Bandwidth Requirements of GPU Architectures

Benjamin C. Johnstone

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Bandwidth Requirements of GPU Architectures

by

Benjamin C. Johnstone

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Sonia Lopez Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2014

Approved by:

Dr. Sonia Lopez Alarcon, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Amlan Ganguly, Assistant Professor
Committee Member, Department of Computer Engineering

Dr. Marcin Lukowiak, Associate Professor
Committee Member, Department of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:
Bandwidth Requirements of GPU Architectures

I, Benjamin C. Johnstone, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

_________________________________________________________________________
Benjamin C. Johnstone

_________________________________________________________________________
Date
Dedication

I dedicate this thesis to my parents for their guidance and support.
I would like to thank my advisors, especially Sonia, for all of their help throughout this process. I would also like to thank Emilio Del Plato and Paul Mezzanini for fielding all of my cluster-related questions.
Abstract

Bandwidth Requirements of GPU Architectures

Benjamin C. Johnstone

Supervising Professor: Dr. Sonia Lopez Alarcon

A new trend in chip multiprocessor (CMP) design is to incorporate graphics processing unit (GPU) cores, making them heterogeneous. GPU cores have a higher bandwidth requirement than CPU cores, as they tend to generate much more memory requests. In order to achieve good performance, there must be sufficient bandwidth between the GPU shader cores and main memory to service these memory requests in a timely manner. However, designing for the highest possible bandwidth will lead to high energy costs. The communication requirements of GPU cores must be determined in order to choose a proper interconnect. To this end, we have simulated several CUDA benchmarks with varying bandwidths using the GPGPU-Sim simulator.

Our results show that the communication requirements of GPUs vary from workload to workload. We suggest that cores be connected using a photonic interconnect capable of supporting different bandwidths in order to reduce power consumption. For each transmission, the interconnect used will depend on how the bandwidth affects performance. We determined that the ratio of interconnect-shader stalls to the total number of execution cycles is a good indicator of whether or not an application will be bandwidth-sensitive. We used this finding to develop a bandwidth selection policy for GPU applications using a photonic NoC. With our policy selections, the photonic interconnect used 12.5% less power than a photonic interconnect with optimal performing choices, which only gave a performance improvement of 1.37% compared to our policy. The photonic interconnect with our policy also had the lowest energy-delay product out of the interconnects we compared it against.
Contents

Dedication ......................................................... iii

Acknowledgments .................................................. iv

Abstract ........................................................ v

1 Introduction ....................................................... 1

2 Background and Related Work ................................. 4
   2.1 Chip Multiprocessors ........................................ 4
   2.2 Graphics Processing Units ................................. 5
      2.2.1 Architecture ........................................ 7
   2.3 Network on Chip ........................................... 8
      2.3.1 Photonic Interconnects ............................. 10
   2.4 Related Work ............................................. 14

3 Proposal and Methodology ..................................... 21
   3.1 GPGPU-Sim ............................................... 21
   3.2 Benchmarks ............................................... 25
   3.3 Testing Method ........................................... 27

4 Experimental Results ........................................... 30
   4.1 Speedup Analysis ........................................ 30
   4.2 Detailed Analysis ........................................ 32
List of Tables

3.1 GPGPU-Sim Baseline Configuration ........................... 24
4.1 Comparison of Benchmark Speedup and Stall Ratio ........ 35
5.1 Bandwidth Selection Policy Groupings ......................... 43
5.2 Photonic Interconnect Energy Dissipation ..................... 44
List of Figures

1.1 Intel Core i7-2600K Die Map [22] . . . . . . . . . . . . . . 2

2.1 Kepler Architecture Block Diagram . . . . . . . . . . . . . 8

2.2 Comparison of Computer Networks and NoCs . . . . . . . . 9

2.3 Comparison of Latency Optimized Electronic and Photonic
   On-Chip Interconnects (16 nm Process) [20] . . . . . . . . . 11

2.4 Corona Architecture Layout [24] . . . . . . . . . . . . . . . 12

2.5 Ring Resonators as Photonic Interconnect Components [20] 14

3.1 SIMT Core Cluster Architecture [10] . . . . . . . . . . . . . 22

3.2 Memory Request Data Flow [10] . . . . . . . . . . . . . . . 22

3.3 Interconnect Injection and Ejection Interfaces . . . . . . . . 23

3.4 GPGPU-Sim Output . . . . . . . . . . . . . . . . . . . . . 28

3.5 GPGPU-Sim Interconnect Statistics . . . . . . . . . . . . . . 29

4.1 Speedup of 1024 Byte Flit vs 32 Byte Flit . . . . . . . . . . 31

4.2 Speedup Versus Flit Size for Selected Benchmarks . . . . . 32

4.3 Selected Statistics for leukocyte Benchmark . . . . . . . . . 33

4.4 Selected Statistics for BFS Benchmark . . . . . . . . . . . . 34

4.5 Selected Statistics for NQU Benchmark . . . . . . . . . . . 35

4.6 Selected Statistics for MUM, b+tree, and cfd . . . . . . . . 36

4.7 Selected Statistics for bfs, kmeans, and particlefilter . . . . 37

4.8 Selected Statistics for CP, LIB, and LPS . . . . . . . . . . . 38

4.9 Selected Statistics for NN, RAY, and hotspot . . . . . . . . 39

4.10 Selected Statistics for myocyte, nw, and backprop . . . . . 40
4.11 Selected Statistics for gaussian, heartwall, and srad  
5.1 Electronic and Photonic Power Consumption Versus Distance  
5.2 Comparison of Bandwidth Selection Policy and Optimal Performance  
5.3 Power Consumption for Each Benchmark and Interconnect Type  
5.4 Average Energy-Delay Product for Each Interconnect
Chapter 1

Introduction

Chip Multiprocessors (CMPs) contain Central Processing Unit (CPU) cores which are geared for low instruction latency. However, for compute intensive applications, accelerators are often necessary to achieve good performance. Highly-parallel applications can very effectively make use of modern Graphics Processing Units (GPUs), which are now programmable to allow general purpose computing. GPUs are currently being incorporated into modern CMPs, such as the Intel i7-2600K shown in Figure 1.1 and AMD Fusion [7], but the bandwidth requirements for general-purpose GPU applications are not well known. CPU cores and GPU cores have vastly different communication patterns. Most CPU cores only have a few threads running at once, and therefore do not generate a lot of memory requests. An ideal GPU application has sufficient parallelism to have hundreds or thousands of threads running in parallel. As a result, GPU cores tend to generate a high volume of memory requests. Therefore, GPU cores have a much higher bandwidth requirement than CPU cores. Naively inserting a GPU core into a CMP will result in poor performance for GPU applications.

Another CMP design consideration is the method of connecting the cores and memory to each other. Early CMPs used a variety of interconnection methods including buses, crossbars, and point-to-point connections. These methods are not scalable because in addition to becoming a performance bottleneck, they make design more difficult as the number of cores increases. Networks on Chip (NoCs) are a better alternative, as they are easier to design and perform well, even with several cores. In essence, a NoC is very similar to a network of personal computers (PCs). Data is divided
into packets and routers at each core send packets to their destination. Several different NoC topologies exist, including torus, ring, and butterfly. The topology dictates the number of hops needed for data to get from one core to another. Cores are typically connected by wires, but alternative technologies such as wireless and photonic interconnects are emerging.

NoCs used in multi-core processors represent a significant portion of CMP power consumption [18][14]. Research has shown that this portion can be anywhere from 30-50% of total chip power. The incorporation of GPU cores into CMPs means that NoCs will need to provide more bandwidth, causing an increase in NoC energy consumption. This is another reason why characterizing GPU communication requirements is important. Any over-provisioning of bandwidth to a GPU will result in unnecessary energy expenditure.

These factors make it necessary to characterize GPU bandwidth requirements. To this end, we have simulated several benchmarks with varying bandwidths. The effects of changing the bandwidth have been explored in other works [10][18]. However, we focused solely on the interconnect and compared to [10] we used a newer architecture and increased the bandwidth to a much higher level. The performance metrics from our simulations showed that high interconnect bandwidth is a necessary but not sufficient condition for optimal performance. Some benchmarks benefited from
having a higher bandwidth, while others did not. Using the results from our GPU communication characterization, we decided that a heterogeneous NoC, capable of supporting more than one bandwidth, would be more energy efficient than having a single high-bandwidth interconnect for all applications.

In order to make the heterogeneous NoC to be effective, we developed a policy to select the bandwidth to use for a given application. Our simulations showed that the ratio of interconnect-to-shader stalls to execution cycles is a good indicator of how bandwidth will affect performance. We have developed a bandwidth selection policy for a photonic NoC based on this observation. For each application, the interconnect used will depend on how the bandwidth affects performance. If it is determined that a higher bandwidth will significantly help performance, then the higher bandwidth can be used. Otherwise, a lower bandwidth will be used, thus saving energy. In order to evaluate the policy, it was compared against the optimal performance bandwidth choices, as well as multiple high-bandwidth wired connections of varying lengths. Although using the bandwidth selection policy yielded a slightly lower average speedup, it was more energy-efficient and had the lowest energy-delay product.
Chapter 2

Background and Related Work

The work that we present in this thesis leveraged previous work in CMP and NoC design. Also, all of our data was collected from the simulation of a GPU architecture. In this chapter we will present the necessary background needed to have an understanding of our work. We also present prior works related to this thesis.

2.1 Chip Multiprocessors

Throughout the 1980’s and 90’s, technological advances allowed single-core processors to bring progressively more processing power to home computers and computing clusters. These processors eventually delivered Giga Floating Point Operations per Second (GFLOPs) of computing power, allowing for more computationally intensive software and user interfaces. Each successive generation featured faster clock speeds and smaller transistors on what could be considered the same hardware, at least from a programming standpoint. In the early 2000’s, technological limitations were beginning to be reached, which slowed the advance of single-core processors. Increasing the processor clock speed and number of activities was becoming infeasible due to the amount of energy being consumed and heat generated. This led to a shift in processor design paradigm. Instead of increasing clock speed and adding more hardware to a single core, the addition of processor cores would be used to increase throughput [19].

There are two main trajectories for multiprocessor design. Multicore processor design focuses on increasing the speed of sequential programs while increasing the number of cores on the chip for higher computing power. The
first multicore processors had two cores, and the number of cores has been doubling every few years. An example of a multicore processor, or CMP, is the Intel Core i7. The many core design paradigm focuses on increasing the throughput of highly parallel applications. GPUs are the exemplar for the many core design paradigm. Many core processors can have hundreds or even thousands of processor cores. These cores however, are less complex than the CPU cores found in CMPs. A sequential program would run much faster on a CMP than a GPU.

Recent CMP architectures have incorporated graphics hardware in an attempt to provide an on-chip solution for improving graphics performance. For example, in 2010, Intel released its first set of processors with built-in Intel HD Graphics hardware. This hardware was designed to provide high definition video and audio for PCs and laptops without the need for a separate graphics card [17]. In addition, AMD has their own line of what they call Accelerated Processing Units (APUs) which feature a built-in GPU as well as a Heterogeneous System Architecture (HSA) for performing computationally intensive tasks using the GPU core [7]. This trend will likely continue on to future processor generations. As the number of CPU cores on-chip increases, we expect that the number of GPU cores will increase as well.

### 2.2 Graphics Processing Units

With advances in technology, graphics hardware went from being expensive, standalone machines to becoming smaller, and more affordable coprocessors for home computers [19]. The early graphics processors of the 80’s and 90’s featured a fixed-function graphics pipeline for more efficient graphics computation. As performance steadily improved, developers created increasingly complex programs, necessitating more functionality which wouldn’t be practical to offer in fixed hardware. The need for programmability was becoming apparent.

In 2001, NVIDIA released the GeForce 3, which opened its instruction set to developers. Microsoft also released shader extensions for the DirectX 8 and OpenGL application programming interfaces (APIs). Eventually,
the pixel stage of the graphics pipeline was opened for programmability as well. With both the pixel and vertex stages programmable, the trend in GPU development shifted to having a unified processor instead of having discrete pixel and vertex hardware [19]. One key motivation for this trend was the fact that developers were creating even more complex applications, which required faster floating point operations. This meant that the processor cores would need to be run at a higher clock rate. From a design standpoint, it is harder to develop a processor when it has to run at a higher clock rate. It made sense then, to create a single type of processor to handle all of the stages as opposed to developing two or three different processor designs. The GeForce 8800 was released in 2006 and featured a unified processor design.

To understand some of the design decisions made in GPU development, it is necessary to have an understanding of the nature of graphics applications. In a given frame, there can potentially be millions of vertices and pixels. Each one of these objects needs to have some sort of operation performed on them in order to end up with the right image for the next frame. Also, in general, all operations are data independent of each other. Put these characteristics together and you have a large number of highly-parallel operations. Therefore, it makes sense for GPUs to have a high degree of hardware parallelism to exploit this. With the advent of the unified processors, researchers came to realize that GPUs had essentially become a large array of identical processors, and that this processing power could be harnessed for general purpose applications [19]. There has been much interest in using graphics processors for general purpose computing (often referred to as general-purpose GPU computing, or GPGPU) due to the fact that GPU performance has been increasing at a much higher rate than multicore performance has. Some GPUs, such as the NVIDIA GT200, are capable of achieving over 1 TFLOP [19].

Early GPGPU applications were cumbersome to develop. Since the only way to access GPU resources was through graphics API calls, problems had to be re-framed as graphics applications. In addition, hardware limitations prevented GPGPU from reaching its full potential. For example, since graphics programs output pixels, which have a fixed XY coordinate
and location in memory, the GPU processor cores had very limited reading and writing ability. There was no way to write to a calculated address, or perform scatter operations. During development of the Tesla architecture, NVIDIA took these problems into account. They decided to make GPUs so that they could be thought of as normal processors by developers. The Tesla GPU architecture featured instruction memory and cache, load and store instructions with more addressing flexibility, and atomic operations, among other things [19]. NVIDIA also released the CUDA C/C++ compiler, allowing GPGPU programs to be written alongside sequential CPU code as an extension of the C++ programming language.

### 2.2.1 Architecture

GPUs contain an array of highly-threaded multiprocessors called stream multiprocessors (SMs) in NVIDIA terminology. Inside each of these SMs are a number of stream processors (SPs) which contain the logic for carrying out arithmetic operations. SPs share the control logic and caches within their SM. A feature of graphics applications is that the rendering algorithms tend to access memory in a way that has high spatial locality. This allows GPU applications to make more efficient use of memory bandwidth. Consequently, graphics applications are less sensitive to memory latency than CPU applications. For this reason, GPUs are designed to maximize the amount of computational logic instead of having large cache memories like in CPUs. The number of SPs has been increasing in each new generation of GPUs. The G80 GPU has 128 SPs, giving it a peak performance of 500 GFLOPs. NVIDIA’s more current GK104 (Kepler architecture) has 1536 SPs (also called CUDA cores) for a peak performance of 3.09 TFLOPs. Figure 2.1 shows a block diagram of a Kepler GPU architecture as an example.

All of the SMs in a GPU share a global memory, which is implemented as graphics double data rate dynamic random access memory (GDDR DRAM). While accessing DRAM does have a high latency, this cost is masked when there are enough threads running. An ideal GPGPU application typically runs anywhere from 5000-12000 threads [19]. The SMs are connected to the
DRAM controllers with an arbitrated bus. While there is a high bandwidth from the SMs to global memory (192 GB/sec in Kepler), this bandwidth is still finite, and congestion is likely to occur when thousands of threads start making memory requests at the same time. Thus, in order to achieve peak performance, a GPGPU application developer must design their program to have a high compute to global memory access ratio (CGMA). One method of increasing the CGMA is to take advantage of the different types of memories found in current GPUs. Modern GPU architectures feature a L1 data cache in each SM as well as a global L2 cache which is shared among all of the SMs. There is also a software controlled shared memory implemented as a partition within the L1 cache. This memory is allocated per-block and the size of the partition can be configured by the programmer to be either 16KB or 48KB.

2.3 Network on Chip

Buses have long been a standard technology connecting separate components in System on Chip (SoC) and CMP architectures. In this paradigm, all of the components or blocks are interconnected with a single group of wires. A bus is a circuit-switched interconnect, meaning that two communicating nodes have a dedicated channel to communicate with. Therefore, only one
block is allowed to send data across the bus at a given time. An arbiter is typically used to regulate the bus and choose the block that is allowed to use it. Although buses have been adequate for previous architectures, there are several trends in SoC and CMP design which indicate that a better method of interconnecting blocks may be needed [1]; one of which being the increase in transistor density according to Moore’s Law. This means that as technology progresses, there will be more components on-chip that will need to be connected to the bus. Since only one component at a time can use the bus, its throughput is very limited. As more components are connected and contending for use of the bus, average communication latency increases. Buses are not scalable enough to keep up with the increased logic and core count of future SoCs and CMPs. Other interconnect types, such as crossbars and point-to-point connections, improve on the performance of buses, but become more difficult to design as the number of cores increases.

Research has demonstrated that NoCs are a viable alternative to buses and other non-scalable interconnects. Applying well-established computer networking concepts to the interconnect yields a scalable solution with better throughput and power consumption than a bus can provide. Unlike buses, NoCs are packet-switched, meaning that all transmitted data is divided into packets. Packet switched networks don’t use a single dedicated channel, so multiple nodes in the network are allowed to send packets simultaneously.
Depending on the flow control mechanism used, some NoCs break packets down even further into flow control digits, also known as flits. In a NoC, nodes are organized in a chosen topology and connected with a series of point-to-point connections. The connections are made according to the network topology. Each node has a switch that sends flits to the next node on the path to their destination. This path is determined by the routing algorithm used in the network.

NoCs offer many advantages over buses. One key advantage is that NoCs offer much higher throughput and lower latency. Performance is greatly dependent on the amount of contention for shared resources [1]. In a NoC, there are several point-to-point connections that can be used by multiple nodes. When using a bus interconnect, all of the nodes are contending for use of the one shared resource, the bus itself. This means more arbitration must occur, resulting in longer latencies and lower throughput. This issue is exacerbated as the number of nodes increases. A NoC, on the other hand, still performs well as the number of nodes increases. Compared to buses, NoCs can also run at a higher frequency and use less dynamic and static power [1].

2.3.1 Photonic Interconnects

As CMP core counts increases, the need for a lower power, low-latency interconnect becomes more evident. Additional cores results in higher power consumption, driving the need for energy efficiency. In addition, more cores means more interconnect traffic. A low latency interconnect will be needed to prevent starvation and congestion. Research has shown that current NoC wired interconnects can make up as much as 30-50% of total power consumption [14]. Also, the power consumption and delay of electronic interconnects increases with wire length. These facts have motivated research into alternative interconnect technologies.

Photonic communication channels such as fiber optic cables have been used for years to communicate over long and medium distances. Advances in photonic technology have made it feasible for use as an on-chip interconnect. A diagram of Corona, a proposed CMP architecture which uses
Figure 2.3: Comparison of Latency Optimized Electronic and Photonic On-Chip Interconnects (16 nm Process) [20]

A photonic NoC, is shown in Figure 2.4 Figure 2.3 illustrates the potential advantages of photonic interconnects. Using data from the International Technology Roadmap for Semiconductor, it shows photonic interconnects having a lower latency as well as better energy efficiency over a distance of about 2cm, which is a feasible length for a wire used for global communication [20]. It should be noted that the length at which the photonic interconnect becomes more efficient, also known as the partition length, is dependent on many factors and is not constant across all configurations. One of the benefits of photonic interconnects is that it’s power consumption is essentially independent of length [24].

A photonic NoC has four key components: a light source, modulators, detectors, and a waveguide. Lasers are typically used as a light source in photonic NoC research, as they are already commonly used in telecommunication applications and are available ”off the shelf” [20]. These lasers
Figure 2.4: Corona Architecture Layout [24]
operate in the C band (1530 to 1565 nm) and have a wide, flat power spectrum. One key benefit of using lasers is that they are able to transmit multiple bits through the same beam of light though a technique known as Wavelength Division Multiplexing (WDM). Using WDM allows bits to be stored in different wavelengths of light. WDM can be achieved by using multiple narrow-band lasers or a single mode-locked laser with a comb filter [20]. WDM increases the interconnect bandwidth by a factor equal to the number of wavelengths used. For example, a photonic interconnect with a single wavelength using a 10 Gbps modulator would have a bandwidth of 10 Gbps. Using WDM to increase the number of wavelengths to 18 would increase the bandwidth to 180 Gbps.

The light emitted by the laser travels through the chip along a waveguide. Waveguides can be made from a variety of materials; however, silicon is most common [20]. For example, the Corona architecture [24] uses two materials: crystalline silicon and silicon oxide. The silicon has a high index of refraction and forms the core, whereas the silicon oxide has a low index of refraction and forms the cladding around the core. One of the benefits of photonic interconnects is that the waveguides are a very low loss transmission medium compared to wires. This means that the signal being transmitted does not degrade as much as it travels along the waveguide. In photonic waveguides, there are four sources of signal propagation loss: absorption of the material, radiative coupling, scattering, and Two-Photon Absorption (TPA), which is the least common source. Absorption in silicon and silicon dioxide is negligible. Radiative coupling occurs when the geometry changes suddenly. This can be avoided by increasing the radius of turns. The main source of loss is scattering, which is caused by the roughness of the side wall of the interconnect. Silicon waveguides have been shown to have losses of 2-3 dB/cm with a bend radius of 10 $\mu$m [24]. Each waveguide is capable of supporting 64 wavelengths.

There are two methods of encoding data into light: direct modulation and separate modulation [24]. Direct modulation involves turning the light source on and off. The absence of light would represent a 0 and a 1 would be registered when light is detected. However, it is more difficult to build a laser on a CMOS process capable of being turned on and off at a high
Figure 2.5: Ring Resonators as Photonic Interconnect Components [20]

enough rate to make the photonic interconnect worthwhile, so separate modulation is more common. Microring resonators have been shown to be capable of modulating at a rate of over 10 Gbps [24]. Each modulator is sensitive to a single specific wavelength. This makes them useful when using WDM because they will not interfere with other wavelengths. The wavelength that the modulator is sensitive to depends on the circumference and the index of refraction of the ring’s materials. By heating or adding charge to the ring, the wavelength can be changed. When light passes by the modulator, any light with the wavelength that the modulator is sensitive to gets enhanced in the ring and as a result, becomes dampened in the waveguide as shown in part b of Figure 2.5. Light at any other wavelength is unaffected by the resonator. This is illustrated in part a of Figure 2.5. To let the wavelength pass by, the resonator has to be heated or charged such that the wavelength no longer resonates in it. By adding germanium to the resonator, it can be made into a wavelength-sensitive detector [24] as in part d of Figure 2.5. When a wavelength of light enters the detector, the germanium will generate a photocurrent. The detector will register a 1 or 0 depending on if the wavelength it is sensitive to is present. A photonic injector, which allows wavelengths of light to be injected from one waveguide to another, is shown in part c of Figure 2.5. Part e of Figure 2.5 is an SEM image of a ring resonator.

2.4 Related Work

Bakhoda et. al. developed GPGPU-Sim in order to characterize non-graphics applications written in CUDA [10]. They were attempting to figure out why some programs, even those with high degrees of parallelism, were
not achieving peak performance. GPGPU-Sim simulates NVIDIA’s Parallel Thread Execution (PTX) instruction set. As part of their research, the group simulated several CUDA programs using various hardware configurations. One experiment involved changing the topology of the interconnection network. They found that the topology used did not generally have a huge impact on performance. When compared to the baseline, there was less than a 20% change in performance for most topologies. Another experiment involved modifying the interconnection latency and bandwidth. Overall, the group found that performance was more sensitive to interconnect bandwidth than latency. It took a 800% increase in router latency (2 cycles to 16 cycles) to get an average performance degradation of 25%. The bandwidth of the interconnect was modified by changing the flit size, which is equivalent to the width of the interconnect. Starting with a baseline of 16 bytes, decreasing the flit size to 8 bytes had a significant detrimental effect on most applications. Increasing the flit size to 32 bytes only had a noticeable effect on a few benchmarks. The effect of further increasing the flit size to 64 bytes was negligible. Bakhoda et. al. suggest that this was because increasing the flit size to 32 completely eliminated stalls on the interconnect from memory to the cores. There is some change in performance, however negligible it may be. Our experiments are dealing with a larger range of bandwidths, from a baseline of 32 bytes to 1024 bytes. In addition, our baseline architecture is based on the GTX 480, a more modern architecture than the one simulated in this work.

The boundary between CMPs and GPUs is not very well defined. Guz et. al. sought to better define these two architectures by creating a unified model and using it to identify areas for which one was better suited over another [15]. They referred to multi-core processors, such as Intel’s Larrabee processor, as Many-Core (MC) machines. MC machines rely on cache memories to reduce the average memory access time and improve performance. Processors capable of running a much higher number of simple threads simultaneously are referred to a Many-Thread machines. These machines rely on having groups of threads run while others are blocked on memory accesses to hide latency. The authors developed a unified model of the two architectures. It consists of several processing elements (order
of 1024) and a large (e.g. 16 MB) cache, effectively making it a hybrid of MC and MT machines. To characterize the tradeoffs of the two machine types, Guz et. al. developed a mathematical formula of the performance of their unified model and determined how varying the number of threads affected performance. They found three distinct operating regions. At lower thread counts, there is enough cache memory to service all of the threads. Since there is adequate cache memory, the performance increases as the thread count increases. This region is called the MC Region. Eventually as the thread count increases, there becomes too many threads for the cache to service properly. This results in more cache misses and higher average memory access times. In this region, there is also not enough threads to mask the memory latency. As a result, performance in this region is lower than the maximum achieved in the MC region. This region is called the valley, due to its shape. As even more threads are added, the performance begins to increase again because more threads are running while others are blocked on memory accesses. Once the performance exceeds the maximum achieved in the MC region, the unified model enters the MT region. The valley region can be reduced or re-shaped by changing some of the parameters in the model. One parameter that the group experimented with was the bandwidth to off-chip memory. What they found was that limiting the bandwidth limited the achievable performance in the MT region. This paper does not use any actual applications in their analysis. We are taking a variety of real CUDA applications, with different characteristics, and determining the effects of changing bandwidth.

Most DRAM memory controllers use an out of order scheduling algorithm, such as First-Ready First Come First Serve (FR-FCFS), to schedule requests in a way that minimizes the number of times the same rows of memory have to be accessed and put into the row buffer [25]. Improving this row buffer locality (RBL) reduces the average memory access time, thus improving performance. Some of these policies also take fairness and request rates into account. An first-in-first-out (FIFO) algorithm would only work well in applications where there is an inherently high degree of RBL. Yuan, Bakhoda, and Aamodt [25] found that each stream of requests coming from a shader core has good RBL. However, the streams are interleaved
as they come into the memory controllers from the interconnects, destroying much of the inherent RBL in the process. This interleaving is what makes out of order scheduling algorithms more effective in GPU systems. One of the drawbacks of out of order schedulers is that they require more logic and take up more area than in order schedulers. Yuan, Bakhoda, and Aamodt developed an arbitration scheme which preserved the RBL of the requests coming in. Instead of using a round-robin policy, the arbiter would give priority to the source which had most recently had an output accepted, thus preserving the stream of requests coming from that source. Since the stream’s RBL is also preserved, the new arbitration scheme makes in order scheduling more effective. The group showed that with their arbitration scheme, an in order scheduling algorithm could achieve up to 91% of the throughput achievable with an out of order scheduler. Interconnect arbitration is not within the scope of our work, which pertains to the bandwidth of the interconnects.

The vast difference between GPU and CPU communication patterns can cause issues not found in homogeneous CMPs. For example, the high number of memory requests coming from GPU cores can interfere with the effectiveness of current DRAM memory controller scheduling algorithms [8]. GPU cores switch execution to another warp when the current warp is stalled on a memory access. As a result, there could potentially be thousands of outstanding memory requests from a GPU core at any given time. Given that the memory controller has a finite request buffer, there is less visibility of the requests from CPU cores because they generate much fewer requests than the GPU cores. This can lead to unfairness and starvation of the CPU cores because the scheduler cannot see the request patterns of the CPU cores while it is inundated with GPU requests. Ausavarungnirun et. al. proposed a staged memory scheduler to solve this problem. The first stage formulates batches of requests coming from the same application that need to access the same row. Next, the batch scheduler stage selects batches to go on to the next stage. The requests in each batch all access the same row, so RBL is maximized. The batch scheduler ensures fairness and efficiency by prioritizing batches from CPU applications, which are more latency sensitive, while ensuring that GPU applications are still able to progress. The
final stage is the DRAM Command Scheduler, which handles the low-level commands to the DRAM bank and ensures that DRAM protocol is being observed. Simulation of the proposed design showed that it provided better performance and fairness than current scheduling algorithms, and was also simpler to implement. This work shows that CPU and GPU cores cannot be naively combined on a chip and be expected to perform well. Our work will ensure that there is adequate bandwidth for the GPU memory requests to reach the memory controller in a timely fashion, which is not in the scope of the work done by Ausavarungnirun et. al.

Interconnects have been shown to contribute significantly to the energy expenditure of CMP systems [14]. Flores, Aragon, and Acacio devised a way to mitigate this using heterogeneous interconnects. All of the interconnects in the system are wired, but their power and latency properties are varied by changing the number of repeaters and the width of the wires. The authors proposed that replies from memory be divided into two types: partial and ordinary. Partial replies contain only the word requested by the core, while ordinary replies contain the entire cache line with the word in it. Since only the word requested is needed for the core to continue, partial replies are considered critical and therefore use the higher-power, lower-latency interconnects. Ordinary replies use the lower-power interconnects, thus saving energy. Simulations of this design showed that it saved an average of 7% execution time and 70% in the Energy-Delay squared Product (ED2P). Our motivation for using heterogeneous interconnects is the heterogeneity of the cores. This work assumes homogeneous CPU cores. While it does not apply directly to heterogeneous CMP, it shows the feasibility of heterogeneous interconnects.

Interconnects not only have a large effect on power consumption, but area as well [18]. A straightforward method to increase interconnect bandwidth is to increase the channel width. Kim, et. al. demonstrated that increasing the NoC router frequency is actually a more effective way of increasing bandwidth. This was done by comparing 3 different interconnects with the same bandwidth, but different channel width and frequency values. They found that the connection with the highest frequency used less area and energy, while also giving better performance. The downside to
this is that the maximum frequency of the router is limited by the critical path of its arbitration logic. In order to remove the arbitration logic, the group proposed a direct all-to-all network overlaid on mesh NoC architecture for the reply network. The design is similar to a channel-sliced network where each memory controller has its own dedicated router at each core. The problem with this is that the memory controller can only inject into a single router, thus limiting the injection BW. DA2mesh leverages the sliced network by having the memory controllers use a different slice for each column in the mesh. For example, memory controller 1 could use slice 0 to send packets to column 0. This design works because of the few-to-many communication pattern of the reply network. Since there much fewer memory controllers than shader cores, there only needs to be a small number of slices when compared to the size of the entire network. Synthesis of the DA2mesh routers showed that they used 23% less area were able to run at 2.5x higher clock rate than the baseline routers. In addition, simulation with GPGPU-Sim showed that applications performed an average of 36% better with the DA2mesh architecture. While this work focuses on changing the router architecture to improve performance, our work will focus on the interconnect technology. We will be modifying the channel width in our simulations in order to simulate different interconnect technologies.

Bakhoda, Kim, and Aamodt attempted to design a NoC that would be more throughput-effective than current NoC designs, meaning that it would provide more performance per unit of area [9]. The motivation for this work was the trend of increasing numbers of cores and threads running on many-core accelerators. Simulations were first performed in order to characterize application performance. One key finding that the authors made was that speedup (with a perfect interconnect) and the memory injection rate have a strong correlation. They also found that many-to-few-to-many traffic pattern exhibited by the manycore architecture is caused not only by the ratio of cores to memory controllers, but by the difference in packet sizes as well. Shader cores tend to send mostly read requests to memory controllers, which send back read replies, which are 8 times larger than the requests. This results in the injection rate for an MC being an average of 6.9 times
greater than that for a shader core. Higher injection rates means that the interconnect to the shader cores is stalled more often, leading to a reduction in the memory injection rate. The authors proposed a checkerboard NoC with full and half-routers to exploit the GPU communication pattern and reduce the overall NoC area. A full-router has full connectivity, while a half-router is limited in that it cannot change the direction that a packet is traveling in. In addition to the checkerboard NoC, the authors also proposed adding additional injection/ejection ports to the routers connected to the MCs in order to reduce congestion, as well as using a channel sliced double network to further reduce area and help load balancing. The proposed modifications reduced the area of the NoC and increased throughput, resulting in an overall improvement in throughput effectiveness by 25.4%. We used the findings related to the interconnect stalls and memory injection rate in our work. This thesis differs from [9] in that we focus on interconnect technology and do not attempt to modify the topology or router structure.
Chapter 3
Proposal and Methodology

Bandwidth requirements of GPU applications need to be better characterized so that they can be effectively integrated into a heterogeneous CMP. We did this by simulating several CUDA benchmark programs using GPGPU-Sim. The experimental results in Chapter 4 show that not all applications benefit from very high bandwidth. A system with interconnects capable of supporting multiple bandwidths may consume less power than a system with only one high-bandwidth interconnect. We believe that a photonic NoC would be an effective method of implementing such an interconnect. The bandwidth of a photonic interconnect can be changed by varying the number of wavelengths used. In order to decide which bandwidth to use for each application without having to profile, we needed to find a correlating factor between bandwidth increase and performance increase. To find this correlation, we performed a detailed analysis of the simulation results.

3.1 GPGPU-Sim

We used GPGPU-Sim to generate the data used in our research. GPGPU-Sim is a GPU simulator that is commonly used in GPU research [18][9]. It has been proven to be an accurate simulator; there was a 97.3% correlation between simulation and hardware when running the Rodinia benchmark suite on a Fermi architecture. GPGPU-Sim simulates the PTX instruction set [10]. It is important to note that the GPU portions of CUDA programs are compiled into PTX, but the PTX code is not actually run on the GPU hardware. When running on hardware, the PTX code is further compiled and optimized by the ptxas compiler into SASS code to perform better on
The microarchitecture model of GPGPU-Sim is analogous to that of a physical graphics processor. There are a number of Single Instruction Multiple Thread (SIMT) cores, which are the equivalent of SMs [10]. Each SIMT core contains a multithreaded execution pipeline. An SP would be equivalent to one lane in this pipeline. The SIMT cores are grouped together in a SIMT core cluster, which is connected to a series of memory partitions with an interconnection network. Each memory partition has an L2 cache bank and a channel to one or more DRAM chips.

The interconnection network is modeled by a modified version of the Booksim simulator [10]. Booksim was modified to support two separate
Figure 3.3: Interconnect Injection and Ejection Interfaces
networks: one for communication from the cores to the memory partitions, and another for communication from the memory partitions to the cores. The motivation for using two separate networks is that it is a method of preventing protocol deadlock. Each SIMT core cluster has an injection port buffer through which its SIMT cores inject packets into the network. Traffic coming into each SIMT core cluster is received in the cluster’s response FIFO and then sent to the intended SIMT core. There are injection and ejection buffers separating the interconnect clock domain from the DRAM and core clocks [5]. One packet per cycle may be injected into or ejected from each buffer. Likewise, one flit per cycle may enter the network from each boundary buffer, regardless of the size of the flit. This property effectively allows the width of the interconnects to be changed by changing the flit size. Figure 3.3 illustrates the injection and ejection interfaces.

Table 3.1: GPGPU-Sim Baseline Configuration

<table>
<thead>
<tr>
<th>Processing Clusters</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores per Cluster</td>
<td>1</td>
</tr>
<tr>
<td>Memory Controllers</td>
<td>6</td>
</tr>
<tr>
<td>DRAM Chips per MC</td>
<td>2</td>
</tr>
<tr>
<td>Core Clock Rate</td>
<td>700 MHz</td>
</tr>
<tr>
<td>Interconnect Clock Rate</td>
<td>700 MHz</td>
</tr>
<tr>
<td>L2 Clock Rate</td>
<td>700 MHz</td>
</tr>
<tr>
<td>DRAM Clock Rate</td>
<td>924 MHz</td>
</tr>
<tr>
<td>Warps per Cluster</td>
<td>48</td>
</tr>
<tr>
<td>Lcnt. Topology</td>
<td>Butterfly</td>
</tr>
<tr>
<td>Flit Size</td>
<td>32 B</td>
</tr>
</tbody>
</table>

Table 5.2 shows the baseline GPGPU-Sim configuration that we used for our simulations. This configuration came packaged with the simulator. It is supposed to model a NVIDIA GTX 480 GPU, which is a Fermi architecture. This configuration is set up such that it has 15 processing clusters and one core per cluster. There is a one-to-one mapping of SIMT cores to SIMT core clusters instead of having multiple SIMT cores within each cluster. The only option we ever changed throughout the course of our experiments was the flit size.
3.2 Benchmarks

For our benchmark programs, we drew from the Rodinia benchmark suite [11][3] as well as the benchmarks used in [10], which came packaged with GPGPU-Sim.

**BFS [10]:** Breadth-First Search algorithm used to find nodes in a graph. Each node in the graph is mapped to a different thread. This benchmark has heavy global memory traffic. The search was performed on a random graph with 65,536 nodes, with 6 edges per node on average.

**CP [10]:** Coulombic Potential algorithm used in molecular dynamics. Several optimizations have been made including manually unrolled loops and storing point charge data in constant memory. 200 atoms were simulated on a 256x256 grid. This program is part of the Parboil Benchmark suite.

**LIB [10]:** LIBOR Monte Carlo simulations; threads read a large number of variables from constant memory. We simulated with default inputs, 4096 paths for 15 options.

**LPS [10]:** 3D Laplace Solver is used for finance. It features shared memory usage and coalesced global memory accesses. We simulated 1 iteration on a 100x100 grid.

**MUM [10]:** MUMmerGPU DNA sequence alignment program. The reference string is stored as a suffix tree in texture memory. The tree’s large size results in high cache miss rates. The simulation was done with a 140,000 character reference string and 50,000 randomly generated 25-character queries.

**NN [10]:** Neural Network recognizes handwritten digits. Neuron weights and input digits are stored in global memory. Original source code was modified so that multiple digits could be recognized at once. 28 digits from the Modified National Institute of Standards Technology were simulated.

**NQU [10]:** N-Queens solver places N queens on an NxN chess board such that no queen can capture another. Majority of computation is performed by a single thread. For our simulations, we used N=10.

**RAY [10]:** Ray-tracing is a graphics rendering algorithm. Each thread corresponds to a rendered pixel. Simulation was done with a 256x256 image
with up to 5 levels of reflections and shadows being accounted for.

**b+tree [3]**: Graph traversal algorithm. Nodes in b+ trees typically have a large number of children.

**backprop [11]**: Back Propagation an algorithm used in machine learning to train nodes on a neural network.

**bfs [11]**: Another implementation of the Breadth-First Search algorithm.

**cfd [3]**: Computational Fluid Dynamics solver for 3 dimensional Euler equations for compressible flow. Modified to reduce the number of global memory accesses.

**gaussian [3]**: Gaussian elimination solves for all variables in a linear system. Solves a fixed 4x4 matrix, then an internally generated 16x16 matrix.

**heartwall [3]**: Tracks the movement of a mouse heart over a series of 104 609x590 ultrasound images.

**hotspot [11]**: HotSpot is a program which estimates processor temperature based on a given floor plan and power estimations.

**kmeans [11]**: K-means clustering algorithm assigns data points to clusters. After each iteration, a new cluster centroid is computed. This process continues until convergence.

**leukocyte [11]**: Leukocyte tracking detects and tracks white blood cells through several frames of a video microscopy of blood cells.

**myocyte [3]**: Models cardiac myocyte by solving a set of 91 differential equations and 480 supporting equations for a series of time steps. One instance of the simulation is run for a time interval of 100 ms.

**nw [11]**: Needleman-Wunsch method for DNA sequence alignment. Pairs of sequences are organized in a matrix and a trace-back process finds the optimal alignment. The sizes of the two sequences are 2048 and 10.

**particlefilter [3]**: Estimates a target object’s position based on noisy measurements of its location and an approximation of the target’s path. Used a variety of fields including video surveillance.

**srad [11]**: Speckle Reducing Anisotropic Diffusion is an algorithm used to remove speckles from images while preserving any important features. An ultrasound image is used as the input.
3.3 Testing Method

The first test we ran was to see how having an “infinite” bandwidth would affect performance. Since it is impossible to simulate an infinite bandwidth, we had to create a case where the bandwidth would be high enough that it wouldn’t be a limiting factor to performance. We used a 1024B flit size as our “infinite” case and compared the performance against the baseline configuration. Next we simulated the benchmarks with intermediate flit sizes. Starting with the baseline, we increased the flit size by powers of two until we reached 1024B. In addition, we decreased the flit size to 16B to see the effect of decreased bandwidth. This method of changing the flit size was used in [10] and is an accepted method of changing the interconnect bandwidth in GPGPU-Sim.

The simulator generates a variety of statistics including cycles per instruction (CPI), total cycles, cache miss rates, bottleneck analysis, memory usage, and interconnect statistics. Figure 3.4 shows an excerpt of a GPGPU-Sim output file, while Figure 3.5 shows another excerpt from the interconnect section of the output. A new set of statistics is generated after a kernel is completed, but some of the statistics are tracked across the entire application. In addition to documenting statistics from the simulation output, we also track how these statistics change as the bandwidth changes. We do this for each statistic by calculating the ratio compared to the baseline.

As part of the bottleneck analysis, GPGPU-Sim keeps track of the number of cycles that the interconnect outputs to DRAM are stalled [10], which we will refer to as DRAM stalls, and the number of cycles that the DRAM channels can’t output any data due to interconnect congestion; we refer to these as interconnect to shader stalls or just interconnect stalls. Another part of our investigation involved calculating the ratio of stalls to simulation cycles for each bandwidth. The idea to do this was inspired by [9]. We expanded on the idea by looking at how the ratio changed as the bandwidth increased.
kernel_name = _58Kernel24Wcode010b82_81_82_i
kernel_launch_uid = 9
gpu_sim_cycle = 1429
gpu_sim_insn = 917504
gpu_ipc = 642.0602
gpu_tot_sim_cycle = 761203
gpu_tot_sim_insn = 16800488
gpu_tot_ipc = 22.0535
gpu_tot Issued_cto = 0
gpu_stall_dram_full = 1914356
gpu_stall_lcnt2sh = 476526
gpu_total_sim_rate = 23269

--------- Core cache stats ---------

L1I_cache:
L1I_total_cache_accesses = 938432
L1I_total_cache_misses = 993
L1I_total_cache_miss_rate = 0.0011
L1I_total_cache_pending_hits = 0
L1I_total_cache_reservation_fails = 6682

L1D_cache:
L1D_cache_core[0]: Access = 83401, Miss = 89733, Miss_rate = 0.836, Pending_hits = 4895, Reservation_fails = 606287
L1D_cache_core[1]: Access = 61945, Miss = 67730, Miss_rate = 0.831, Pending_hits = 4682, Reservation_fails = 591830
L1D_cache_core[2]: Access = 68027, Miss = 68027, Miss_rate = 0.820, Pending_hits = 4900, Reservation_fails = 596226
L1D_cache_core[3]: Access = 81750, Miss = 87732, Miss_rate = 0.829, Pending_hits = 4791, Reservation_fails = 589443
L1D_cache_core[4]: Access = 63239, Miss = 65315, Miss_rate = 0.833, Pending_hits = 5004, Reservation_fails = 601747
L1D_cache_core[5]: Access = 62241, Miss = 66766, Miss_rate = 0.835, Pending_hits = 4829, Reservation_fails = 602521
L1D_cache_core[6]: Access = 78022, Miss = 85617, Miss_rate = 0.830, Pending_hits = 4576, Reservation_fails = 583950
L1D_cache_core[7]: Access = 82983, Miss = 89254, Miss_rate = 0.835, Pending_hits = 4873, Reservation_fails = 606581
L1D_cache_core[8]: Access = 64777, Miss = 66745, Miss_rate = 0.824, Pending_hits = 4932, Reservation_fails = 597527
L1D_cache_core[9]: Access = 82788, Miss = 84023, Miss_rate = 0.834, Pending_hits = 4987, Reservation_fails = 604841
L1D_cache_core[10]: Access = 84656, Miss = 70846, Miss_rate = 0.836, Pending_hits = 4952, Reservation_fails = 605063
L1D_cache_core[11]: Access = 60522, Miss = 67100, Miss_rate = 0.839, Pending_hits = 4705, Reservation_fails = 608892
L1D_cache_core[12]: Access = 83906, Miss = 89991, Miss_rate = 0.833, Pending_hits = 4980, Reservation_fails = 604879
L1D_cache_core[13]: Access = 82211, Miss = 68732, Miss_rate = 0.836, Pending_hits = 4819, Reservation_fails = 604829
L1D_cache_core[14]: Access = 62068, Miss = 69109, Miss_rate = 0.835, Pending_hits = 4795, Reservation_fails = 606520
L1D_total_cache_accesses = 1217915
L1D_total_cache_misses = 1029718
L1D_total_cache_miss_rate = 0.8322
L1D_total_cache_pending_hits = 72410
L1D_total_cacheReservation_fails = 6597046

Figure 3.4: GPGPU-Sim Output
<table>
<thead>
<tr>
<th>Traffic class</th>
<th>Packet latency average</th>
<th>Network latency average</th>
<th>Flit latency average</th>
<th>Fragmentation average</th>
<th>Injected packet rate average</th>
<th>Accepted packet rate average</th>
<th>Injected flit rate average</th>
<th>Accepted flit rate average</th>
<th>Injected packet size average</th>
<th>Accepted packet size average</th>
<th>Hops average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32.207s (9 samples)</td>
<td>21.141s (9 samples)</td>
<td>15.949s (9 samples)</td>
<td>0.0300472s (9 samples)</td>
<td>0.0515354s (9 samples)</td>
<td>0.0515354s (9 samples)</td>
<td>0.125313s (9 samples)</td>
<td>0.125313s (9 samples)</td>
<td>2.48991s (9 samples)</td>
<td>2.48991s (9 samples)</td>
<td>1 (9 samples)</td>
</tr>
</tbody>
</table>

===== Overall Traffic Statistics =====
Chapter 4

Experimental Results

In this chapter we show our speedup analysis and detailed analysis of the simulated benchmarks, as well as how the results from these analyses lead to the bandwidth selection policy in Chapter 5.

4.1 Speedup Analysis

In our first experiment, we increased the flit size to 1024 B to simulate an "infinite" bandwidth, meaning that the bandwidth between the shader cores and DRAM was not a performance bottleneck. Figure 4.1 shows the speedup of using 1024B flits versus the baseline 32B flit. With a 1024 B flit size, the bandwidth of the interconnect would be $5.73 \, Tbps$. The increased bandwidth improved the performance of some benchmarks, but in several cases the performance was unaffected. BFS and cfd experienced the greatest speedups, with 1.63 and 1.59 respectively. However, 13 out of the 21 benchmarks only saw a change in performance of less than 4%. In the 16B flit configuration, only 7 of 21 benchmarks experienced a performance decrease by the same margin. Although, the more bandwidth-sensitive benchmarks (BFS, cfd, kmeans) experienced a more dramatic drop in performance, some of the less sensitive benchmarks saw a drop in performance as well. Based on these results, we conclude that high bandwidth is a necessary but not sufficient condition for optimal performance. If a GPU were designed to have the highest desirable bandwidth, it would use more power and performance would not improve for all applications. Therefore, we suggest that an interconnect be used that can provide a higher bandwidth only when it would improve the performance. This would allow for better performance than the
baseline interconnect, but at the same time it would use less energy than an interconnect which can only deliver a higher bandwidth.

The second experiment that we performed was to simulate the benchmarks with flit sizes between 32 and 1024 B in order to see how the performance changed as bandwidth increased and determine the point at which bandwidth is no longer a limiting factor. Starting at 32 B, we doubled the simulated flit size and repeated this until the flit size reached 1024 B. Looking at the speedup values from the intermediate bandwidths, we saw that bandwidth-sensitive benchmarks reached a performance plateau around 128B. This behavior is illustrated in Figure 4.2. In addition, detailed analysis shows that across all benchmarks, the statistics we recorded stayed relatively constant beyond 128B flit size. This shows that over-provisioning bandwidth to a GPU will be a waste of resources; the power consumption would increase, but there would be little to no improvement in performance.
4.2 Detailed Analysis

In an attempt to find a statistic that could indicate how increasing bandwidth could affect performance, we performed a more detailed analysis of the simulation output by looking at metrics other than the CPI. We wanted to find a way to predict how applications would behave when the bandwidth increased without actually having to run them with multiple bandwidths. We then planned to incorporate our findings into a bandwidth selection policy for our photonic interconnect. Initially, we looked at the number of memory accesses for each benchmark. Intuitively, it makes sense that a benchmark with more memory accesses would benefit from having a higher bandwidth. However, we found that this is not the case. Some of the benchmarks that did not improve had numbers of memory accesses comparable to those that did improve.

For each benchmark, we then compared the number of DRAM and interconnect stalls at each bandwidth to the baseline. Among the benchmarks that
Figure 4.3: Selected Statistics for leukocyte Benchmark

...performed better with higher bandwidth, we found that both the number of DRAM and interconnect stalls decreased as the bandwidth increased. Figure 4.4 shows a detailed view of BFS, which exhibits this behavior. However, for some other benchmarks, the interconnect stalls would decrease but this would be counteracted by an increase in DRAM stalls. This behavior is illustrated in Figure 4.5. Presumably, the increased bandwidth allows for data from DRAM to be sent back to the shader cores faster. This means that blocks are not stalled on memory accesses as long, which in turn means that they generate memory requests at a higher rate, resulting in more DRAM stalls. We believe this increase in the number of DRAM stalls effectively canceled out the benefit of having fewer interconnect stalls. In some other cases, such as in the case of leukocyte, both the DRAM and interconnect stalls decreased, but bad memory locality resulted in an increase in the L2 miss rate. The increase in cache misses meant that more memory accesses were required, which would negate some of the benefit of the reduction in stalls. All of these observations are results of increasing the bandwidth and therefore not useful in making a prediction as to how bandwidth will affect performance. We were finally able to find a correlation when we looked...
at the rate at which stalls were occurring when compared to the number of execution cycles. The idea for this came from [9], which showed that there was a correlation between speedup and the memory injection rate when using a perfect NoC (zero latency and infinite bandwidth). Congestion in the MC to shader core interconnect causes stalls, which leads to a lower memory injection rate and lower performance. Increasing the bandwidth of the interconnect helps alleviate the congestion and reduce the number of stalls.

![Figure 4.4: Selected Statistics for BFS Benchmark](image)

We attempted to estimate the percentage of time that the shaders were stalled by looking at the ratio of each type of stall compared to the total number of execution cycles. We found that in most cases, the benchmarks that benefited from higher bandwidth had a high (greater than 0.95) ratio of interconnect stalls to execution cycles in the baseline configuration. Figures 4.4 and 4.5 show this behavior. BFS, which benefits from higher bandwidth, has a baseline ratio of 6.25; whereas NQU, which does not benefit, has a baseline ratio of 0.01. The correlation between this ratio and performance improvement is further illustrated in table 4.1.
Figure 4.5: Selected Statistics for NQU Benchmark

Table 4.1: Comparison of Benchmark Speedup and Stall Ratio

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Interconnect Stall Ratio</th>
<th>Speedup with 1024B Flit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>6.26</td>
<td>1.63</td>
</tr>
<tr>
<td>cfd</td>
<td>8.04</td>
<td>1.59</td>
</tr>
<tr>
<td>kmeans</td>
<td>10.12</td>
<td>1.28</td>
</tr>
<tr>
<td>particlefilter</td>
<td>0.42</td>
<td>1.25</td>
</tr>
<tr>
<td>b+tree</td>
<td>2.52</td>
<td>1.20</td>
</tr>
<tr>
<td>bfs</td>
<td>5.02</td>
<td>1.14</td>
</tr>
<tr>
<td>MUM</td>
<td>7.27</td>
<td>1.13</td>
</tr>
<tr>
<td>heartwall</td>
<td>1.87</td>
<td>1.09</td>
</tr>
<tr>
<td>srad</td>
<td>1.57</td>
<td>1.04</td>
</tr>
<tr>
<td>leukocyte</td>
<td>0.42</td>
<td>1.04</td>
</tr>
<tr>
<td>gaussian</td>
<td>0.01</td>
<td>1.03</td>
</tr>
<tr>
<td>backprop</td>
<td>0.97</td>
<td>1.03</td>
</tr>
<tr>
<td>myocyte</td>
<td>0.00</td>
<td>1.02</td>
</tr>
<tr>
<td>hotspot</td>
<td>0.56</td>
<td>1.01</td>
</tr>
<tr>
<td>LIB</td>
<td>0.00</td>
<td>1.01</td>
</tr>
<tr>
<td>NN</td>
<td>0.08</td>
<td>1.01</td>
</tr>
<tr>
<td>LPS</td>
<td>2.65</td>
<td>1.01</td>
</tr>
<tr>
<td>CP</td>
<td>0.02</td>
<td>1.00</td>
</tr>
<tr>
<td>nw</td>
<td>0.01</td>
<td>1.00</td>
</tr>
<tr>
<td>RAY</td>
<td>0.16</td>
<td>0.98</td>
</tr>
</tbody>
</table>
Figure 4.6: Selected Statistics for MUM, b+tree, and cfd
(a) bfs

(b) kmeans

(c) particlefilter. This benchmark does not exhibit a high interconnect ratio yet gets relatively high speedup.

Figure 4.7: Selected Statistics for bfs, kmeans, and particlefilter
(a) CP
(b) LIB
(c) LPS. Exhibits high interconnect stall ratio but does not benefit much from increased bandwidth

Figure 4.8: Selected Statistics for CP, LIB, and LPS
Figure 4.9: Selected Statistics for NN, RAY, and hotspot
Figure 4.10: Selected Statistics for myocyte, nw, and backprop
Figure 4.11: Selected Statistics for gaussian, heartwall, and srad
Chapter 5

Bandwidth Selection Policy

In the previous section we suggested that power consumption may be reduced by using heterogeneous interconnects. When integrated into a CMP, a GPU core may be connected via photonic interconnects, which can provide higher bandwidth more efficiently than electronic interconnects (especially over long distances) [20]. The bandwidth of these photonic interconnects may be modified by varying the number of wavelengths used. Increasing the number of wavelengths increases the bandwidth, but also increases the power consumption. Only those applications that benefit from increased bandwidth should use it.

Our bandwidth selection policy is based on our finding that in general, the applications that benefit from increased bandwidth have a higher ratio of interconnect stalls to execution cycles. Knowing this, we separated the benchmarks into two groups, one for benchmarks with a high baseline ratio and one for benchmarks with a low baseline ratio. Using the statistics gathered from our simulations, we calculated the power consumed by the interconnect for each application when using the appropriate bandwidth.

Table 5.1 shows how we divided the benchmarks based on their interconnect stalls ratio. Benchmarks with a ratio greater than or equal to 0.95 were placed in the high bandwidth group. Those with a ratio less than 0.95 were placed in the low bandwidth group. It’s important to note that there are some cases in which the policy mispredicts how the benchmarks will behave. For example, particlefilter has a ratio of 0.42 yet has a speedup of 1.25 when the flit size was increased to 1024 B. There is also a case where the policy mispredicts a benefit from high bandwidth; LPS has a stall ratio
Table 5.1: Bandwidth Selection Policy Groupings

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Bandwidth</td>
<td>BFS, cfd, kmeans, LPS, b+tree, bfs, MUM, heartwall, srad, backprop</td>
</tr>
<tr>
<td>(640 Gbps)</td>
<td></td>
</tr>
<tr>
<td>Low Bandwidth</td>
<td>particlefilter, gaussian, leukocyte, myocyte, hotspot, LIB, NN, NQU, CP, nw, RAY</td>
</tr>
<tr>
<td>(180 Gbps)</td>
<td></td>
</tr>
</tbody>
</table>

of 2.65 but only has a speedup of 1.01 when simulated with "infinite" bandwidth. We attempt to take these discrepancies into account by comparing our policy against the "optimal" choices, or the choices which result in the best performance. In the optimal group, all of the benchmarks except CP and RAY use the high bandwidth.

5.1 Power and Bandwidth Calculations

Our photonic network model consists of a single photonic waveguide. A waveguide is capable of transmitting up to 64 different wavelengths using WDM [24]. The modulators for each wavelength can operate at 10 Gbps. Therefore, a single waveguide can provide a bandwidth of 640 Gbps. At the moment, GPGPU-Sim is unable to simulate a photonic network. Therefore,
Table 5.2: Photonic Interconnect Energy Dissipation

<table>
<thead>
<tr>
<th>Action</th>
<th>Energy per cycle (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch bit</td>
<td>0.15 [23]</td>
</tr>
<tr>
<td>Modulate bit</td>
<td>0.04 [13]</td>
</tr>
<tr>
<td>Tune detector by 1 nm</td>
<td>0.24 [13]</td>
</tr>
</tbody>
</table>

we decided to use a configuration with an equivalent bandwidth to represent a photonic network. Below are the calculations for determining the flit size at 700 GHz to provide this equivalent bandwidth:

\[
BW_{\text{photonic}} = f_{\text{tuning}} \times n_{\text{wavelength}} = 10 \text{ Gbps} \times 64 = 640 \text{ Gbps} \quad (5.1)
\]

\[
\frac{640 \text{ Gbps}}{8} = 80 \text{ GBps} \quad (5.2)
\]

\[
\frac{80 \text{ GBps}}{700 \text{ MHz}} = 114.29 \text{ B} \quad (5.3)
\]

We decided to use the closest power of two as the equivalent flit size, which was 128B. This high bandwidth photonic interconnect was compared to another photonic interconnect with bandwidth equal to our baseline. In order to calculate the power consumption of such an interconnect, the number of wavelengths used needed to be determined. This was done with the following calculations:

\[
BW_{\text{baseline}} = 32B \times 700 \text{ MHz} = 179.2 \text{ Gbps} \quad (5.4)
\]

\[
n_{\text{wavelength}} = \frac{179.2 \text{ Gbps}}{f_{\text{tuning}}} = 17.9 \approx 18 \text{ wavelengths} \quad (5.4)
\]

In a photonic interconnect, energy is needed to power the light source as well as heat the modulators to tune them. The following formulas were used to calculate power consumption for our photonic interconnects (\(S_{\text{flit}}\) represents the flit size in bits):
Figure 5.1: Electronic and Photonic Power Consumption Versus Distance

\[ E_{\text{cycle wavelength}} = 0.15 + 0.04 + 0.24 \times \left( \frac{n_{\text{wavelength}}}{2} \right) \text{pJ} \quad (5.5) \]

\[ E_{\text{cycle}} = E_{\text{cycle wavelength}} \times n_{\text{wavelength}} \quad (5.6) \]

\[ E_{\text{flit}} = E_{\text{cycle}} \times \frac{S_{\text{flit}}}{BW_{\text{photonic}}} \times f_{\text{clk}} \quad (5.7) \]

\[ E_{\text{photonic}} = E_{\text{flit}} \times n_{\text{cycles}} \times R_{\text{flit}} \quad (5.8) \]

\[ P_{\text{photonic}} = E_{\text{photonic}} \times \frac{f_{\text{clk}}}{n_{\text{cycles}}} \quad (5.9) \]

We compared the performance of our heterogeneous photonic interconnect against that of a standard wired interconnect. Equation 5.10 was used to determine the power consumption. As an example, we compared the power consumption of our high-bandwidth photonic interconnect against that of a wired interconnect with equivalent bandwidth. For the flit rate and execution cycles numbers we used the values from the MUM benchmark, which has the highest flit rate among bandwidth-sensitive benchmarks, with a flit
size of 128B. The result is shown in figure 5.1. At wire lengths of 5.5 mm and lower, the wired interconnect actually uses less power than the photonic interconnect. The die size of a GTX480 is 529 mm$^2$ [4]. Assuming the die is square, it would be 23 mm $\times$ 23 mm. When comparing the photonic interconnect against the wired interconnect, we considered three wire lengths: a worst-case length of 46 mm (equivalent to the length of two sides of the die) and an average case of 16.25 mm (half of the diagonal) and 5.5 mm.

$$P_{\text{wired}} = l_{\text{wire}} \times 0.2 \text{pf/mm} \times (0.5 \times \text{flit rate}) \times V_{DD}^2 \times f_{clk} \times S_{\text{flit}} \quad (5.10)$$

### 5.2 Evaluation

To evaluate our bandwidth selection policy, we compared the bandwidth choices made using the policy against the choices that would yield the highest performance, which we call the optimal choices. In addition, we compared the selected and optimal bandwidths against wired interconnects.

As expected, the optimal choices resulted in better average speedup. However, the increase in power consumption was much greater than the increase in performance. The average power consumption of the benchmarks using the optimal bandwidth choices was 12.50% greater than the average power consumption when using our policy’s bandwidth choices. Figure 5.2 shows the speedups achieved by the bandwidth selection policy and the optimal choices. The average increase in performance from using the optimal choices opposed to the policy choices was only 1.37%. Most of this increase in performance is due to the fact that our policy mispredicted the performance of particlefilter. Even though its interconnect stalls ratio was only 0.42, particlefilter experienced a speedup of 1.21 when the bandwidth was increased to 640 Gbps.

The wired interconnects only provide the high bandwidth, making them less flexible than the photonic interconnect. The average performances of the wired interconnects are approximately equal to that of the photonic interconnect with optimal bandwidth choices. However, the wired interconnects
with the lengths that we chose (46 mm and 16.25 mm) consumed significantly more power than the photonic interconnect did. We additionally show a 5.5 mm wired interconnect, which was shown in Figure 5.1 to have approximately equal power consumption to the photonic interconnect when using high bandwidth. Figure 5.3 shows the power consumption for each benchmark with each of the different interconnects. The 46 mm wire, on average, used 9.79 times more power than the photonic interconnect with the policy choices and 8.70 times more power than the photonic interconnect with optimal choices. Compared to the policy selected and optimal photonic bandwidth choices, the 16.25 mm interconnect used 3.46 and 3.07 times more power, respectively. Even the 5.5 mm interconnect used 17% more power than the policy chosen interconnect and 4% more than the optimal photonic interconnect. On average, the electronic interconnects used
Figure 5.3: Power Consumption for Each Benchmark and Interconnect Type

4.56 mJ per millimeter of wire, meaning that a 4.7 mm wire would consume the same average power as a photonic interconnect using our policy, and any wire shorter than that would consume less power, on average.

Energy Delay Product (EDP) is a common metric to measure the tradeoff between performance and energy consumption. Our method for calculating the EDP for each benchmark and interconnect is shown in Equation 5.11. Figure 5.4 illustrates the average EDP for each of the interconnects. It shows that the photonic interconnect, when using our bandwidth selection policy, has the lowest EDP. This means that it has the best balance of performance and energy consumption. Intuitively, this makes sense because the photonic interconnect with our policy choices offers nearly identical performance and uses on the order of 10% less power than the next most efficient interconnect (photonic with optimal bandwidth choices). Ultimately, the decision of whether or not to use our bandwidth selection policy depends on the design goals of the architecture. If performance is the primary concern, then it would make sense to use only the high bandwidth or use profiling to find the performance-optimal bandwidth choices. In an energy constrained design paradigm, wired interconnects would make sense if the wire lengths
are short enough that the wires are more energy efficient than the photonic interconnect. Otherwise, our bandwidth selection policy offers comparable performance with good energy efficiency.

\[
EDP = E_{\text{interconnect}} \times n_{\text{cycles}} = P_{\text{interconnect}} \times \frac{n_{\text{cycles}}}{f_{\text{clk}}} \times n_{\text{cycles}} \quad (5.11)
\]
Chapter 6

Conclusion

The bandwidth requirements for modern GPGPU applications need to be better understood. Modern CMP architectures are adding GPU cores in order to leverage their raw processing power for highly-parallel applications. Having a better understanding of the GPU’s communication requirements will allow the GPU cores to be more effectively integrated with the CPU cores, thus improving performance and energy efficiency. Other works have attempted to characterize the bandwidth requirements for general-purpose GPU applications. We expanded on these works by using a wider range of bandwidths, using a more modern architecture, and focusing solely on the interconnect technology used.

Using GPGPU-Sim, we simulated a set of benchmarks with an ”infinite” bandwidth. The bandwidth was changed by modifying the flit size used by the interconnect. This experiment showed us that not all benchmarks benefited from the increased bandwidth. Next we simulated the benchmarks with a series of flit sizes, ranging from 16B to 1024B. From the results we saw that in the cases where the bandwidth increased, the increase would level off in most cases at around 128 B or 256 B flit size. In addition, decreasing the flit size from the baseline 32 B to 16 B resulted in a decrease in performance for several benchmarks. We concluded that high bandwidth is a necessary but not sufficient condition for good performance.

The results from the previous experiments showed us that a GPU could benefit from offering multiple bandwidths. We saw that not all applications benefit from having the highest necessary bandwidth, so providing this for all applications would then be a waste of energy. Using an interconnect capable of providing more than one bandwidth would give the GPU the
flexibility to provide very high bandwidth to those applications that would benefit from it and a lower bandwidth to those that would not, thus reducing power consumption. We proposed that the GPU use a photonic interconnect, which is capable of providing multiple bandwidths when using WDM by varying the number of wavelengths used.

In order to choose the proper bandwidth for an application without having to run it several times, we had to develop a bandwidth selection policy for the photonic interconnect. To do this, we first needed to find a correlating factor between bandwidth increase and performance increase. We performed a detailed analysis of the statistics generated by our previous GPGPU-Sim simulations in order to find this correlating factor. The ratio of interconnect stalls to execution cycles (in the baseline configuration), a measure of how often the interconnect was stalled, was the statistic that we found best correlated with performance. A higher ratio suggested that an application would benefit from increased bandwidth, whereas an application with a lower ratio typically did not benefit much from having a higher bandwidth. Using these findings we decided that the policy would be that any application with a baseline interconnect stall ratio of 0.95 or greater would use the high bandwidth and any application with a lower ratio would use the lower bandwidth.

We evaluated our policy by comparing its decisions against the decisions for a photonic interconnect that would provide the optimal performance. In addition, we also compared it against three different lengths of wired interconnects. The optimal decisions provided only 1.37% better performance, yet caused the photonic interconnect to consume 12.5% more energy. The wired interconnects provided approximately the same performance as the photonic interconnect with optimal bandwidth choices. Both groups of photonic interconnect decisions consumed less energy than the three wired interconnects that we chose. However, as the wire length decreases, electronic interconnects use less power while providing the same bandwidth. This indicates that electronic interconnects would be the ideal choice for a NoC when the wire lengths are short. Analysis of the energy-delay products for the different interconnects shows that the photonic interconnect with our bandwidth selection policy provides the best balance of performance and
power consumption.

Our work has shown that photonic NoCs can be a flexible and power-efficient alternative to electronic interconnects. While electronic interconnects are still more efficient over short distances, photonic NoCs could be viable in a larger scale system, such as a heterogeneous CMP. Our work used a very simple power model for the wired interconnects. It would be worthwhile to compare the performance of the photonic interconnect against a more power-efficient wire model. In addition, wire lengths from actual CMP architectures should be used. Interconnect area is another important metric which should be taken into account in future research.
Bibliography


