January 2007

Logic Simulation Using Graphics Processors

Atchuthan S. Perinkulam
University of Massachusetts Amherst, atchuth@gmail.com

LOGIC SIMULATION USING GRAPHICS PROCESSORS

A Thesis Presented

by

ATCHUTHAN S. PERINKULAM

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

September 2007

Master of Science in Electrical and Computer Engineering

LOGIC SIMULATION USING GRAPHICS PROCESSORS

A Thesis Presented

by

ATCHUTHAN S. PERINKULAM

Approved as to style and content by:

_____
Sandip Kundu, Chair

_____
Wayne P. Burleson, Member

_____
Ramgopal Mettu, Member

_____
C.V.Hollot, Department Head
Department of Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank my advisor, Sandip Kundu, for his thoughtful and patient guidance, motivation and support. I would also like to extend my gratitude to the members of my committee, Wayne P. Burleson and Ramgopal Mettu, for their helpful comments and suggestions on all stages of this thesis.

I would like to thank Ian Buck and his team at Stanford University whose material I have used, excerpted and referenced in this thesis. The members of the GPGPU (General Purpose Computation using Graphics Hardware) forums, particularly Mike Houston and the Brook GPU community at Stanford University deserve appreciation. The forums were an invaluable source of information and helped clear many concepts. The GPU Gems 2 book was an extremely useful reference.

A special thank you to all those, whose support and friendship helped me to stay focused on this project and provided me with constant encouragement. Thanks are also due to my family for their support, love and affection.

ABSTRACT

LOGIC SIMULATION USING GRAPHICS PROCESSORS

SEPTEMBER 1, 2007

ATCHUTHAN S. PERINKULAM,

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sandip Kundu.

Logic Simulation is widely used to verify the logical correctness of hardware designs. In this work, we present the implementation of a generic graphics processor based logic simulator and compare it with the corresponding CPU (desktop) based implementation. The motivation for this study arises from the increasing computational power of graphics processors (GPUs). Graphics hardware performance is roughly doubling every six months, and they are outpacing CPUs in raw computational power. GPUs are becoming increasingly programmable and their prices are falling steeply. Most desktops now come built-in with programmable graphics processors. The highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving higher arithmetic intensity with the same transistor count. Applications such as Ray Tracing, Fluid Modeling, Radiology imaging etc have shown speed-ups on graphics processors. This led us to investigate the use of GPUs to run concurrent algorithms for logic simulation. We present the implementation and analyze performance bottlenecks and finally draw conclusions as to whether the GPU can be used for speeding up the logic simulation algorithm.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Today's electronic systems are incredibly complex and hence very expensive. Any design defect would lead to not only huge monetary losses, but also significantly reduce the competitive advantage of the company designing and building the system. Also, as the complexity of the system increases, the design process is broken down into separate modules that are designed and verified, built and re-tested to rule out manufacturing defects and then integrated to form the complete system. The "Correct-by-construction" methodology is frequently used in the design process.

Logic Simulation

Logic Simulation is defined as the use of a computer program to simulate the operation of a digital circuit. It is used for verifying the logical correctness of hardware designs. Improving the performance and speed of logic simulators has many benefits, such as decreasing overall production time and speeding up debugging and processing schedules. More test cases can be run in shorter time and the associated benefits are enormous. This work focuses on logic simulation using graphics processors. The motivation for this study is the increasing programming power and affordability of graphics processors. Many applications such as Radiology imaging, Fluid modeling etc have shown speed-ups when mapped to graphics processors and we wanted to study if a similar speed-up could be achieved in the case of logic simulation (GPGPU). In this work, we have developed, implemented and tested a logic simulation algorithm that is generic and can be applied to any combinational or sequential circuit and any

programmable graphics processor, that supports high level programming languages such as those described in this material. Results are presented for four sets of publicly available standard benchmarks namely the ISCAS85, ISCAS89, PICOJAVA and the BEAST benchmarks.

Types of logic simulation

There are 2 primary types of logic simulation, 1) compiled code logic simulation and 2) event driven logic simulation. A very brief overview is presented here. A highly detailed description of the various methods is presented in (Meister, 1993). The current work focuses on compiled code logic simulation. Event driven simulation tries to eliminate the "execution time wastage" of compiled code logic simulation, by using time-stamping. To exploit parallelism using MIMD (Multiple Instruction – Multiple Data) machines, various data access and distribution methods were developed such as full replication with distributed data, partitioning, remote data access from a central repository etc. Very fast and expensive hardware accelerators and Emulators such as the Quickturn System M3000 series are also available. While many techniques have been developed to speed-up logic simulation, the problem of finding a generic optimal logic simulation algorithm remains NP-complete.

CPU based Logic Simulation

Currently, logic simulators are run on CPUs. In this work, we focus on a desktop GPU based logic simulator. The logic simulation programs/algorithms designed for the CPU are usually serial in nature, leading to large simulation run-times for large circuits. Typically, such an algorithm would proceed by first levelizing the circuit and then

computing the output of one node at a time, until all individual node output values are obtained. However, many algorithms such as audio and signal processing, ray tracing, radiology imaging etc. have been successfully mapped onto the Graphics Processing Units (GPUs) with higher performance benefits(GPGPU), leading us to the study of logic simulation on GPUs.

Graphics Processors (GPUs)

The interaction between humans and computers has evolved a long way from teleprinters to using the monitor, or the visual display unit. Graphics processors or GPUs perform the translation of binary data into pixels (picture elements) that are displayed on the screen. A pixel is one of the tiny dots that make up the entire image. The steps required to display an image on the screen are:

1) Creating a wire frame out of straight lines
2) Rasterizing the image ie filling in the remaining pixels
3) Adding lighting, texture and color
4) Performing a refresh about 60 times a second.

This process is very intensive, and earlier, CPUs performed all the required computations. Highly specialized processors called Graphics Processing Units were later developed to perform these functions effectively, so that they could take over the display functionality from the CPU. With the advancement of technology, we see that nowadays, computers and workstations are generally shipped with sophisticated graphics cards and gaming technology, three dimensional effects and processing power that was virtually impossible a few decades ago.

GPU based logic simulation

An emerging trend is the usage of commodity graphics processing units (GPUs) for general purpose graphics applications. In 1965, Gordon Moore from Intel observed that the number of transistors that could be economically fabricated on a single transistor die was doubling every year and that this trend was likely to continue in the future.

Following this trend, graphics processors (and silicon chips in general) started becoming increasingly programmable, powerful and capable co-processors. Also, their prices have become affordable. With this increasing functionality, programmers started using GPUs for general purpose applications such as Ray Tracing, Fluid modeling, Radiology imaging etc. This created a new area of research and development, known as GPGPU, "General Purpose Computation using Graphics Processing Units" (GPGPU)

A recent survey on general purpose computation on graphics architecture by John D. Owens et al. states that, "In general, the computational capabilities of GPUs have compounded at an average yearly rate of 1.7× (pixels/second) to 2.3× (vertices/ second). This rate of growth outpaces the oft-quoted Moore's Law as applied to traditional microprocessors; compared to a yearly rate of roughly 1.4× for CPU performance. Put another way, graphics hardware performance is roughly doubling every six months. The highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving higher arithmetic intensity with the same transistor count." As an example, the NVIDIA GeForce 3 series had 57 million transistors, while the NVIDIA GeForce 7800GTX has about 300 million transistors.

A closer look at logic simulation reveals that the algorithm can be modified and implemented on parallel architectures. This leads us to investigate the use of GPUs to run concurrent algorithms for logic simulation.

Why use the GPU for computation?

As stated earlier, GPUs are becoming increasingly affordable and programmable. “Economics and the rise of video games as mass-market entertainment have driven down prices to the point where you can now buy a graphics processor capable of several hundred billion floating-point operations per second for just a few hundred dollars.”, as observed by Simon Green of NVIDIA Corporation. However, one must not assume that all problems will map well and result in a performance speed-up when implemented on the GPU. The GPU is a special processor and was not originally designed for performing general purpose computations. Every time data is transferred from the CPU to the GPU and back, there are associated delays.

For a fruitful implementation, the number of computations performed on the GPU must be much higher than the data transfer rate. This is called as “high arithmetic intensity”. The advantage of formulating an efficient algorithm for logic simulation on the GPU is that it removes the need for an additional hardware/software emulator and comes with no added price.

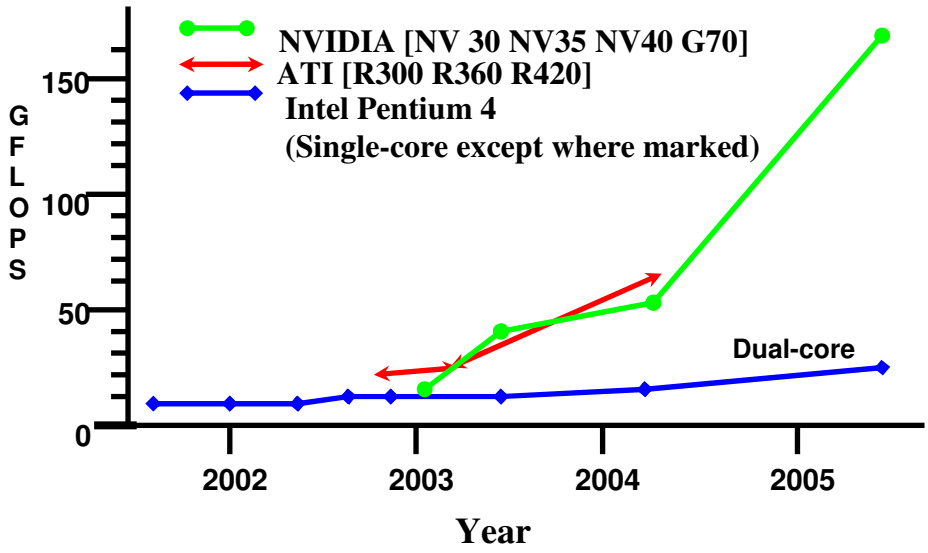

Figure 1: CPU vs GPU Floating point performance comparison (Courtesy I. Buck)

Figure 1 shows that the, "programmable floating point performance of GPUs (measured on the multiply-add instruction as 2 floating operations per MAD) has increased dramatically over the last four years when compared to CPUs." (Owens et al, 2007).

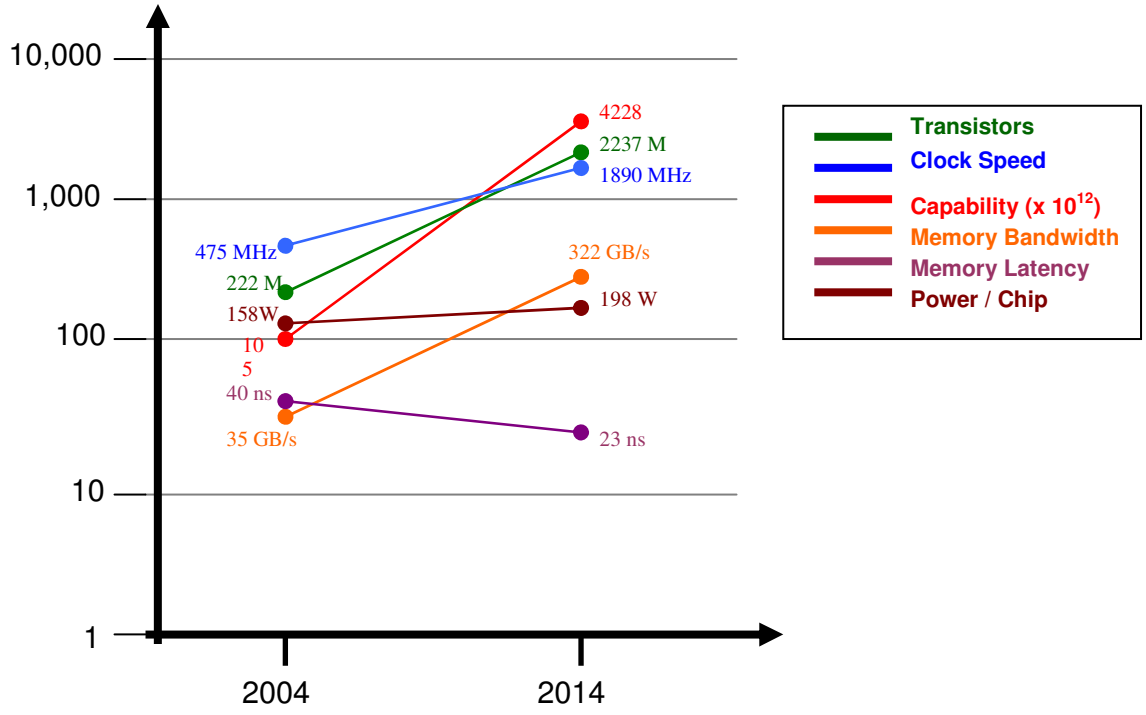

Figure 2: Changes in Key GPU properties over time

(Courtesy, Ian Buck, Stanford University, Ch 29, GPU Gems 2, NVIDIA Corporation).

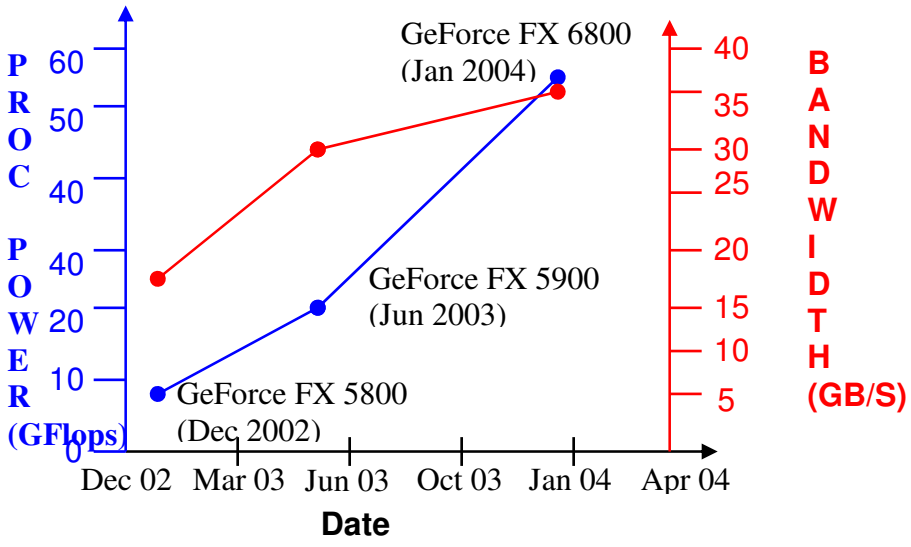

Figure 3: Rapidly Changing GPU Capabilities
(Courtesy Ian Buck, Stanford University)

The number of observed floating-point operations per second on the GeForce FX 5800, 5950, and the GeForce 6800 has been growing at a rapid pace, while off-chip memory bandwidth has been increasing much more slowly.

Keys to High Performance Computing

This section provides a summary of various techniques used for high performance computing as discussed in Ch. 29, GPU Gems 2. As discussed earlier, technology advancements result in enormous computation power in the form of transistors. We can make optimum use of these resources by providing efficient communication techniques and by allowing them to operate in parallel. The usage of transistors is broadly divided into three categories, the control, the data-path and the storage. For maximum efficiency, we must maximize the performance of those transistors in the data-path devoted to performing computations.

CPUs have more transistors devoted to control hardware as they have more complex control requirements (such as branch prediction and out of order execution) as compared to GPUs, leaving only a small fraction devoted to computation. Also, as CPUs are general purpose processors, they do not have the specialized hardware found in GPUs. CPUs are dedicated to minimizing latency (by using several layers of caches), GPUs try to maximize throughput. This makes CPUs inefficient for processing data that is accessed only once, as is the case, in parallel algorithms. The GPU has a huge performance advantage in this regard.

Focus

The objective of this work is to study the implementation of a generic GPU based logic simulator and compare it with the corresponding CPU (desktop) based implementation. Results are presented for four sets of publicly available standard benchmarks namely the ISCAS85, ISCAS89, PICOJAVA and the BEAST benchmarks. The benchmarks span a wide array of circuits ranging from sequential to combinational, structurally sparse to dense and hence, form a reasonably good study set for analysis. The effects of technology mapping and circuit restructuring are studied and analyzed in detail. A comprehensive test suite is built for comparing the benchmarks, on different platforms. The methods presented in this study are benchmark independent and can be used for any kind of circuit as mentioned earlier. Also, any kind of programmable graphics processor may be used. The results will vary and are subject to the graphics card and platform performance constraints.

CHAPTER 2

GPU ARCHITECTURE AND PROGRAMMING MODEL

This chapter summarizes the evolution of the GPU and describes its current architecture and trends. Today's entire commodity GPUs structure their graphics computation in the organization shown below, called the graphics pipeline. The graphics pipeline is designed so that the throughput is maximized. High computation rates are achieved by incorporating parallelism.

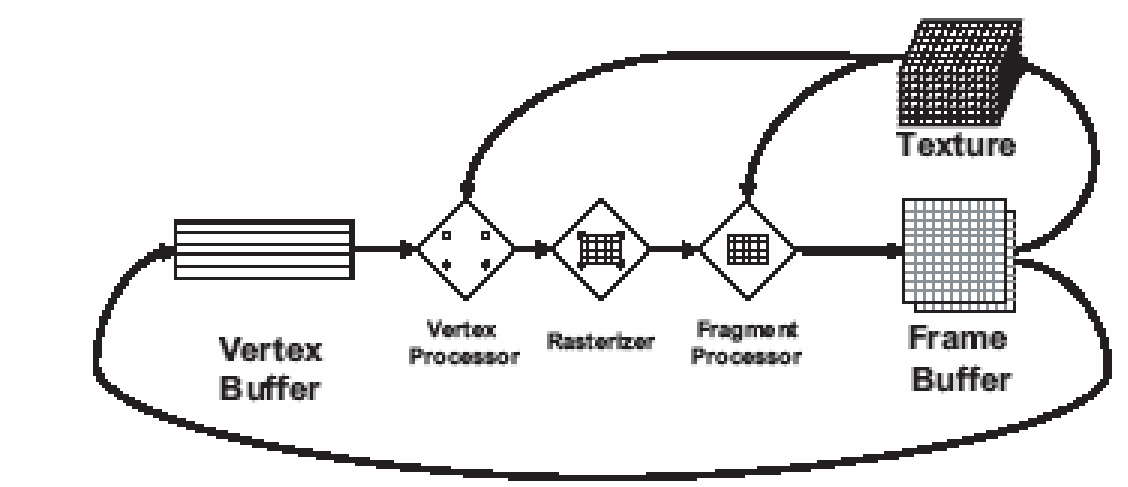

Figure 4: The modern graphics pipeline with programmable vertex and fragment processors.

(Picture adapted from "A Survey of General Purpose Computation on Graphics Hardware", by Owens et al, 2007).

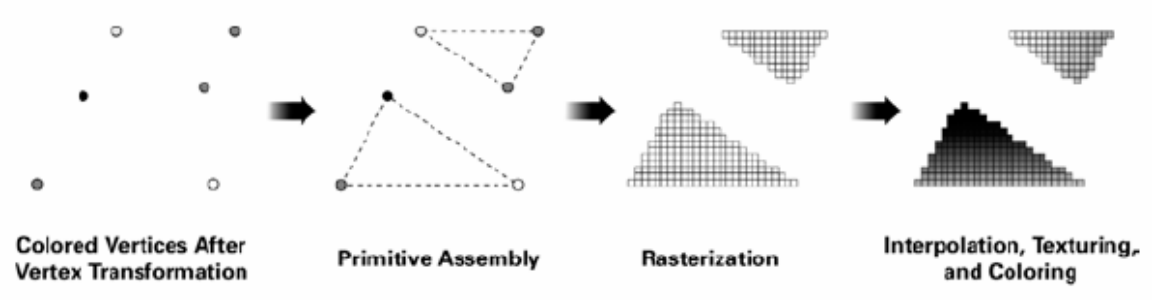

Figure 5: Visualizing the graphics pipeline

(Courtesy, "The Cg Tutorial", NVIDIA Corporation- Figure 1.6).

. The above two figures demonstrate the architecture of the graphics pipeline and the functionality of its various components. All geometric primitives pass through each stage namely vertex operations, primitive assembly, rasterization, fragment operations and composition into final image. A **vertex** is a data structure for a point in a mesh, containing position, normal, texture coordinates etc. A **fragment** refers to a tentative pixel (picture element) or a sub-pixel, of a rasterized image.

GPUs have evolved from the earlier generation fixed function processors, where the vertex and fragment processors did specialized tasks such as vertex transformations, lighting calculations (vertex processor) and color determination (fragment processor), to programmable vertex and fragment processors. These new processors can now execute vertex and fragment programs, called as **shaders** that are written by the user. The process of writing to a memory address is termed as a scatter operation, while that of reading from a memory address is a gather operation. Typically, fragment processors have the capability to fetch data from textures i.e. gather operations, but not scatter operations. Vertex processors are capable of scatter operations and have limited gather capability.

Rasterization Process

Given a set of rays and a primitive, rasterization is defined as the process of efficiently computing the set of rays that hit the primitive.

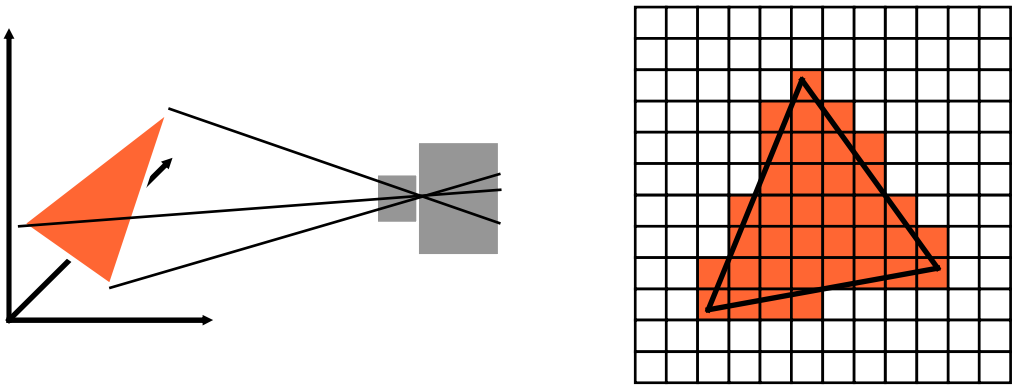

Figure 6: The Rasterization process
(Courtesy Slusallek et al, SIGGRAPH 2005)

The process is described below:

- The input to the rasterizer is a stream of vertices that define the scene.
- The stream of transformed vertices is assembled into a stream of triangles, each triangle keeping the attributes of its three vertices. This operation is called as clipping.
- Each triangle from this stream then passes through a rasterizer that generates a stream of fragments, which are discrete portions of the triangle surface that correspond with the pixels of the rendered image. Fragment attributes are derived from the triangle vertex attributes. This stream of fragments may pass through a number of stages performing a number of visibility tests (stencil, depth, alpha and scissor) that will remove non-visible fragments and then the stream of fragments will pass through a second computation stage.

Having discussed the various stages in the pipeline and the rasterization process, let us look at the architecture of the GeForce 6 series of graphics processors from NVIDIA corporation.

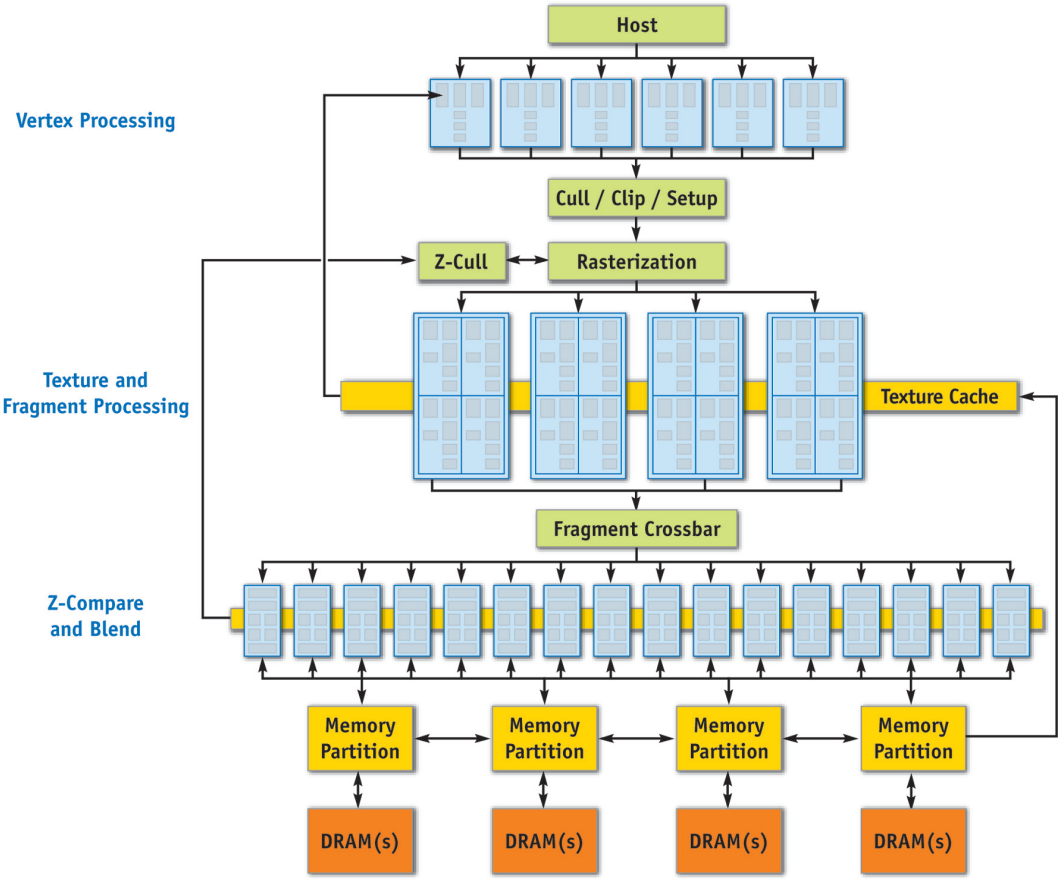

Figure 7: Block diagram of NVIDIA GeForce 6 series architecture

(Excerpted from Chapter 30, GPU Gems 2, NVIDIA Corporation 2005)

A detailed explanation of the various stages of this processor can be found in the technical specifications, available in the NVIDIA web site, or in the book GPU Gems 2. This figure helps us to understand the parallelism that is built into the processor at the architectural / hardware level, as opposed to a general purpose central processing unit. Since the vertex and fragment processors are programmable, it is also understandable that applications that will show performance improvements over their CPU counterparts

will be typically data-parallel. Also, we can visualize the GPU architecture as a set of SIMD machines working in unison.

GPU Programming Model

GPUs are a compelling solution for applications that require high arithmetic rates and data bandwidths. GPUs achieve this high performance through data parallelism, which requires a programming model distinct from the traditional CPU sequential programming model. This programming model is defined as the "Stream Programming Model".

Because typical scenes have more fragments than vertices, in modern GPUs the programmable stage with the highest arithmetic rates is the fragment stage. A typical GPGPU program uses the fragment processor as the computation engine in the GPU.

Structure of a GPU program

The structure of a typical GPU program is explained by Mark Harris as follows:

1. First, the programmer determines the data-parallel portions of his application. The application must be segmented into independent parallel sections. Each of these sections can be considered a **kernel** and is implemented as a fragment program. The input and output of each kernel program is one or more data arrays, which are stored (sometimes only transiently) in textures in GPU memory. In stream processing terms, the data in the textures comprise **streams**, and a kernel is invoked in parallel on each stream element.

2. To invoke a kernel, the range of the computation (or the size of the output stream) must be specified. The programmer does this by passing vertices to the GPU. A typical

GPGPU invocation is a quadrilateral (quad) oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array. Note that GPUs excel at processing data in two-dimensional arrays, but are limited when processing one-dimensional arrays.

3. The rasterizer generates a fragment for every pixel location in the quad, producing thousands to millions of fragments.

4. Each of the generated fragments is then processed by the active kernel fragment program. Note that every fragment is processed by the same fragment program. The fragment program can read from arbitrary global memory locations (with texture reads) but can only write to memory locations corresponding to the location of the fragment in the frame buffer (as determined by the rasterizer). The domain of the computation is specified for each input texture (stream) by specifying texture coordinates at each of the input vertices, which are then interpolated at each generated fragment. Texture coordinates can be specified independently for each input texture, and can also be computed on the fly in the fragment program, allowing arbitrary memory addressing.

5. The output of the fragment program is a value (or vector of values) per fragment. This output may be the final result of the application, or it may be stored as a texture and then used in additional computations. Complex applications may require several or even dozens of passes ("multipass") through the pipeline. (Harris, 2005)

Having understood the structure of a typical GPU program, we will discuss some of the GPU programming languages being used currently.

GPU Programming languages

Just as there are many CPU programming languages such as Basic, C, C++, Java etc, there is a wide array of GPU programming languages such as Cg, HLSL etc. A shader is a program that is executed on the graphics processing unit. Shaders are of two types, vertex shaders and pixel or fragment shaders. A brief summary of each language is presented below:

ARB low-level assembly language

The OpenGL Architecture Review Board established ARB (GPU assembly language) in 2002 as a standard low-level instruction set for programmable graphics processors. High-level OpenGL shading languages compile to ARB for loading and execution. Unlike high-level shading languages, ARB assembly does not support flow control or branching. However, it continues to be used for portability to a variety of GPUs.

C for graphics (Cg)

As a result of technical advancements in graphics cards, some areas of 3D graphics programming have become quite complex. To simplify the process, new features were added to graphics cards, including the ability to modify their rendering pipelines using vertex and pixel shaders.

In the beginning, vertex and pixel shaders were programmed at a very low level with only the assembly language of the graphics processing unit. Although using the

assembly language gave the programmer complete control over code and flexibility, it was fairly hard to use. A portable, higher level language for programming the GPU was needed, so Cg was created to overcome these problems and make shader development easier. Cg programs are merely vertex and pixel shaders, and they need supporting programs that handle the rest of the rendering process, Cg can be used with two APIs, OpenGL or DirectX, each has its own set of Cg functions to communicate with the Cg program, like setting the current Cg shader, passing parameters, and such tasks.

High Level Shader Language (HLSL)

The High Level Shader Language or High Level Shading Language (HLSL) is a proprietary shading language developed by Microsoft for use with the Microsoft Direct3D API. It is analogous to the GLSL shading language used with the OpenGL standard. It is very similar to the NVIDIA Cg shading language mentioned above.

Open GL Shading language

GLSL - OpenGL Shading Language also known as GLslang, is a high level shading language based on the C programming language. It was created by the OpenGL ARB to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages.

Production quality shading languages

Many other shading languages such as Renderman Shading Language, Houdini VEX shading language and Gelato Shading language have been developed for

production-quality rendering. A more detailed discussion of these languages is beyond the scope of this thesis.

Brook for GPUs

Brook is a high level GPU programming language developed at Stanford University, and facilitates the usage of graphics processors for General purpose computational applications. The rest of this thesis work is based on Brook and it will be discussed in the following chapters

GPU Programming Difficulties

Originally, GPUs could only be programmed using assembly languages. Programming a GPU is harder than programming a CPU. Buck et al (GPU Stream Computing) provide a detailed explanation of the difficulties in programming a GPU. Higher level shader languages such as Microsoft's HLSL, NVIDIA's Cg and OpenGL's GLslang require that the programmer have extensive knowledge of the underlying graphics hardware and the modern APIs. Also, the algorithms must be mapped to graphics primitives such as textures and triangles, and hence there is little abstraction provided. The stream programming model captures computational locality not present in the SIMD or vector models through the use of streams and kernels. Stream, kernel and memory management etc. must be performed by the programmer and hence require a good operational background in graphics. They conclude that "code written today to perform computation on GPUs is developed in a highly graphics-centric environment, posing difficulties for those attempting to map other applications onto graphics hardware".

Selection of programming language

We conducted a preliminary analysis of various graphics programming languages such as NVIDIA's Cg (C for graphics), Microsoft's HLSL (High level shading language), OpenGL (OpenGL shading language) and Brook for GPUs. Brook was selected for its ease of implementation and abstractness. It makes general purpose programming on graphics processors much easier and provides a good level of abstraction. This is obtained by introducing an intermediate layer, runtime (BRT) and a compiler (BRCC). The details are provided on the Brook web page at Stanford University (BrookGPU). We can think and code in terms of modules (the typical top-down or bottom-up approach) rather than in terms of shading. Brook provides a new perspective to general purpose programming on graphics processors.

CHAPTER 3

BROOK GPU PROGRAMMING LANGUAGE DETAILS*

Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient language. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor. It presents a compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware. Brook abstracts the GPU as a streaming coprocessor. The general computational model, referred to as *streaming*, provides two main benefits over traditional conventional languages:

- *Data Parallelism*: Allows the programmer to specify how to perform the same operations in parallel on different data.
- *Arithmetic Intensity*: Encourages programmers to specify operations on data which minimize global communication and maximize localized computation.

BrookGPU System Architecture

The BrookGPU compilation and runtime architecture consists of a two components. **BRCC** is the BrookGPU compiler is a source to source meta-compiler which translates Brook source files (.br) into .cpp files. The compiler converts Brook primitives into legal C++ syntax with the help of the **BRT**, the **B**rook **R**un **T**ime library.

*Some material in this chapter is excerpted from the Brook GPU website at Stanford University (BrookGPU) and (GPU Stream Computing). It provides the background for this thesis work.

The BRT is an architecture independent software layer which implements the backend support of the Brook primitives for particular hardware. The BRT is a class library which presents a generic interface for the compiler to use. The implementation of the class methods are customized for the hardware supported by the system. The backend implementation is chosen at runtime based on the hardware available on the system or at request of the user. The back-ends include: DirectX9, OpenGL ARB, NVIDIA NV3x, and C++ reference

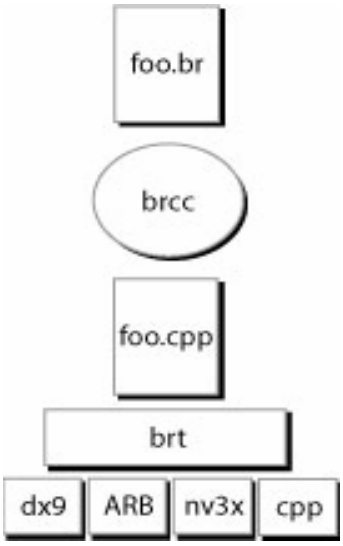

Figure 8: BrookGPU System Architecture

Brook Stream Programming Model

Brook was a developed as a language for streaming processors such as Stanford's Merrimac streaming supercomputer, the Imagine processor etc. It has been adapted to the capabilities of graphics hardware. The design goals of the language include:

1) Data Parallelism and Arithmetic Intensity: By providing native support for streams, Brook allows programmers to express the data parallelism that exists in their applications. Arithmetic intensity is improved by performing computations in kernels.

2) Portability and Performance: In addition to GPUs, the Brook language maps to a variety of streaming architectures. Therefore the language is free of any explicit graphics constructs. Brook implementations have been created for both NVIDIA and ATI hardware, using DirectX and OpenGL, as well as a CPU reference implementation. Despite the need to maintain portability, Brook programs execute efficiently on the underlying hardware.

In comparison with existing high-level languages used for GPU programming, Brook provides the following abstractions:

1) Memory is managed via streams: named, typed, and "shaped" data objects consisting of collections of records.

2) Data-parallel operations executed on the GPU are specified as calls to parallel functions called kernels.

3) Many-to-one reductions on stream elements are performed in parallel by reduction functions.

Features of the Brook language

Important features of the Brook language are discussed in the following sections.

Streams

A stream is a collection of data which can be operated on in parallel. Streams are declared with angle-bracket syntax similar to arrays, i.e. float s<10,5> which denotes a 2-dimensional stream of floats. Each stream is made up of elements. In this example, s

is a stream consisting of 50 elements of type float. The shape of the stream refers to its dimensionality. In this example, s is a stream of shape 10 by 5. Streams are similar to C arrays, however, access to stream data is restricted to kernels (described below) and the streamRead and streamWrite operators, that transfer data between memory and streams. Streams may contain elements of type float, Cg vector types such as float2, float3, and float4, and structures composed of these native types

Kernels

Brook kernels are special functions, specified by the kernel keyword, which operate on streams. Calling a kernel on a stream performs an implicit loop over the elements of the stream, invoking the body of the kernel for each element. An example kernel is shown below.

```
kernel void saxpy (float a, float4 x<>, float4 y<>, out float4 result<>)
{
        result = a*x + y;
}
```

Kernels accept several types of arguments:

1) Input streams that contain read-only data for kernel processing.

2) Output streams, specified by the “out” keyword, that store the result of the kernel computation. Brook imposes no limit to the number of output streams a kernel may have.

3) Gather streams, specified by the C array syntax (array []): Gather streams permit arbitrary indexing to retrieve stream elements. In a kernel, elements are fetched, or

“gathered", via the array index operator, i.e. array [i]. Like regular input streams, gather streams are read-only.

4) All non-stream arguments are read-only constants.

If a kernel is called with input and output streams of differing shape, Brook implicitly resizes each input stream to match the shape of the output. This is done by either repeating (123 to 111222333) or striding (123456789 to 13579) elements in each dimension. Certain restrictions are placed on kernels to allow data-parallel execution. Memory access is limited to reads from gather streams, similar to a texture fetch. Operations that may introduce side-effects between stream elements, such as writing static or global variables, are not allowed in kernels. Streams are allowed to be both input and output arguments to the same kernel (in-place computation) provided they are not also used as gather streams in the kernel.

The use of kernels differentiates stream programming from vector programming. Kernels perform arbitrary function evaluation whereas vector operators consist of simple math operations. Vector operations always require temporaries to be read and written to a large vector register file. In contrast, kernels capture additional locality by storing temporaries in local register storage. By reducing bandwidth to main memory, arithmetic intensity is increased since only the final result of the kernel computation is written back to memory.

Reductions

While kernels provide a mechanism for applying a function to a set of data, reductions provide a data-parallel method for calculating a single value from a set of

records. Examples of reduction operations include arithmetic sum, computing a maximum, and matrix product. In order to perform the reduction in parallel, we require the reduction operation to be associative. This allows the system to evaluate the reduction in whichever order is best suited for the underlying architecture.

Reductions accept a single input stream and produce as output either a smaller stream of the same type, or a single element value. Outputs for reductions are specifed with the “reduce” keyword. Both reading and writing to the reduce parameter are allowed when computing the reduction of the two values. If the output argument to a reduction is a single element, it will receive the reduced value of all of the input stream's elements. If the argument is a stream, the shape of the input and output streams is used to determine how many neighboring elements of the input are reduced to produce each element of the output.

Additional language features

1) The “indexof” operator may be called on an input or output stream inside a kernel to obtain the position of the current element within the stream

2) Iterator streams are streams containing pre-initialized sequential values specified by the user. Iterators are useful for generating streams of sequences of numbers.

3) The Brook language specification also provides a collection of high-level stream operators useful for manipulating and reorganizing stream data, such as grouping elements into new streams and extracting sub-regions of streams and explicit operators to stride, repeat, and wrap streams. These operators can be implemented on the GPU

through the use of iterator streams and gather operations. Their use is important on streaming platforms which do not support gather operations inside kernels.

4) The Brook language provides parallel indirect read-modify-write operators called ScatterOp and GatherOp which are useful for building and manipulating data structures contained within streams. However, due to GPU hardware limitations, these operations are currently performed on the CPU.

CHAPTER 4

CPU-GPU DATAPATHS, MAPPING METHODOLOGY AND CONSTRAINTS

CPU-GPU Datapaths

The overall system architecture of a computer is shown below.

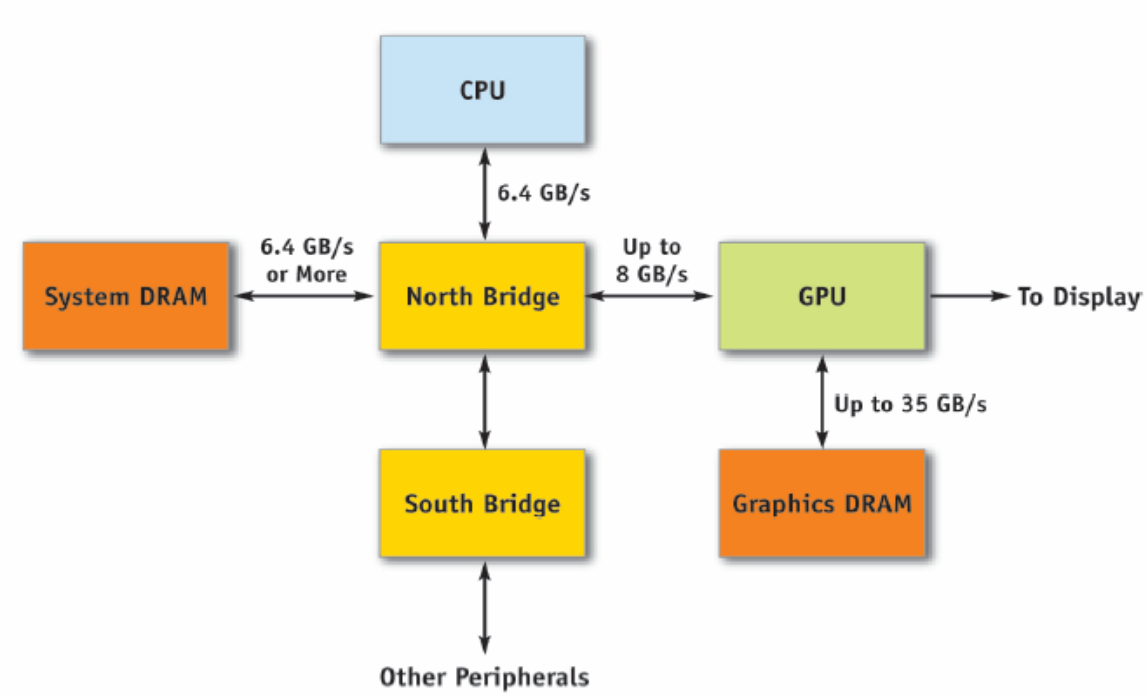

Figure 9: Computer System Architecture
(Courtesy, Ch 30, GPU Gems 2, Fig 30-2)

The South Bridge is connected to the slower components and peripherals such as the printer, the keyboard, the serial port, hard disk etc. The South Bridge connects to the CPU via the North Bridge. The North Bridge is connected to the high speed devices such as the System DRAM, the GPU, Gigabit Ethernet etc. as shown above. Thus, the primary channel of communication between the GPU and the CPU is the North Bridge. The CPU sends information regarding the images to be displayed on the monitor to the GPU. This information is then processed by the GPU and the final image is rendered on the display unit.

When we map a computational problem from the CPU onto the GPU, the bottleneck becomes the bus that connects the two processors. Data transfer from the CPU to the GPU and back is a very expensive process in terms of time. Hence it is essential that the amount of computations that are transferred to the GPU justify the investment in the mapping process and result in performance benefits. In other words, we need high arithmetic intensity, and must re-arrange data so that the parallelism is exploited to the maximum extent.

Mapping Methodology

Logic simulation in CPUs is an inherently sequential process with the output of only one gate being evaluated at every instant of time. If we could evaluate the outputs of multiple gates at the same time, then there would be significant speed improvements in the entire logic simulation process. This would require the following:

1) The underlying hardware should support parallel processing. GPUs are well suited for parallel processing, are effective stream co-processors and hence can crunch large amounts of data at higher speeds than CPUs (as discussed in the introduction).
2) A high level language that allows a sufficient degree of abstractness to provide an interface for general purpose programming on the graphics processor. This is provided by Brook.
3) An effective system to process and organize circuit information such that the multiple computations are made possible concurrently. This work is discussed in the following chapters.

We have to note that there are delays when data is passed from the CPU onto the graphics card and back. The amount of computation on the GPU should be high enough to compensate for this delay and further show a speed improvement

Brook Constraints

Though Brook offers a high level of abstraction, there are quite a few restrictions. The usage of pointers is heavily restricted. Static storage classes are disallowed. Recursion is disallowed, and precise exceptions are not supported. There is no support for the integer data type or variable length streams. Also we have experienced some problems while using multiple nested loops such as for, while etc. Hence the code has to be written in such a way that we obey these restrictions and use alternate methods to get the same result. The final speed will be affected by these constraints. A major factor that we must also consider is the underlying hardware, the graphics card. Hence, this study would compare the CPU and GPU implementations for the logic simulation process.

CHAPTER 5

VARIOUS APPROACHES STUDIED

Many methods were studied for optimizing the logic simulation process on the GPU as a stream computational process. The CPU is used for user interaction, and intermediate data re-arrangement. Though each of these approaches would have worked in terms of results, some approaches were extremely inefficient in terms of overall processing time. We discuss each approach below and analyze its drawbacks, leading us to the final implementation algorithm.

The simple approach: One kernel per gate

The simplest approach is to map each gate of the circuit into a kernel of the graphics processor. As defined earlier, kernel functions are user specified functions which are executed over the set of input streams to produce elements onto the set of output streams. A stream is a collection of objects to be processed. This approach was very slow as it achieved very little arithmetic intensity (ratio of arithmetic computations to the data transferred over the CPU-GPU bus) and does not use any parallelism.

As bitwise logical operations are not supported by the GPU, the kernel functions are implemented using the conditional Boolean operators that are supported by Brook (Brook Language Specifications). The inputs and outputs are represented in terms of floating point numbers.

A typical example for a 2 input nand gate would be as follows:

```
kernel void nand_2ip(float ip1<>, float ip2<>, out float op<>)
   {
      if((ip1 == 0.0)||(ip2 == 0.0)) //any inputs = logic 0
      {
            op = 1.0;
      }
      else
      if((ip1 == 2.0)||(ip2 == 2.0)) //any input = don't care
      {
            op = 2.0;
      }
      else op = 1.0; //all inputs are logic 1

   }
```

In the above example, logic 0 is represented by the number 0.0, logic 1 is represented by the number 1.0, and logic X is represented by the number 2.0. ip1 and ip2 are the input streams and op<> is the output stream.

Using Combinational Fan-Out Free Cones (CFOFs)

To improve on the above approach, we partitioned the circuit into Combinational Fan-Out Free Cones called as super gates, and tried to obtain arithmetic equations for the output of each cone in terms of its input nodes. The reason for trying to map the output of these gates as a function of their inputs using arithmetic equations is that GPUs are much faster when solving equations than when performing logical operations. Each super-gate has one or more inputs and by definition is either an output gate or a multiple fan-out gate. The equations were obtained using regression and curve fitting techniques and were found to be sub-optimal. This was because the presence of XOR gates made these functions non-monotonic and hence irregular to be fitted accurately into curves.

Bubble propagation method

As a result, we used the bubble-propagation approach to make these functions monotonic. The bubble-propagation approach starts at the output of the super-gate and propagates NOT gates (inverters) all the way back to its inputs in a step-wise manner. Equations could now be written for the output of each super-gate in terms of its inputs using back-propagation. This would however be serial in nature, and yet again inefficient.

Threshold gate mapping algorithm

We then analyzed if representing each super-gate as a threshold gate would work. A similar work has been done by Zhang et al, in 2005. They have also developed a threshold logic synthesis tool called TELS for this purpose. Firstly, the functions are assured to be unate, because of the bubble propagation approach used earlier. Hence the necessary condition for threshold functions is satisfied. However, not all unate functions are threshold functions and hence further resolution of these super-gates into threshold functions becomes necessary. This requires Integer Linear Programming (ILP) methods and hence access to ILP tools. The TELS tool runs over 4000 lines of code and requires other ILP tools. Also we would need tools for electronic design automation (EDA) scripting, such as the Simplified Wrapper and Interface Generator (SWIG) (Chen, 2007). This would lead to lots of CPU system calls and lower the overall execution speed.

Usage of Multidimensional streams

One approach is to format the entire circuit as a multidimensional stream, send this stream and the individual gate values as an array and do the processing completely inside the graphics processor. This would require reading and writing information into and from the GPU only once, thus reducing the transit time. This may have been faster, but current GPUs do not support the scatter and gather operations that are necessary to perform this. The scatter and gather operations are emulated on the CPU, and hence very slow.

Final computational approach

From the above experiments, the following methodology is proposed:

1) If the circuit is sequential, convert it into a combinational circuit This is done by converting all the inputs of the flip-flops to primary output gates and converting all outputs of the flip flops to primary input gates
2) Perform technology mapping on the circuit to obtain a unified representation of the circuit. By unified representation, we mean that the entire circuit is mapped into either NAND, NOR or Multiplexer (universal) gates. This is useful for performing SIMD like streaming computations on the data. In this study, we have compared the effects of technology mapping into these universal gates.
3) Balance, and levelize the circuit.
4) Arrange the circuit information into arrays such that input gate information can be sent in parallel as streams. This forms the crucial and most important step.

5) For each level, perform a computation for all the gates in that level, in parallel, using kernel functions for each gate.
6) Arithmetic intensity is achieved using kernel reduction functions and parallelism is obtained using the streams.
7) Measure simulation execution time (exclude file read-write access time) over a large number of simulations, providing random inputs and take the average execution time.
8) Perform a comparison between CPU and GPU execution time and report results.
9) Repeat the study for various technology mapped circuits, benchmarks and platforms and analyze the results.

CHAPTER 6

SIMULATIONS AND RESULTS

Preparation for simulation

Before simulation, any sequential circuits are converted to combinational circuits, technology mapped into the target technology and converted into a final numeric representation of the gates in which all nodes are assigned unique numbers. The conversion of sequential circuits into combinational circuits and the final representation of the circuits in the form of gates were done using Perl scripts. The technology mapping process was done using ABC, a sequential synthesis and verification tool developed by Alan Mishchenko at the University of California, Berkeley. The final circuit representation, in which each node/gate is given a unique number, facilitates the development of a standard system, in which the circuit can be directly read into arrays and later converted into streams to be used for graphics processing. The benchmarks used in this study are a set of standard benchmarks, namely the ISCAS85, ISCAS89, PICOJAVA and the BEAST benchmarks. Henceforth, I will refer to a 4 input NAND gate as NAND4, a 5 input AND gate as AND5 and so on. Mapping was done into NAND4, NAND5, NOR4, NOR5, AND4 and AND5 technologies (with the inverter gate always present). The Beast benchmarks were best suited to AND gate technology mapping i.e., resulting in the least number of total gates. Thus the entire pre-simulation process required the use of a wide variety of tools such as ABC, Synopsys Design Tools, and the Perl Scripting Language.

Measuring program execution time

A very important aspect of this work is the precise measurement of program execution time. By execution time, we mean the process time and not the actual wall clock time, since during any interval of time, there are several multiple processes being executed. If we do not measure the execution time correctly, the entire evaluation process falls apart. Many time measurement approaches are available such as the stop-watch method, the UNIX date command, the UNIX time command, prof and grof profiling mechanisms, the clock( ) function, software analyzers, logic analyzers etc (Stewart,2006)**.** The most accurate measurements among these are found to be obtained from calls to the clock ( ) function and measuring program access cycle counters using register level operations, that are explained in Chapter 19 of the book, Computer Systems: A Programmer's Perspective by Randal E. Bryant and David R. O' Hallaron. I have used these precise counter based measurements to obtain the execution time of my programs. Furthermore, I used the suggestions from the Computer Science Course CS455, Lecture 21 by Jon Squire at the University of Maryland, Baltimore County while performing my simulations.

Measuring transfer delay between the GPU and the CPU

To measure the transfer delay/time due to passing data back and forth between the CPU and the GPU per level, I did the following steps:

- I first measured the time taken to run the logic simulation algorithm with logic kernels, using the GPU as the co-processor.

- Then, I replaced the Nand kernels using buffer logic kernels and repeated the simulation time measurements for the same set of benchmarks.
- Since the buffer logic is a simple assignment operation, the execution time on the GPU can be taken as zero (approximation), and the measured time is the time lost in transferring data back and forth the CPU and the GPU.
- The resulting difference between the above two simulations will be the GPU-only execution time.
- The GPU only execution time is then compared with the CPU execution time.
- We can also obtain the transfer time per level.
- One important point is that though we can obtain the approximate GPU execution time, comparing only this with the CPU execution time would be unfair as we have to look at the data-delay overhead as well. We must compare the overall time (using the GPU as a co-processor, including the data transfer time) with the CPU only time, and make our final judgment.
- However, comparing the GPU-only execution time gives us an idea as to how well logical operations map onto the GPU (which has no native support for bit-wise logical operations).
- The transfer overhead per level was found to be about 275us in both the systems (configuration given below) under study.

Simulation Results

The simulations were performed for the following two systems:

VLSITEST3: Dell Precision 370

System configuration:
CPU: Intel Pentium4 processor, 3.00 GHz Dual Core, 512 MB RAM
GPU: NVIDIA Quadro NVS 280 PCI-E, 64 MB RAM, 250 MHz.

VLSITEST4: Dell Dimension 9200

System configuration:
CPU: Intel Pentium4 processor, 1.86 GHz Dual Core, 1 GB RAM.
GPU: NVIDIA GeForce 8800 GTX, 768 MB RAM, 575 Mhz

The following tables are the results obtained from the simulations conducted on the above 2 platforms. We first present all the results in tabular format. After all the results are presented, a comprehensive analysis of these results is provided. A few notes on reading the tables:

1) The cells marked with *** represent those circuits for which the GPU simulations failed, due to insufficient memory.
2) CPU4 represents the logic simulation time using the CPU of VLSITEST4.
3) CPU3 represents the logic simulation time using the CPU of system VLSITEST3.
4) Similarly, C+GPU4 and C+GPU3 represent the simulation time using the GPUs of systems VLSITEST4 and VLSITEST3 respectively as co-processors.

5) I have used black color to indicate CPU runtimes and red color to indicate C+GPU runtimes. This is to avoid any confusion, as in the case of some benchmarks, they are of different orders of magnitude.

6) %G-IMP represents the percentage improvement obtained when using the higher end GPU as co-processor (VLSITEST4 system) over VLSITEST3's GPU as co-processor.

ISCAS85 Benchmarks- Logic Simulation Results

Table 1: Simulation results- ISCAS85 benchmarks – NAND4 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c17	6	3	1.56	1.31	0.89	1.24	28.23
c432	201	22	17.1	16.59	6.68	8.81	24.18
c499	434	17	27.78	27.7	5.02	6.83	26.50
c880	340	15	30.91	27.58	4.56	5.99	23.87
c1355	434	17	28.43	27.36	5.03	6.78	25.81
c1908	472	27	29.6	26.12	8.2	10.74	23.65
c2670	649	16	88.95	76.48	4.77	6.39	25.35
c3540	977	30	56.49	48.09	9.13	11.95	23.60
c5315	1583	34	105.22	104.88	10.09	13.69	26.30
c6288	2560	91	120.35	102.27	27.64	***	
c7552	2152	29	151.17	141.24	8.88	11.77	24.55

Table 2: Simulation results- ISCAS85 benchmarks – NAND5 mapping

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c17	6	3	1.59	1.28	0.91	1.22	25.41
c432	201	22	17.84	13.88	6.69	8.82	24.15
c499	434	17	28.65	27.97	5.02	6.77	25.85
c880	326	15	27.62	27.57	4.58	5.97	23.28
c1355	434	17	27.54	26.11	5.02	6.83	26.50
c1908	463	27	30.32	29.04	8.2	10.75	23.72
c2670	633	15	84.9	74.84	4.47	6.11	26.84
c3540	973	30	57.99	48.44	9.14	12.09	24.40
c5315	1574	33	108.63	104.86	9.81	13.16	25.46
c6288	2555	90	100.99	95.89	26.98	***	
c7552	2104	28	152.59	112.01	8.35	11.25	25.78

*** **Nomem**

Table 3: Simulation results- ISCAS85 benchmarks – NOR4 mapping

			MICROSECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c17	13	5	1.84	1.58	1.52	1.98	23.23
c432	233	27	18.94	17.44	7.97	10.81	26.27
c499	535	22	31.5	30.44	6.5	8.84	26.47
c880	400	23	31.91	28.85	6.97	9.24	24.57
c1355	535	22	32.24	30.21	6.52	8.73	25.32
c1908	522	33	30.99	23.28	10.02	13.25	24.38
c2670	871	22	97.26	74.65	6.53	8.9	26.63
c3540	1087	39	57.69	44.39	11.84	15.53	23.76
c5315	1721	37	110.95	90.39	10.98	14.75	25.56
c6288	2744	104	112.77	105.63	27.85	***	
c7552	2402	30	153.96	144.02	9.16	12.11	24.36

*** Nomem

Table 4: Simulation results- ISCAS85 benchmarks – NOR5 mapping

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c17	13	5	1.86	1.57	1.52	2	24.00
c432	219	23	18.09	14.61	6.78	9.16	25.98
c499	549	23	32.21	26.3	6.95	9.22	24.62
c880	394	21	31.72	24.82	6.22	8.45	26.39
c1355	549	23	32.66	26.53	7	9.25	24.32
c1908	502	33	30.23	22.8	9.77	13.14	25.65
c2670	852	23	93.57	59.39	7.03	9.39	25.13
c3540	1054	37	52.11	46.53	10.95	14.94	26.71
c5315	1694	37	114.92	109.66	11.3	15	24.67
c6288	2731	103	128.35	117.02	27.79	***	
c7552	2323	30	148.35	147.36	9.16	12.17	24.73

*** Nomem

IMPORTANT NOTE: Please note that the GPU and CPU simulations are orders of magnitude different in this set of benchmarks and hence the GPU simulation time has been highlighted in red. We observe that the GPU Simulations are much slower than their CPU counterpart.

ISCAS89 Benchmarks- Logic Simulation Results

Table 5: Simulation results- ISCAS89 benchmarks – NAND4 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c27	16	6	2.53	2.18	1.78	2.41	26.14
c208	75	8	9.56	9.18	2.73	3.64	25.00
c298	109	9	9.96	9.15	2.67	3.63	26.45
c344	123	13	11.72	11.37	3.84	5.22	26.44
c349	125	13	11.52	10.13	3.83	5.17	25.92
c382	135	9	12.14	10.72	2.74	3.58	23.46
c420	157	13	15.62	15.41	3.84	5.16	25.58
c444	169	9	13.4	12.08	2.75	3.59	23.40
c510	214	9	16.01	13.65	2.67	3.58	25.42
c526	197	9	15.95	15.45	2.75	3.59	23.40
c641	163	15	20.97	18.48	4.44	6	26.00
c713	171	15	21.15	20.91	4.55	6	24.17
c820	305	11	21.57	16.98	3.27	4.38	25.34
c838	309	23	28.67	28.87	6.99	9.27	24.60
c953	352	10	27.67	24.61	2.99	3.99	25.06
c1423	594	38	50.58	46.89	11.55	15.14	23.71
c1488	611	13	35.33	24.88	3.95	5.25	24.76
c9234	1993	26	158.81	146.77	7.75	10.59	26.82
c13207	3184	27	319.32	300.62	8.42	11.14	24.42
c15850	3803	36	329.1	310.45	10.94	14.74	25.78
c35932	12123	13	923.71	873.45	4.55	***	
c38417	11144	21	913.14	791.46	6.54	***	
c38584	12435	23	933.31	827.79	7.03	***	

*** **Nomem**

Table 6: Simulation results- ISCAS89 benchmarks – NAND5 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c27	16	6	2.54	2.27	1.78	2.39	25.52
c208	65	7	7.59	6.99	2.13	2.81	24.20
c298	104	7	8.48	8.23	2.08	2.8	25.71
c344	123	13	11.33	10.14	3.95	5.22	24.33
c349	125	13	11.87	10.94	3.84	5.16	25.58
c382	129	8	11.94	11.87	2.44	3.22	24.22
c420	139	10	15	14.54	2.97	3.99	25.56
c444	165	9	13.57	13.47	2.73	3.63	24.79
c510	200	7	17.24	16.05	2.06	2.81	26.69
c526	180	7	15.68	15.04	2.13	2.78	23.38
c641	156	15	20.83	19.65	4.45	6.03	26.20
c713	163	15	20.51	20.34	4.58	5.98	23.41
c820	277	11	20.16	18.84	3.25	4.39	25.97
c838	273	18	28.5	24.79	5.48	7.22	24.10
c953	312	8	27.89	27.5	2.39	3.2	25.31
c1423	590	37	46.88	41.51	11.25	14.88	24.40
c1488	603	13	24.56	24.49	3.88	5.25	26.10
c9234	1949	25	156.1	134.74	7.66	10.11	24.23
c13207	3135	26	307.33	306.57	7.95	10.72	25.84
c15850	3717	33	325.88	303.33	10.34	13.45	23.12
c35932	12123	13	924.24	919.22	4.52	***	
c38417	11056	21	926.7	897.47	6.51	***	
c38584	12205	22	949.66	823.84	6.97	***	

*** **Nomem**

Table 7: Simulation results- ISCAS89 benchmarks – NOR4 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
c27	10	5	2.27	2.06	1.52	2.02	24.75
c208	98	12	10.57	8.13	3.53	4.82	26.76
c298	94	8	9.19	8.22	2.44	3.22	24.22
c344	135	14	12.57	11.31	4.14	5.63	26.47
c349	137	14	12.65	12.16	4.16	5.61	25.85
c382	124	9	12.26	11.03	2.73	3.59	23.96
c420	188	18	16.82	16.36	5.33	7.13	25.25
c444	160	9	13.63	13.14	2.74	3.63	24.52
c510	246	12	18.72	17.87	3.56	4.78	25.52
c526	186	9	15.88	15.03	2.73	3.59	23.96
c641	221	18	23.38	22.41	5.33	7.13	25.25
c713	222	18	23.25	21.64	5.48	7.17	23.57
c820	309	12	20.01	16.38	3.56	4.83	26.29
c838	362	28	30.97	27.75	8.48	11.14	23.88
c953	415	14	30.01	25.24	4.16	5.58	25.45
c1423	516	40	46.19	38.05	11.26	***	
c1488	634	14	30.7	29.8	4.17	5.64	26.06
c9234	2140	32	150.57	134.68	9.77	12.92	24.38
c13207	3612	33	325.16	299.16	9.98	13.41	25.58
c15850	4216	47	347.64	318.71	***	***	
c35932	10708	12	915.12	876.51	4.47	***	
c38417	9816	25	861.96	833.55	6.67	***	
c38584	13922	29	1029.24	990.92	7.64	***	

***** Nomem**

Table 8: Simulation results- ISCAS89 benchmarks – NOR5 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	Nodes	Levels	CPU4	CPU3	C+GPU4	C+GPU3	%G-IMP
c27	10	5	2.32	2.08	1.53	2.02	24.26
c208	88	10	8.69	8.51	3.03	3.97	23.68
c298	91	7	8.96	8.62	2.06	2.78	25.90
c344	135	14	12.05	11.73	4.25	5.55	23.42
c349	137	14	11.32	10.7	4.14	5.56	25.54
c382	118	8	12.03	11.16	2.44	3.19	23.51
c420	174	14	15.71	14.41	4.16	5.63	26.11
c444	149	9	12.78	11.11	2.75	3.64	24.45
c510	230	12	18.66	17.23	3.56	4.83	26.29
c526	180	9	15.52	12.97	2.45	3.58	31.56
c641	211	18	22.91	22.16	5.33	7.17	25.66
c713	212	18	22.14	20.09	5.45	7.22	24.52
c820	278	12	16.42	15.11	3.55	4.8	26.04
c838	340	20	30.23	30.32	6.08	7.97	23.71
c953	374	12	29.27	25.8	3.55	4.8	26.04
c1423	518	40	45.63	46.87	11.16	***	
c1488	628	14	34.6	30.83	4.14	5.64	26.60
c9234	2114	32	161.44	160.24	9.78	12.98	24.65
c13207	3571	32	339.59	327.75	9.67	13	25.62
c15850	4143	42	343.11	304.28	***	***	
c35932	10708	12	868.66	838.16	4.43	***	
c38417	9668	25	819.97	806.56	6.59	***	
c38584	13698	29	1062.37	976.52	7.59	***	

*** **Nomem**

PICOJAVA Benchmarks- Logic Simulation Results

Table 9: Simulation results- PICOJAVA benchmarks – NAND4 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	Nodes	Levels	CPU4	CPU3	C+GPU4	C+GPU3	%G-IMP
dcu_comb	3250	17	319.91	316.88	5.2	6.98	25.50
ifu_comb	5345	27	262.45	255.08	8.46	11.16	24.19
rcu_comb	10186	29	612.87	541.03	9.93	13	23.62
ex_comb	15573	34	1056.97	907.51	11.26	***	
iu_comb	41475	39	2283.57	2237.98	***	***	
trap_comb	408	13	44.43	37.11	3.82	5.23	26.96
hold_logic_comb	59	7	6.44	5.47	2.05	2.78	26.26
pipe_comb	1042	9	107.31	95.15	2.75	3.7	25.68
ucode_comb	8151	30	524.46	447.05	9.54	12.52	23.80

*** **Nomem**

Table 10: Simulation results- PICOJAVA benchmarks – NAND5 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	Nodes	Levels	CPU4	CPU3	C+GPU4	C+GPU3	%G-IMP
dcu_comb	3214	17	314.73	230.11	5.17	7	26.14
ifu_comb	4703	24	240.97	183.14	7.48	9.95	24.82
rcu_comb	9918	26	569.71	498.93	9.22	11.7	21.20
ex_comb	15160	32	998.27	945.28	10.58	***	
iu_comb	39980	39	2403	2308.5	***	***	
trap_comb	363	13	42.45	36.77	3.95	5.2	24.04
hold_logic	51	6	6.13	5.81	1.78	2.38	25.21
pipe_comb	1027	9	107.24	93.43	2.78	3.64	23.63
ucode_comb	7570	26	504.41	431.56	8.13	11.02	26.23

Table 11: Simulation results- PICOJAVA benchmarks – NOR4 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	Nodes	Levels	CPU4	CPU3	C+GPU4	C+GPU3	%G-IMP
dcu_comb	3945	22	336.22	311.94	6.97	9.14	23.74
ifu_comb	6087	37	294.51	280.2	11.89	15.38	22.69
rcu_comb	11014	40	627.33	621.52	12.38	***	
ex_comb	17331	43	1154.62	1021.16	***	***	
iu_comb	44285	51	2669.87	2008.61	***	***	
trap_comb	413	19	45.24	34.38	5.78	7.64	24.35
hold_logic_comb	66	7	6.87	5.58	2.13	2.81	24.20
pipe_comb	1303	15	117.22	96.04	4.5	6.06	25.74
ucode_comb	9308	43	546.34	509.67	13.61	17.87	23.84

Table 12: Simulation results- PICOJAVA benchmarks – NOR5 mapping.

			MICRO-SECONDS		MILLI-SECONDS		
Circuit	Nodes	Levels	CPU4	CPU3	C+GPU4	C+GPU3	%G-IMP
dcu_comb	3891	19	327.62	306.86	6.03	7.81	22.79
ifu_comb	5526	34	283.87	251.03	10.45	13.94	25.04
rcu_comb	10705	34	615.56	595.55	11.06	14.59	24.19
ex_comb	16609	38	1158.86	968.84	12.85	***	
iu_comb	42678	49	2631.88	1905.26	***	***	
trap_comb	377	15	43.78	37.17	4.45	5.95	25.21
hold_logic_comb	62	7	6.82	5.45	2.12	2.8	24.29
pipe_comb	1293	15	117.23	103.74	4.49	6.02	25.42
ucode_comb	8540	35	514.45	510.28	11.11	14.61	23.96

BEAST Benchmarks – Logic Simulation Results

Table 13: Simulation results- BEAST benchmarks – AND4 mapping.

			MILLISECONDS		MILLI-SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
beast10k	8663	11	1.2	0.82	4.19	6.23	32.74
beast12k	12308	13	1.57	1.1	4.68	7.59	38.34
beast14k	13915	14	1.62	1.4	5.39	8.23	34.51
beast16k	14944	11	1.67	1.55	4.69	7.04	33.38
beast18k	16558	11	2.13	1.82	4.32	***	
beast20k	19485	12	2.16	1.76	5.44	***	

Table 14: Simulation results- BEAST benchmarks – AND5 mapping.

			MILLI SECONDS		MILLI SECONDS		
Circuit	**Nodes**	**Levels**	**CPU4**	**CPU3**	**C+GPU4**	**C+GPU3**	**%G-IMP**
beast10k	7940	10	1.21	0.84	3.95	5.69	30.58
beast12k	11701	12	1.5	1.25	4.48	7.22	37.95
beast14k	13160	12	1.51	1.4	5	7.31	31.60
beast16k	13778	9	1.63	1.35	3.84	5.94	35.35
beast18k	15237	10	2.08	1.63	4.13	6.52	36.66
beast20k	18291	11	2.1	1.6	4.21	***	

NOTE: In tables 13 and 14, we observe that simulation runtime using GPU as co-processor is the same order as the CPU runtime. The BEAST benchmarks in general perform better on the CPU as compared to the other benchmarks, due to higher structural density of the circuit. However, GPU simulations are still slower than CPU simulations.

CHAPTER 7

OBSERVATION, ANALYSIS AND CONCLUSIONS

Observations

In all the simulation results, we observe that:

1. The GPU simulations are slower than the CPU simulations.
2. CPU simulations are faster on VLSITEST3 which has a faster CPU than VLSITEST4.
3. CPU simulation time is heavily influenced by the number of nodes in the circuit, while GPU simulation time is influenced by the distribution of the circuit i.e the number of levels and the number of nodes per level.
4. GPU simulations on VLSITEST4 are faster than VLSITEST3 (20% to 40%), as it has a much faster processor and higher on card memory.
5. Larger circuits are simulated on VLSITEST4, though they resulted in GPU memory failures on VLSITEST3.
6. Another clear outcome of the simulations is that the denser benchmarks, with fewer nodes and more levels such as the BEAST benchmarks perform relatively better when compared to the sparser circuits with lower node to level ratio.
7. Among all the benchmarks in this study, the best results were obtained for the BEAST benchmarks.
8. Sparser benchmarks such as ISCAS85 and ISCAS89 suffer heavily on GPU execution speed, when compared to their CPU counterparts.

9. Thus, we see that small and sparse circuits with fewer nodes per level are much better off being simulated on the CPU and larger and dense benchmarks might be suited to GPU processing.

Analysis

Based on the above observations, we come up with the following analysis and try to understand why logic simulation is slower on the GPU as compared to the CPU. Firstly, data dependency exists between the layers, since the inputs to the kernels/gates of any stage (except the primary inputs) are dependant on the outputs of the previous stage. Since the write position of a processed fragment is determined in advance by the vertex-parameters and cannot be changed within the fragment program, fragment processors are incapable of performing memory scatter. This led us to move data back and forth between the CPU and the GPU for every level and the resulting data transfer overhead offset any performance gains obtained from GPU processing. The lack of support for bitwise logical operations necessitates the use of larger kernels and floating point numbers, reducing program speed. We also need to use if-else statements in the GPU kernels, and they result in GPU-stall cycles. The nature of the circuit is important, as a widely uneven circuit may outrun the maximum stream size and lead to memory issues, leading to runtime crashes. This is why, some circuits such as the NOR4 mapped version of c15850 (ISCAS89 benchmarks) failed to execute on the GPU. The memory on a graphics card determines the maximum size of the circuits that can be simulated on it. Representing logic in terms of floating point numbers results in wastage of memory. Larger memories facilitate larger circuit simulations. Circuits such as c38584 that do

not run on a lower memory chip ran on a higher memory chip. Memory is not the only influencing factor though. If scatter operations could be conducted on the GPU rather than being emulated on the CPU, there would be performance enhancements. Though we see improvements in GPU simulation results as we move to larger circuits, we have also seen that large circuits lead to errors due to insufficient GPU memory and hence we cannot claim that GPU simulations of very large circuits will perform better than their CPU counterparts. Thus, the resulting computational intensity on GPUs is insufficient and as it does not compensate for the data transfer overhead, we see that CPUs outperform the GPUs on compiled code logic simulation.

Conclusions

In this work, we have implemented and compared generic logic simulation algorithms on GPUs and desktops (CPUs). We evaluated the performance for four standard sets of benchmarks, namely ISCAS85, ISCAS89, PICOJAVA and BEAST. We mapped these benchmarks using various technology libraries such as NAND4, NAND5, NOR4, NOR5, AND4 and AND5 (with inverters) and studied the effects of technology mapping on simulation results. Simulations were performed for two different platforms and graphics cards. We described various approaches for mapping the logic simulation algorithm on the GPU and analyzed the performance bottlenecks. We finally concluded that the resulting computational intensity on GPUs is insufficient and as it does not compensate for the data transfer overhead, we see that CPUs outperform the GPUs on compiled code logic simulation.

Future Work

We tried mapping to all the universal gates to observe the effects of technology mapping. When I tried using the MUX based target library, the number of gates and levels inflated by at least a factor of 2. I also tried using the Synopsys design compiler for this purpose and used the "infer_mux" compiler derivative. Another approach I tried was to write my own library, using the Synopsys library compiler. Both the ABC tool and the Synopsys design compiler require that the target library specification contain at least an inverter and an AND2 or NAND2 gate. Upon technology mapping to the MUX target library, I observed that most gates were mapped to these two default gates and very few MUX2 gates were instantiated. This led to the increase in the overall number of nodes and levels. Though MUX4 and MUX8 gates were present in the target library, no instantiations of these gates were found. Thus, the MUX based technology mapping was not used in these simulations.

Future work in this area would be identifying problems in the Computer Aided Design area that are better suited to implementation on the Graphics processor. One such example suggested by my advisor, Prof Kundu was Aerial imaging simulation. We could also work on recently developed languages such as CUDA (Compute Unified Device Architecture) produced by NVIDIA, which supports more fine-grained control over GPU operations.

BIBLIOGRAPHY

"BrookGPU." Computer Graphics At Stanford University. 24 Apr. 2007. <http://graphics.stanford.edu/projects/brookgpu/>.

"Brook Language Specifications." Stanford U,. 24 Aug. 2007 <http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf>.

Bryant, Randal E., and David R. O' Hallaron, eds. "Measuring Program Execution Time." Computer Systems: a Programmer's Perspective. Prentice Hall, 2003. 632-649.

"GPU Stream Computing". Buck, Ian, Tim Foley, Daniel Horn, Jeremy Sugerman, Kavyon Fatahalion, Mike Houston, and Pat Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware." ACM Transactions on Graphics 23 (2004): 777-786.

"Cg Developer Zone." NVIDIA. 22 Aug. 2007 <http://developer.nvidia.com/page/cg_main.html>.

Chen, Pinhong, D. A. Kirkpatrick, and K. Keutzer. "Scripting for EDA Tools: a Case Study." International Symposium on Quality Electronic Design (2001): 87-93. 24 Aug. 2007 <http://ieeexplore.ieee.org/iel5/7308/19762/00915211.pdf?arnumber=915211>.

"DirectX 10." Microsoft Games for Windows. 24 Aug. 2007 <http://www.microsoft.com/windows/directx/default.mspx>.

"GPGPU". General-Purpose Computation Using Graphics Hardware. 22 Aug. 2007 <http://www.gpgpu.org/>.

Hansen, M., H. Yalcin, and J. P. Hayes. "Unveiling the ISCAS-85 Benchmarks: a Case Study in Reverse Engineering." IEEE Design and Test 3rd ser. 16 (1999): 72-80.

Harris, Mark. GPU Gems. 2nd ed. Addison Wesley, 2005. 493-508.

Harris, Mark. "Mapping Computational Concepts to GPUs." GPU Gems 2. Ed. Matt Pharr. Addison Wesley, 2005. 493-508.

"High Level Shader Language." Wikipedia. 25 Aug. 2007 <http://en.wikipedia.org/wiki/High_Level_Shader_Language>.

"HLSL." Microsoft Developer Network. 24 Aug. 2007 <http://msdn2.microsoft.com/en-us/library/bb509561.aspx>.

"ISCAS Benchmark Circuits." Technical University of Liberec. 22 Aug. 2007 <http://www.fm.vslib.cz/~kes/asic/iscas/>.

Jha, Niraj, comp. CAD Projects At Princeton University. Princeton University. 24 Aug. 2007 <http://www.princeton.edu/~cad/projects.html>.

Meister, G. A survey on parallel logic simulation. Technical report, Department of Computer Science, University of Saarland, 1993. <http://citeseer.ist.psu.edu/article/meister93survey.html>

"OpenGL." Wikipedia. 25 Aug. 2007 <http://en.wikipedia.org/wiki/Open_GL>.

Owens, John D., David Leubke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Tim Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, 2007.

Slusallek, Philip. Introduction to Real Time Ray Tracing. SIGGRAPH, 2005.

Squire, Jon. "Lecture 21, Benchmarks, Time and Size." CMSC 455 Selected Lecture Notes. UMBC. 25 Aug. 2007 <http://www.csee.umbc.edu/~squire/cs455_lect.html#L21>.

Stewart, David B. Measuring Execution Time and Real Time Performance. Embedded Systems Conference, Sept. 2006.

Wilson, Tracy V., and Jeff Tyson. "How Graphics Cards Work." How Stuff Works. 22 Aug. 2007 <http://computer.howstuffworks.com/graphics-card.htm>.

“TELS.” Zhang, R., P. Gupta, L. Zhong, and N. K. Jha. "Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies." IEEE Transactions on Computer-Aided Design 1st ser. 24 (2005): 107-118.