto the hardware in the way that is best for the specific hardware platform. Of course, it should also be possible to map it efficiently onto all the desired platforms.

The second goal is particularly important — the purpose of shading code (or any code) is to describe **what** you want done. The compiler can and will made different choices about **how** (within limits — it can't change the algorithms you use, but it can rearrange the execution order, unroll loops, decide if a certain operation should be computed or looked up in a texture, etc.).

### Single Program, Multiple Data

Shading is inherently a very parallel task. Whether we are talking about vertices in an object, a surface diced into micropolygons, ray-traced intersection points, or screen pixels, there is always some relatively common set of operations being applied to a set of samples on the surface. It is this parallel nature that makes shading so approachable by hardware and allows us to even consider real-time shading.

The model that almost every shading system adopts is Single Program/Multiple Data (SPMD), with no processor-to-processor communication. You write a shader to describe what happens to a single sample on the surface (single program). That same single program is run at every sample on the surface (multiple data). SPMD is closely related to the Single Instruction/Multiple Data (SIMD) model of parallel computation, but SIMD implies more about how the program will be executed. With SIMD, a set of parallel processors will runs the same set of instructions in lockstep, but with different data at each processor. SPMD runs the same program, but without any implication of whether the same path through the program must be taken by all processors. On a pure SIMD array of processors, conditional code is handled by disabling a subset of the processors, who must wait while the others process the conditional instructions. Contrast this with the Multiple Instruction/Multiple Data (MIMD) model, where every processor can be running a completely different program.

SPMD is sometimes referred to as "SIMD on MIMD" or "effective SIMD", as it is uses a SIMD style of programming, but can include programs to run on a single processor, MIMD machine or SIMD machine.

### No communication

One of the aspects of shading that has allowed the explosion of fast shading hardware is the independence of each shading sample from every other shading sample. One of the most difficult and expensive aspect of general-purpose parallel machines is the communication network allowing the processors or nodes to communicate with each other. If the need for this communication is removed, the need for synchronization between the processors disappears, as does the need for physical connections between processors. The processors can be packed much more densely and are free to execute on batches of samples, samples in a pipeline, samples one at a time — whatever is most efficient.

Communication costs are also generally so high relative to computational costs, and so dependent on the machine architecture, that introducing processor to processor communication into a SPMD model greatly reduces the kinds of hardware a program can use effectively. The longer we can avoid communication, the more general our shaders will be.

Shaders don't need sample to sample communication because shaders are typically restricted to computing only local lighting models. Anything that makes the appearance of one point on the surface depend on points elsewhere on the surface introduces the need a sample to sample communication. Shadows, global illumination and subsurface scattering are all on the list of effects that break the model to some degree if they are allowed in a shader.

General purpose computations, on the other hand, often require significant processor to processor communication. As graphics hardware becomes more powerful and flexible, there is an increasing desire to use it for other general purpose parallel computation. This comes at a cost in flexibility of the resulting code. I would argue that we need **two** computational models. A model including communication for general purpose computation on NVIDIA and ATI-style hardware, and a model for shading (possibly targeting the general model on hardware that supports it) that is task oriented and unifies vertex and fragment computations.

In the mean time, many people have succeeded in creating general purpose algorithms on GPUs with inter-processor communication. They achieve this feat through the use of multiple passes. On one pass, you write data into textures or buffers in the graphics card. On the following pass, **any** processor can read **any** data from this texture, not just its own. Even if you are willing to accept multiple passes through the hardware, this communication method isn't perfect for all uses. The reader decides what other processor's data to read, and can read at most a handful of data per pass. Some computational algorithms map well to this model, while others would prefer to have the *writer* decide where the data should go. All of that will be covered in more depth later - for now, we'll restrict the discussion to shading.

### *Languages for hardware abstraction*

One of the best examples of a shading language for hardware abstraction is the RenderMan shading language. Shaders written in this language have been successfully targeted to a huge range of different hardware. Pixar's PhotoRealistic RenderMan targets a single processor running each step of the shader in a loop over the micropolygons in a diced-up surface as generated by the REYES algorithm [Cook 1987]. BMRT also targeted a single processor, but as a ray-tracer it ran each shader in its entirety on one ray-intersection sample before moving on to the next sample [Gritz and Hahn 1996]. SGI created a RenderMan implementation targeting multiple rendering passes on graphics hardware, assuming hardware with a fast render-to-texture/read-from-texture or copy framebuffer-to-famebuffer [Peercy et al. 2000]. ATI has created a RenderMan language compiler targeting current shading hardware as part of their ASHLI toolkit.

RenderMan may not turn out to be the best language for hardware shading, but it has done an admirable job at being adaptable to a wide range of hardware. In the model presented by RenderMan, the shader writer tags data as being either *uniform* or *varying*. Uniform data is the

same across a set of samples being shaded at once[1]. Varying data may change from sample to sample.

For compilation of RenderMan shaders, the most important uniform and varying designations are for the inputs to the shader. The shading compiler must derive for itself which intermediate results within an expression are uniform and which are varying. Expressions using only uniform arguments will have uniform results; expressions with any varying arguments will have a varying result. The compiler can use similar logic to decide whether any local variable in a shader is really uniform or varying regardless of how it was specified in the shading code.

Given an accurate idea of exactly which quantities vary across the shaded surface and which don't, the shading compiler can make several choices for actual execution. It can decide to still evaluate every computation at every sample (not using the uniform/varying distinction). It can evaluate the uniform computations once for a set of samples and for each varying computation, loop over the samples to evaluate it. It can execute the varying computations as SIMD instructions across a parallel array of processors. It can execute the entire shader or just the varying computations across a set of MIMD processors. It can create a pipeline of stream processors, each executing one or a few varying instructions on one sample before passing that sample on to the next.

### *Where should we break the portability?*

There are several facets of the RenderMan shading language that are not well suited for graphics hardware. We can expect several of these to be the foundation of differences between real-time languages and the RenderMan shading language, or limitations of hardware-accelerated RenderMan implementations.

Since PRMan version 3.8, the RenderMan shading language has included the ability to call arbitrary code from within a shader. This code can do anything, from compute a specialized noise function to spawn a different style of renderer to download an image from a live camera on the south pole. Until graphics hardware has the ability to run arbitrary code, this won't really be an option for real-time shading.

RenderMan's has just one scalar data type, *float*. Graphics hardware supporting floating point data is now ubiquitous, but the size and precision of the floating point numbers vary. Fixed point or reduced-precision floating point numbers are also provided on some hardware as a faster option than pure 32-bit floating point. With no way to indicate

[1] One RenderMan trick that will tell you something about how many samples are shaded at once, breaking the illusion that all hardware is the same, is to assign a random color into a uniform variable in a RenderMan shader.

the range or precision of computations, a RenderMan compiler cannot know when to use these faster operations. Many of the candidates for a real-time shading language include some reduced precision types for efficiency: the OpenGL Shading Language [Kessenich et al. 2003], Direct3D HLSL [Microsoft 2002], and Cg [Mark et al. 2003].

RenderMan shaders have two computational frequencies (how often a computation happens), uniform and varying. Shading hardware has at least three — compute on the CPU, compute per vertex and compute per fragment, with no interleaving of computation between the levels. All of the languages mentioned above have chosen to break shading computation into separate procedures executing at each of these levels. That choice makes those shaders a poor fit to any hardware or software rendering system that does not follow the CPU/Vertex/Fragment breakdown. However, any alternative language that targeted all three stages must include new types for the new types of computation [Proudfoot et al. 2001].

The RenderMan shading language also includes no real means of communication from sample to sample. This is one of the strengths that allow RenderMan shaders to run on such a wide range of rendering systems, but is a serious restriction for the general computations that are becoming popular on graphics hardware. Communication between processors in current hardware seems best supported by rendering partial results to a texture then using the random access provided by the texturing hardware to find values from other processors in a later pass.

This form of communication is currently limited to fragment shaders and comes at a very high price of communication to instructions. Similar communication at the vertex shader level is possible, though considerably more complicated. The all but the final vertex shader pass can operate on a regular grid of vertices, allowing all vertex and fragment operations to be used (including any vertex and fragment texturing and rendering to texture for communication). In the next-to-last vertex shader pass, the vertex locations (or data necessary to do the final computation) can be rendered in to a vertex array for use in a final vertex shader pass. If multiple passes of fragment shading are needed, they must follow after all vertex shading passes, but need not repeat the multi-pass vertex shading computation.

Obviously, stretching the hardware beyond its intended limits like this introduces a significant burden on the shader developer!  Because users want to write algorithms that use communication, better facilities will appear in real-time shading languages, but as they do they will limit the applicability of those shaders to the class of similar hardware.

Graphics Hardware (2005), pp. 1–8
M. Meissner, B.- O. Schneider (Editors)

# XMT: A PRAM Architecture for Graphics Hardware

Yi Wang[1†] Marc Olano[1‡] Ronny Kupershtok[2§] Uzi Vishkin[2¶]

[1]University of Maryland, Baltimore County [2]University of Maryland, College Park

**Abstract**

*In this paper, we introduce a new graphics hardware architecture based on XMT (Explicit-Multithreading), a fine-grained PRAM (Parallel Random Access Model) design. We demonstrate the advantage of XMT architecture in graphics hardware applications. Compared with the stream architecture, the XMT-based graphics hardware provides higher flexibility, a more friendly programming interface and can achieve similar performance. With an XMT-based GPU architecture, it would be much easier to migrate most general-purpose computation tasks onto the GPU. This capability may increase the utilization of GPU and enhance the performance of the whole computer system. We base our conclusion on architectural analysis, simulation of code segments and example programs.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Hardware Architecture-Graphics processors

## 1. Introduction

The progress of VLSI technology allows us to put more transistors into one chip. What is the best way to use these transistors? For graphics hardware designers, it is a balance between performance and fidelity.

Most current and recent graphics hardware have adopted programmable stream processors to execute vertex and pixel shading programs. Compared with fixed or micro-programmable processing pipeline, this evolution emphasized better rendering effects instead of higher performance.

Current stream-based graphics architectures still have many constraints. They are unable to execute programs with complex control flows efficiently. They have limited memory access. Although we can simulate branching and looping by multi-pass processing, a significant number of stalls is inevitable. There are also several areas where predictable memory accesses are hard to achieve. These include the visibility problem and its various manifestations (radiosity, ray-tracing, photon mapping), view-dependent rendering, physically-realistic graphics (collision detection and response) and rendering of dynamic scenes with moving objects. To solve this problem, we need an architecture that does not block one computation while another concurrent computation is delayed.

On the other hand, as GPUs have become more flexible, people have begun the research to do general-purpose computation using graphics hardware. This is tempting because we want to make full use of GPU even we are not playing 3D games. Some problems have been proved to be solvable on GPU [KSW04, MA03, JBSL03, GHLM05]. But the GPU algorithms to solve these problems are far from direct. Programmers need to have a deep understanding of the graphics processing pipeline of the GPU. They also need to grasp many GPU programming techniques. For example, OCCLUSION_QUERY is often used to test loop conditions. Moreover, the GPU algorithms usually contain some limitations, e.g. preprocessing data on CPU, limited scale of the problem, limited precision, etc.. To easily migrate CPU tasks to GPU, a direct idea is to have GPU architectures as similar as possible to the CPU. But this change may reduce the graphics processing speed of GPU.

† e-mail: yiwang1@umbc.edu
‡ e-mail: olano@umbc.edu
§ e-mail: rkuper@glue.umd.edu
¶ e-mail: vishkin@umiacs.umd.edu

In this paper we introduce XMT (Explicit-Multithreading), a fine-grained PRAM (Parallel Random Access Model) computing model that has the potential to solve these problems. The XMT framework is well-suited for the features of graphics computations: it supports high parallelism both on the data level and the instruction level; it provides high computation rates and optimized local data access. Moreover, the XMT has the ability to support complex programs with random memory access and arbitrary branching and looping. So, most CPU programs can be easily mapped to XMT programs. The stalls caused by logical dependence and memory access can be well hidden by the highly paralleled processing.

In the following sections, we observe recent trends in graphics hardware. We then introduce the XMT framework and its programming model in Section 3. We show its application in graphics field and compare with the stream architecture. In Section 4, we analyze the use of XMT for just the fragment processing portion of the pipeline. In the last section, we conclude our discussion and point our future work.

## 2. GPU Trends

The GPU has experienced fast evolution since its first appearance. It's function has also changed over time, from supporting only 2D operations, to 3D transforming and lighting, to fragment and vertex shading programs.

The OpenGL and DirectX organizations [SA04, Mic02] have been the standards for most graphics hardware in the last two decades. OpenGL defined a simplified and efficient rendering pipeline that can be implemented on hardware. Most current graphics hardware systems implement OpenGL as a streaming pipeline architecture in which the data are processed in different pipeline stages. Each stage is finely tuned to take approximately equal time to keep the pipeline load-balanced. Within each stage, synchronizations between parallel computations is minimized in part to ensure predictable memory accesses.

This organization enabled interactive 3D graphics on PCs. However the built-in shading model is too simple to achieve realistic lighting effects. With the advantage of VLSI technology, programmable shading hardware were added to enable more flexible shading models.

Procedural shading is a technique that uses a shader, a procedure written in a high-level shading language, to calculate object appearance from a set of parameters, including the surface position, surface normal, texture coordinates, texture maps, light direction and colors, etc.. The RenderMan Shading Language [HL90] is the standard for offline procedural shading and is supported by many rendering software packages, e.g. Pixar's PhotoRealistic RenderMan software [Ups90]. Procedural shading is very powerful technique for rendering feature films and commercials in the production industry.

The graphics community has made significant efforts to approach shaders as powerful as RenderMan's in real-time graphics engines. The PixelPlanes 4 and PixelFlow systems use an array of general purpose processors to execute arbitrary shading code on every pixel [RTB*92, OL98]. Peercy et al. used a multi-pass rendering approach to support complex RenderMan shaders [POAU00]. Lindholm et al. were the first to add programmable stream processing for vertex programming to PC graphics hardware [LKM01].

Several real-time shading languages have been proposed to provide a RenderMan like interface for hardware shader writers. The most widely used shading languages include GLSlang [KBR04], Cg [NVD03] and HLSL [Mic05]. The appearance of these C-like shading languages has a series of effects. Programmers can compose shaders more conveniently and they are more likely to create new shaders that are beyond the hardware limitation. Graphics hardware are pushed to become more flexible and evolve toward a direction of general-purpose computation model.

Currently the requirement of general-purpose computations on GPU has little effect on GPU design. But the GPU is a powerful computation resource we would like to be able to use even we are not playing 3D games. GPUs have been proven to be able to perform many other computations than their intended traditional rendering [KSW04, MA03, JBSL03, GHLM05]. They are evolving toward computation intensive coprocessors on the PC system. NVIDIA has published a non-graphics computation model for the GeForce 6 series GPU [PF05].

## 3. XMT Based GPU

### 3.1. XMT Framework

The question of how to think algorithmically in parallel has been a fundamental problem for which past general-purpose parallel architectures did not have an adequate answer. A computational model, the Parallel Random Access Model (PRAM), was developed by numerous algorithm researchers to address this question during the 1980s and 1990s, and was even put into standard algorithm textbooks [Baa88,Man89,CLR90]. However, despite the broad interest PRAM generated, it had not been possible to build parallel machines that adequately support it using multi-chip multi-processors, the only multiprocessors that were buildable in the 1990s. The main insight behind the XMT framework is that this is becoming possible with the increasing amounts of hardware that can be placed on a single chip.

The XMT (Explicit-Multi-Threading) framework currently envisions a 64 bit architecture that supports PRAM-like programming through inter-thread parallelism [VDBN98, NNTV00, NNTV03]. Diagram 1 shows the framework of XMT.

The XMT machine comprises multiple TCUs (Thread

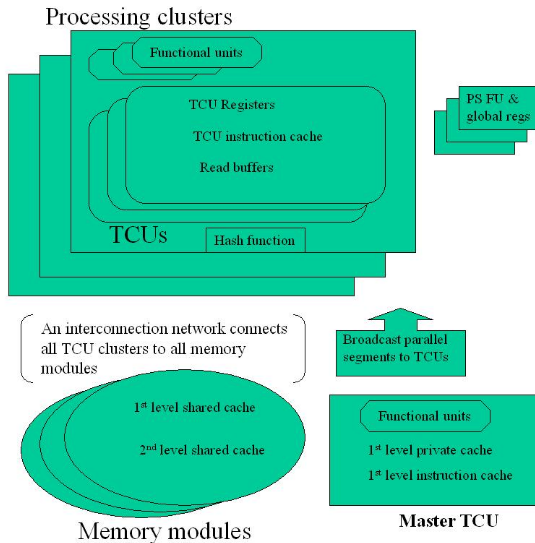


**Figure 1:** *The XMT framework*

Control Unit) and multiple memory modules. The TCUs are divided into clusters. Every TCU is a pipeline with a local set of registers. In addition there is a set of global registers that every TCU can access. All TCUs in the same cluster share a set of functional units. A high throughput highly-pipelined interconnection network between the clusters and the memory modules allows access from every TCU to every memory module.

The interconnection network topology is based on a mesh of binary trees [Lei92]. (i) For every cluster there is a binary fan-out tree and for every memory module there is a binary fan-in tree; each such fan-out tree has one common leaf with each such fan-in-tree. (ii) For every memory module there is a binary fan-out tree and for every cluster there is a binary fan-in tree; each such fan-out tree has one common leaf with each such fan-in-tree.

Among the TCUs there is one MASTER TCU, which is a general purpose super scalar processor that is competitive with today's advanced serial processors. All the other REGULAR TCUs are simple general purpose processors that have the basic functionality to execute general purpose machine code.

An XMT machine code is composed of serial segments and parallel segments alternately. The XMT machine executes in serial mode and in parallel mode, respectively. In serial mode, the MASTER TCU executes serial code segments. Each parallel segment requires the execution of some number of virtual XMT threads. This number can be different from one parallel segment to another. In parallel mode, the REGULAR TCUs execute all the virtual threads of one segment. A *spawn* machine instruction is used to switch from serial mode to parallel mode and a *join* machine instruction is used to switch from parallel mode to serial mode. Spawn and join instructions bound every parallel code segment.

On switching from serial mode to parallel mode, the parallel code segment is broadcasted to the TCUs through a broadcast bus. The parallel code might also include shared data to be used by the TCUs. Concurrent reads from the same memory address are queued. This provides a way to avoid performance penalty due to queuing. Once the first instruction of the parallel code arrives to a TCU, the TCU starts an execution of an XMT virtual thread. Every TCU executes the same code on different data (SPMD) for all the virtual threads on the current segment it implements.

In parallel mode, every TCU executes a virtual thread at its own speed until it reaches a join instruction. A TCU might stall its execution due to a data memory access delay or instruction memory access delay. In any case in the execution of a single virtual thread, a TCU does not depend on the execution of other TCUS. Upon reaching a join, the TCU become available to execute another virtual thread. The hardware uses a low-overhead mechanism to automatically assign a new virtual thread to the available TCU.

In addition to spawn and join, there are three unique instructions in the XMT architecture: *prefix-sum*, *prefix-sum-to-memory* and *single-spawn* (called *fork* in [NNTV00, NNTV03]). The prefix-sum instruction operates on a local register, L, as an incremental variable and a global register, G, as a base variable. The result of prefix-sum (similar to an atomic fetch-and-increment) is that the global register gets the value L+G, while L gets the original value of G. For the case where L is a single bit, prefix sum is an atomic operation, whose concurrent application by several TCUs is implemented in hardware in essentially the same time as a single prefix-sum instruction. Each TCU gets a unique value in its local register L. To prevent concurrent writes to a global register, read or write access to the global register is allowed only by using a prefix sum instruction.

A common use of the prefix sum instruction is for counting. Before prefix-sum, the global register G is assigned the integer value 0 and the L register in every threads gets the integer value 1. After prefix-sum, every thread's L register contains a unique integer between 1 to the number of participating threads. The global register G records the count of threads that executed the prefix-sum. The program segment in Subsection 3.2 shows such a usage. Another usage in graphics applications is the occlusion query.

The outcome of *prefix sum to memory* is similar to the prefix sum instruction, but is not as efficiently implemented. Instead of a global register, it uses a memory address as a base variable. Prefix-sum-to-memory operations are used when several TCUs write to the same memory address. These multiple prefix-sum-to-memory operations are queued at the memory to prevent concurrent write.

Nesting spawn instructions is not allowed. In parallel mode, nesting can be achieved by the *single spawn* instruction which creates a new XMT virtual thread to be executed by a TCU. This instruction also allows greater flexibility for dynamically extracting inter-thread parallelism during execution time. In the future, the compiler should be able to translate nested spawn instructions to nested single spawn instructions.

### 3.2. XMT-C Programming

XMT-C is a high level language that is an extension of the ANSI C language. It adds four unique XMT-C instructions: spawn, prefix sum, prefix sum to memory and single spawn. It also includes a new type *psBaseReg* that is used to declare a global register variable. The XMT-C program is compiled into XMT assembly and run on the XMT hardware or simulator.

The XMT-C programmer (explicitly) provides the parallel segments. Every parallel segment is encapsulated within a spawn block that begins with a spawn instruction. The parameters of the spawn instruction are: (i)low–the lowest XMT virtual thread ID to be executed, (ii) high–the highest XMT virtual thread ID, and (iii) a varied number of values to be passed (broadcasted in hardware) to the virtual threads. The spawn instruction starts a parallel execution during which high-low+1 virtual threads with different IDs in the range [low, high] are executed by the TCUs. Inside a spawn block, "$" represents the thread ID (TID) of a virtual thread and is the only difference among virtual threads, who all execute the same parallel segment.

The following is a simple XMT program segment that performs the back-face culling task of the graphics pipeline. The input is an array of triangles $T1[n]$, an array of triangle normals $N1[n]$ and the front direction $Front$. This program culls the back-facing triangles and compacts the front-facing ones into two new arrays $T2[n]$ and $N2[n]$ in an arbitrary order.

```
psBaseReg m = 0;
spawn(0, n − 1);
{
  int TID;
  if (dot(N[TID], Front) > 0)
  {  registerint k = 1;
     ps(&k, &m);
     B[k] = A[TID];
     M[k] = N[TID];
  }
}
```

In the above program, $n$ threads are spawned with TIDs in the range $[0..n-1]$, and executed in arbitrary order. One (explicit) prefix sum operation *ps* ensures that the TID numbers are unique.

### 3.3. Comparing XMT and Stream Architecture

XMT and stream architecture have important similarity and significant differences. We summarize them in three aspects.

**Design Philosophy:** Both architectures achieve fine-grained on-chip parallelism. They take the benefit of VLSI technology and provide large number of arithmetic units (AUs) and registers. However, they follow different philosophy in organizing these resources. The stream architecture sacrifices random memory access and arbitrary looping and branching for performance. XMT keeps these very useful capabilities and reduces performance lost by other mechanisms.

The stream architecture doesn't encourage frequent memory access and branching. Random memory access and arbitrary branching is not supported inside a kernel, which is a program segment executed on the arithmetic units. The stream architecture instead uses input/output streams to load data from memory to the stream register file (SRF). An arithmetic unit can only randomly access a stream register file which is faster and smaller than the L1 cache. Branching are usually supported only between kernels through some mechanisms like conditional streams in [Owe02].

This feature allows very fast kernel executions with short clock cycle time and without stalls. But it limits the efficiency of global data access which is frequently used in texture lookup, in realistic global lighting effect calculation or any form of interaction between objects.

To support arbitrary looping and branching, the XMT architecture inherently has more logic control units than the stream architecture. In the XMT processor discussed in [NNTV03], each cluster contains two branch units. To support fast random memory access, the XMT architecture needs a significant amount of interconnection between clusters and memory banks. It uses a memory address hashing table to keep memory access balanced. Compared with the stream architecture, XMT sacrifices more chip space for functional flexibility.

XMT works efficiently for tasks with heterogeneous data streams. For example the output of the rasterization process may contain zero to hundreds of fragments for each triangle. XMT begins to rasterize one of the triangles on each TCU. If a triangle is small and finishes earlier than others, the TCU will automatically rasterize one of the remaining triangles.

XMT tries to hide memory access latency and branching delay by switching to execute on another virtual thread whose data is already cached on its host TCU.

**Programming Model:** Both architectures provide a instruction set that requires the programmer or the compiler to design and specify parallelism explicitly. They differ in how the parallelism is specified.

Many difficulties arise when mapping CPU algorithms to stream algorithms. This is complicated by the graphics hardware implementation of stream architectures, with many

short kernels in a system that can write globally accessible memory (in the form of texture) only between rendering passes. Algorithms with global data access and complex branching patterns require creative mapping to multiple passes, textures, and other aspects of the graphics pipeline [KSW04, MA03, JBSL03, GHLM05].

XMT-based architecture provides a clear and easy mapping from CPU program to XMT program. Though all the threads share the same code, at run-time different threads may have different lengths, based on control flow decisions made at run time. The gobal memory access and branching operations are much more similar, in use and performance, to what we have learned to expect from serial CPUs.

**Hardware Organization:** Both architectures group functional units into clusters. Each cluster contains an identical set of functional units. Clusters provide data level parallelism in both architectures. But the XMT and stream processors have different structure within a cluster. In stream processors, functional units provide instruction level parallelism. They operate on the same set of data and may have to stall to avoid data dependencies. No memory access latency exists unless texture lookup is needed. In XMT, however, functional units executes on the data of different TCUs. XMT stores the execution context of several virtual threads on several TCUs of each cluster. When a memory fetch or branching is needed, the functional units switch to process another thread whose context is stored in a TCU. So memory access latency is hidden by data level parallelism.

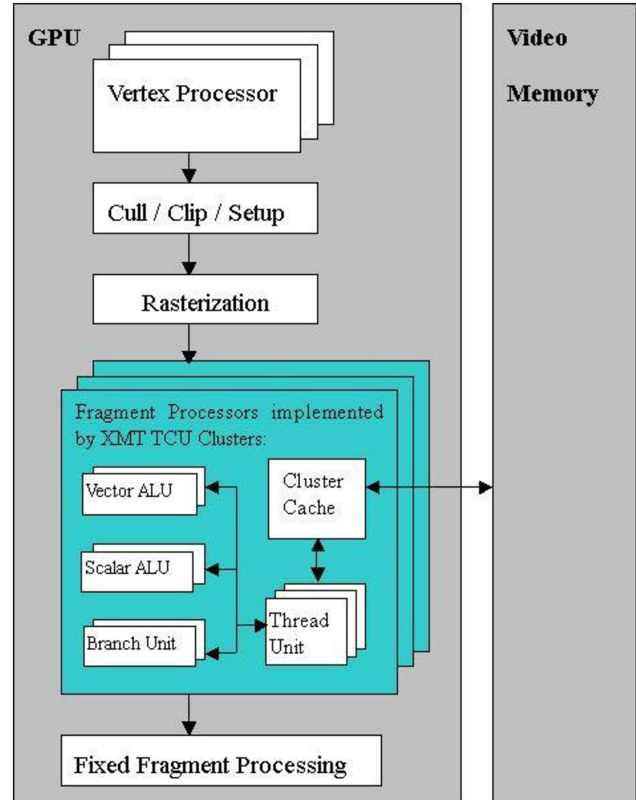


**Figure 2:** *GPU Architecture with XMT-Based Fragment Processor*

## 4. Evaluation and Applications

We evaluate the XMT-based architecture in four aspects: computation capability, programming difficulty, performance and hardware utilization.

### 4.1. Implement Fragment Processor With XMT

The XMT-based fragment processing architecture is shown in Figure 2. The vertex processor could also be replaced with XMT, or with a chain of vertex and geometry processing XMT processors.

To evaluate XMT's fragment processing capability, we built a test bed based on Mesa, a software implementation of OpenGL. We replaced the fragment processing part of Mesa with XMT-C programs. The shaders written in XMT-C are processed by the XMT compiler and assembler, for execution on an XMT simulator.

In current implementation, we specify 8 TCUs in each TCU cluster. Each cluster contains 4 integer ALUs, 2 integer multiply/divide units, 2 floating point adders, 2 floating point multiply/divide units and 2 branch units. All functional unit latencies are set to the SimpleScalar sim-outorder defaults: integer divide, multiply and ALU ops take 20, 3 and 1 cycles respectively, floating point divide, multiply and addition ops take 12, 4 and 2 cycles respectively, and square root takes 24 cycles. Each cluster has a L1 cache of 8 KB. The L2 cache is composed of shared independent banks which total 1 MB. The number of banks is chosen to be twice the number of clusters. The L2 cache latency within each cache bank is 6 cycles and memory latency is 25 cycles. A penalty of 4 cycles is charged each way for every one-way traversal of the interconnection network. Each cluster can issue a single memory request per cycle, but multiple requests may be inflight at once. For example, a single cluster may have 8 loads from 8 different threads being serviced at once.

### 4.2. Computation Capability

Even with some degree of programmability, current graphics hardware still can't render scenes with as good fidelity as CPUs do. Users either render their scene on graphics hardware and lose some interesting effects or render fully-effected image on the CPU with much longer time. For example, most animation products are still rendered totally on the CPU. The random memory access and arbitrary looping and branching capability enables the XMT-based architecture as a powerful graphics engine.

When implementing the fragment operations with XMT-C, we spawn a thread for the operations for each fragment and join the operations that are common for fragments on a whole span (one scan line of a primitive) or on a whole primitive. We implemented several more complex fragment shading programs in XMT-C. Here we discuss how they mapped to XMT.

(1) Figure 3 shows the sketch of a subsurface scattering or volume ray-tracing shader. Notice the scatter write operation in the parallel program segment. This shading program is hard to implement on current fragment processors even in multiple rendering passes.

```
First Pass:
   //scatter light into voxels
   //render from the viewpoint of light source
   spawn(for every fragment)
        calculate intersection with volume
        while (light energy > epslon && intersection)
                lookup voxel in 3D texture
                store obsorbed light energy into voxel
                calculate reflection light //can fork new light here
                calculate intersection with volume
   join()

Second Pass:
   //gather light along view direction
   //render from camera's viewpoint
   spawn(for every fragment)
        while (accumulated opacity < 1)
                lookup lighted voxels in 3D texture
                accumulate light*(1-opacity) to fragment color
   join()
```

**Figure 3:** *Sketch of the Subsurface Scattering Example*

(2) Figure 4 shows the sketch of a shading program that uses varying materials over a surface. The execution length of each thread is only as long as the branches taken.

```
spawn(every fragment)
    texel = texture(texcoord, image1)
    if (texel == 1)
          use texture 1
          use shading model 1
    else if (texel == 2)
          use texture 2
          use shading model 2
    else
          use texture 3
          use shading model 3
join()
```

**Figure 4:** *Sketch of the Varying Material Example*

(3) Subdivision surfaces. Other than fragment shading operations, XMT can supports more operations efficiently. One example is subdivision surfaces. Users can specify high-level smooth surfaces with a small number of control points. The GPU subdivides these surfaces into pixel-sized subsurfaces [CC78, Loo87, HDD*94]. The stream processors cannot control the subdivision steps adaptively because they lack branching units. Recent graphics hardware [PF05] can branch in vertex shader, but they are unable to add new vertices inside the shader. It is also difficult for the stream processors to access global information of other surfaces to prevent surface cracks.

### 4.3. Programming Difficulty

Since XMT provides a compatible programming model with a serial CPU's, programming in XMT is not hard. The only extra work is to find the parallelism between data and specify it by the spawn instruction instead of loop. Failing to specify parallelism will lead to lower performance, not error.

Yet there are two constraints to be considered in XMT programming: the TCU register count and parallel instruction count. Parallel program segments still run correctly if they use more registers than TCU's local ones, but the performance will drop significantly, since the XMT-C compiler will assign global memory space to store intermediate results.

In the current XMT-C implementation, parallel instruction count (that portion between each spawn and its corresponding join) is not allowed to exceed a limit. We expect future versions of the XMT-C compiler to automatically split long parallel segments into shorter ones. This differs significantly from automated splitting into multiple rendering passes [FHH04, RLV*04]; the program never leaves the fragment portion of the pipeline, conserving other pipeline resources, and there is no arbitrary limit on the number of temporaries used to communicate between parallel segments.

### 4.4. Performance Evaluation

Some useful algorithms have been implemented in XMT-C and run on a XMT simulator. [NNTV03] analyzed the XMT performance in detail. We briefly discuss the experiment result in this paper.

Using the XMT architecture introduced in Subsection 4.1, we have experimented with 1, 4, 16, 64 and 256 TCUs. Our conclusion is that XMT can achieve significant speedup on both regular and irregular computations. XMT is especially fast on regular computations with predictable access patterns.

Graphics applications contain a significant amount regular computations with predictable memory accesses. These computations do the same amount of work on a set of independent elements. Figure 5 shows the speedup of regular computations to a serial CPU.

We often meet some irregular computation problems in graphics applications. One example is ray-tracing. The ray shot from a pixel may be bounced zero to a large number of times. Figure 6 shows the speedup of irregular computations to a serial CPU.

Part of the performance drop of irregular computations comes from thread managing overhead. We break down this overhead into three parts: 1) Spawn-Setup: setting up

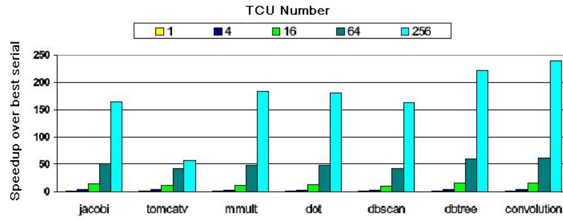


**Figure 5:** *XMT Performance on Regular Computations.*

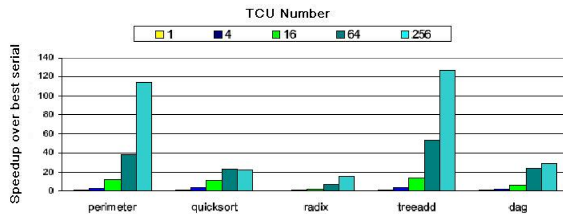


**Figure 6:** *XMT Performance on Irregular Computations.*

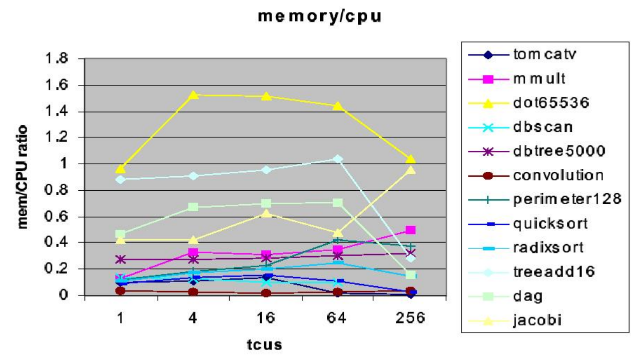


**Figure 7:** *Memory/CPU Time Ratio*

the parallel execution environment and broadcasting data to TCUs. 2) TCU-Init: initializing the TCUs context. 3) Load-Imbalance: idling at the end of a spawn until all threads complete, then transitioning back to serial mode. In our prototype machine, a single Spawn-Setup needs 6 instructions and costs 150-300 cycles. A typical TCU-Init is 25 instructions and costs 150-260 cycles. The Spawn-Setup cost and TCU-Init cost are generally insignificant ( typically less than 10% of total execution time) according to our experiment results. The effect of Load-Imbalancing depends on the number of threads and the number of synchronizations. Applications with large number of threads and few synchronizations have small penalty.

Memory stall is an important performance factor. Figure 7 shows the Memory/CPU ratio of several applications. Our experiment also shows that the ratio of time spent on waiting memory to time spent on processing is largely constant as we increase the number of TCUs from 1 to 256.

In an XMT-based architecture, when TCU local registers can't hold all the intermediate data of a virtual thread, a large array is allocated in memory. Each array element is used to hold a virtual thread. Saturated memory access will lead to more than 5 times longer execution time.

### 4.5. Hardware Utilization

Because XMT's instruction set is compatible with current CPU's serial computation model, XMT-based GPU has big potential to support general-purpose computation. Computation intensive CPU tasks, e.g. matrix multiplication, FFT, sorting etc., can be migrated onto the GPU when the GPU is idle. Since in most non-graphics applications the GPU is less busy than the CPU, an XMT-based GPU has the potential to be used more often and more effectively for general computation.

## 5. Conclusion and Future Work

The XMT-based architecture shares a similar design philosophy as the stream computing model. In addition, it provides random memory access and arbitrary branching capability. Because of its flexibility, it can support more powerful shaders than current graphics hardware. XMT-C programming language is compatible with the programming language of serial CPUs. Compared with stream computing languages, it is easier to write complex programs and easier to compile.

In XMT-based architecture, the parallelism is limited by two factors: the TCU register count and parallel instruction count, though both could be hidden from the programmers by compilers.

In the next phase of our research, we will compare the rendering performance as well as fidelity between XMT-based fragment processor and other graphics hardware.

We dump the OpenGL context and the fragment data before they arrive at the programmable part (i.e. texturing and fogging etc. [KBR04]) of the fragment processing pipeline. These data are used as the input of XMT-C shaders. After XMT-C shaders process these data, we will feed them back to Mesa's fragment processing pipeline and render out final image.

We would also like to explore the use of XMT for the full graphics pipeline. This may lead to an attractive alternative to the fixed pipeline layout of current graphics hardware and allow users to balance between performance and fidelity.

## 6. Acknowledgements

This research is supported by the National Science Foundation under grants No. 0339489 and 0325393.

**References**


[Baa88] BAASE S.: *Computer Algorithms: Introduction to Design and Analysis. Second edition.* Addison-Wesley, 1988.

[CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. In *Computer-Aided Design* (September 1978), vol. 10(6), pp. 350–355.

[CLR90] CORMEN T. H., LEISERSON C. E., RIVEST R. L.: *Introduction to Algorithms*. MIT Press, 1990.

[FHH04] FOLEY T., HOUSTON M., HANRAHAN P.: Efficient partitioning of fragment shaders for multiple-output hardware. In *Graphics Hardware* (2004), SIGGRAPH/Eurographics, pp. 35–44.

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering and transparancy computations among geometric primitives in complex environments. In *Proc. ACM SIGGRAPH Symposium on interactive 3D graphics and games* (2005), pp. 49–56.

[HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., MCDONALD J., SCHWEIZER J., STUETZLE W.: Piecewise smooth surface reconstruction. In *Proc. SIGGRAPH* (1994), ACM Press, pp. 295–302.

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *Proc. of SIGGRAPH* (August 1990), pp. 289–298.

[JBSL03] JHARRIS M. J., BAXTER W. V., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware* (2003), Eurographics Association, pp. 92–101.

[KBR04] KESSENICH J., BALDWIN D., ROST R.: *THE OPENGL SHADING LANGUAGE. Language. Version 1.10.* OpenGL Architecture Review Board, 2004.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: A GPU-based particle engine. In *Graphics Hardware* (2004), ACM SIGGRAPH/Eurographics, ACM Press.

[Lei92] LEIGHTON F.: *Introduction to Parallel Algorithms and Architectures: array, trees, hypercubes*. Morgan Kaufmann, 1992.

[LKM01] LINDHOLM E., KILGARD M. J., MORETON H.: A user-programmable vertex engine. In *Proc. SIGGRAPH* (2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 149–158.

[Loo87] LOOP C.: *Smooth subdivision surfaces based on triangles*. Master's thesis, Department of Mathematics, University of Utah, August 1987.

[MA03] MORELAND K., ANGEL E.: The FFT on a GPU. In *Graphics Hardware* (July 2003), Eurographics Association, pp. 112–119.

[Man89] MANBER U.: *Introduction to algorithms: a creative approach*. Addison-Wesley, 1989.

[Mic02] MICROSOFT: *DirectX 9.0 Programmer's Reference*. Microsoft Corporation, 2002.

[Mic05] MICROSOFT: *Microsoft Developer Network (MSDN)*. Microsoft Corporation, 2005.

[NNTV00] NAISHLOS D., NUZMAN J., TSENG C.-W., VISHKIN U.: Evaluating multi-threading in the prototype XMT environment. In *Proceeding of 4th Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC2000)* (2000).

[NNTV03] NAISHLOS D., NUZMAN J., TSENG C.-W., VISHKIN U.: Towards a first vertical prototyping of an extremely fine-grained parallelprogramming approach. In *Special Issue of SPAA2001: TOCS 36,5* (2003), Springer-Verlag London, UK, pp. 521–552.

[NVD03] NVDIA: *Cg Toolkit User's Manual*. NVIDIA Corporation, 2003.

[OL98] OLANO M., LASTRA A.: A shading language on graphics hardware: the PixelFlow shading system. In *Proc. SIGGRAPH* (1998), ACM Press, pp. 159–168.

[Owe02] OWENS J. D.: *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, Nov. 2002.

[PF05] PHARR M., FERNANDO R.: *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005.

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *Proc. SIGGRAPH* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 425–432.

[RLV*04] RIFFEL A., LEFOHN A., VIDIMCE K., LEONE M., OWENS J. D.: Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware* (2004), SIGGRAPH/Eurographics, pp. 35–44.

[RTB*92] RHOADES J., TURK G., BELL A., STATE A., NEUMANN U., VARSHNEY A.: Real-time procedural textures. In *Symposium on Interactive 3D Graphics* (March 1992), Zeltzer D., (Ed.).

[SA04] SEGAL M., AKELEY K.: *The OpenGL Graphics System: A Specification (Version 2.0)*, September 2004.

[Ups90] UPSTILL S.: *The RenderMan Companion*. Addison-Wesley, 1990.

[VDBN98] VISHKIN U., DASCAL S., BERKOVICH E., NUZMAN J.: Explicit multi-threading (XMT) bridging models for instruction parallelism. In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (1998).