HYBRID NANOPHOTONIC NOC DESIGN FOR GPGPU

A Thesis

by

WEN YUAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2012

Major Subject: Computer Engineering

HYBRID NANOPHOTONIC NOC DESIGN FOR GPGPU

A Thesis

by

WEN YUAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Eun Jung Kim
Committee Members,	Hank Walker
	Paul V. Gratz
Head of Department,	Hank Walker

May 2012

Major Subject: Computer Engineering

ABSTRACT

Hybrid Nanophotonic NoC Design For GPGPU. (May 2012)

Wen Yuan, B.S., National University of Defense Technology

Chair of Advisory Committee: Dr. Eun Jung Kim

Due to the massive computational power, Graphics Processing Units (GPUs) have become a popular platform for executing general purpose parallel applications. The majority of on-chip communications in GPU architecture occur between memory controllers and compute cores, thus memory controllers become hot spots and bottle neck when conventional mesh interconnection networks are used. Leveraging this observation, we reduce the network latency and improve throughput by providing a nanophotonic ring network which connects all memory controllers. This new interconnection network employs a new routing algorithm that combines Dimension Ordered Routing (DOR) and nanophotonic ring algorithms. By exploring this new topology, we can achieve to reduce interconnection network latency by 17% on average (up to 32%) and improve IPC by 5% on average (up to 11.5%). We also analyze application characteristics of six CUDA benchmarks on the GPGPU-Sim simulator to obtain better perspective for designing high performance GPU interconnection network.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Advances in technology have made it possible to accommodate an increasing number of transistors on a die, enabling designers to integrate a vast number of diverse components on a single chip. This has lead to chip multiprocessors(CMP) and systems-on-chip(SoC) becoming the norm in the computing world. This trend has also facilitated, the Modern Graphic Processing Units (GPUs) to have a huge number of parallel processors. Accordingly, the recent GPUs managed to break the teraflop barrier [1] and are being made easier to program for non-graphics applications. Therefore, General Purpose GPU (GPGPU) has recently obtained attention as a cost-effective approach for accelerating compute- and data-intensive applications. The sheer number of components on a modern chip, raises the issue of being able to connect them.

Networks-on-chip(NoC) have emerged as an efficient and scalable solution to the communication problem, replacing the buses that were prevalent up to now [3]. An NoC is implemented as a switched network connecting cores in a scalable and flexible manner, which achieves higher performance and lower power consumption than a crossbar-based interconnect. The performance of the NoCs has become a crucial component of the CMP and GPU performance. Many interconnection topologies and routing algorithms have been proposed for CMPs, of which the Two dimensional mesh is a popular and efficient topology. Although the interconnection network has been well researched in Chip-Multi-Processors (CMPs) [4], the same design techniques cannot be directly used in GPGPU architecture, where communication patterns are different from those in CMPs.

This thesis follows the style of *IEEE Transactions on Automatic Control.*

In the GPGPU paradigm, the threads which are supposed to communicate with each other are allocated and executed on Hardware units(compute cores) that are located in a tightly-knitted cluster. In turn, the threads running in parallel in the different clusters of compute cores have minimal communication [6]. As the number of pins on a chip is increasing by only 10% per year [34], the number of Memory Controllers (MCs) on a GPU is limited. However, the number of compute cores on the GPU increases with the transistor density . This results in the many-to-few-to-many traffic pattern, where requests are sent from many compute cores to a few MCs, and replies are received inversely. Therefore, MCs become the hot spots and bottle neck in conventional mesh interconnect networks. When these MCs are at the central part of a mesh, the traffic being routed through these nodes becomes even heavier.

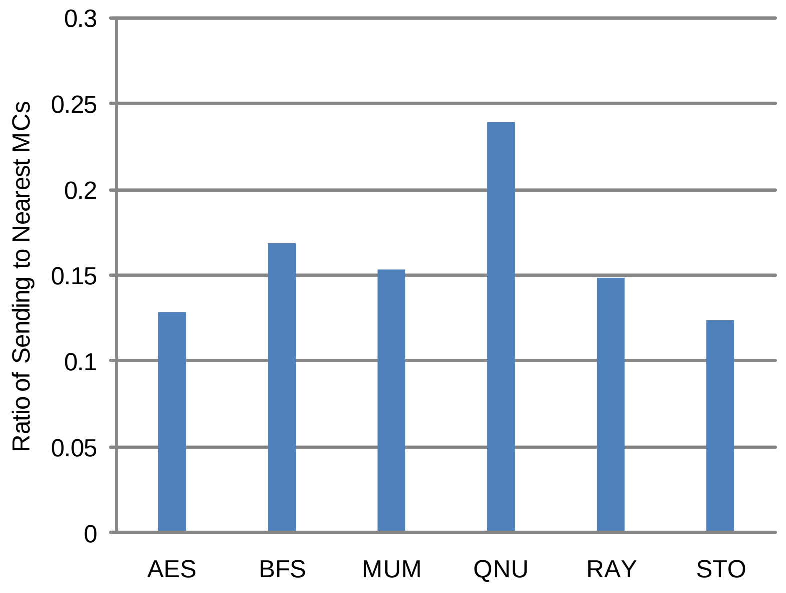

Fig. 1. Ratio of Communication with The Nearest MCs.

Figure 1 shows the ratio of compute cores sending request to the nearest MCs. This ratio is relatively low. It hurts system performance when a large amount of memory requests have to go through distances to reach corresponding MCs to be served. Considering this traffic pattern in GPU interconnection networks, the mesh network is not suitable for high performance on-chip networks design.

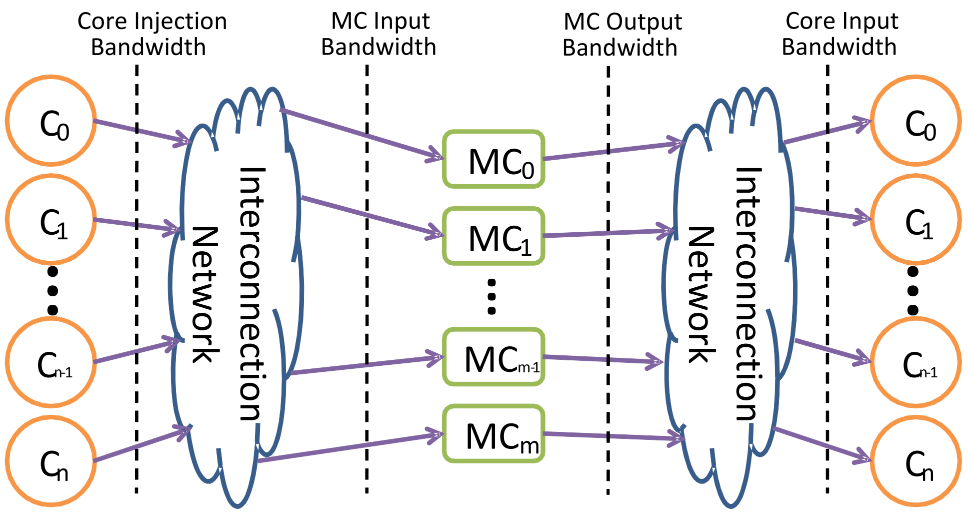

Fig. 2. Many-to-Few-to-Many On-Chip Traffic.

The high level diagram of many-to-few-to-many traffic is depicted in Figure 2. In the figure, C nodes and MC nodes denote the compute cores and the memory controllers, respectively.

Exploiting the many-to-few-to-many traffic behavior of GPGPU architecture, we attempt to optimize the design of NoC for GPGPU architecture. In this research, we first analyze application characteristics of six common GPGPU benchmarks written in CUDA. Since different applications vary in many aspects, including load/store ratio, read request/write request ratios and number of accesses to shared memory, constant memory, texture memory, local memory and global memory, it is important to know these key metrics for further improvement of the GPGPU interconnection network design. Furthermore, based on the conventional mesh network drawback we observed, we propose to add a nanophotonic ring network to further reduce the interconnection network latency and improve the system performance. By exploring this new topology, we can reduce interconnection network latency by 17% on average (up to 32%) and improve IPC by 5% on average (up to 11.5%).

The rest of this thesis is organized as follows: Chapter II presents detailed information about the GPGPU architecture, Chapter III gives an overview of several important aspects of NoC design, Chapter IV describes nanophotonic ring networks architecture, Chapter V presents the simulator we used in this research, Chapter VI conveys the idea of our proposed work, Chapter VII describes experimental methodology and results, Chapter VIII summarizes related work and we conclude our work in Chapter IX.

CHAPTER II

GPGPU HARDWARE AND SOFTWARE ARCHITECTURE

In this chapter, we illustrate the GPGPU hardware architecture and software application specifications and describe the CUDA benchmarks we simulated in the GPGPU-Sim for this research.

A. A GPGPU Baseline Architecture

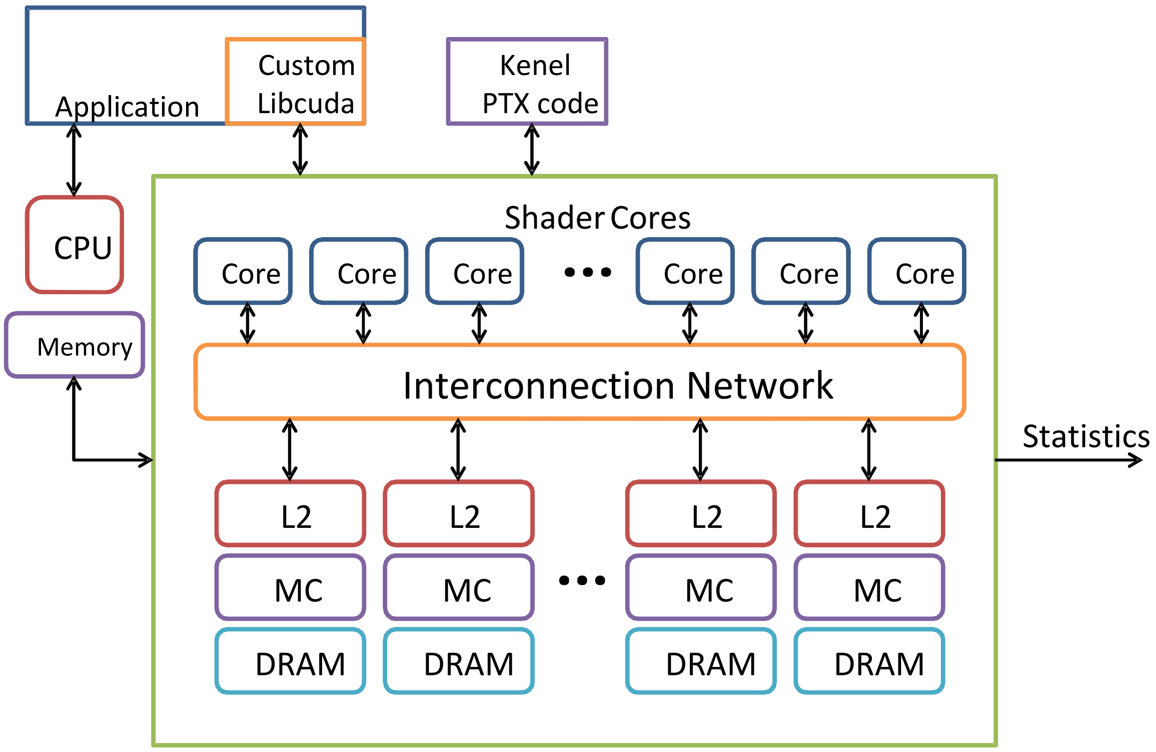

Fig. 3. GPGPU Overview.

Figure 3 shows our baseline GPGPU architecture. The GPGPU consists of a group of small data-parallel compute cores, which labeled shader cores in the figure, connected by an interconnection network to multiple memory controllers. Threads are evenly spreaded to shader cores at the granularity of CTAs, such as shared memory space, registers and thread slots, are not released until all threads within a CTA have finished execution. If the resource is abundant, multiple CTAs can be grouped into a single shader core, this is to share a pipeline for their execution.

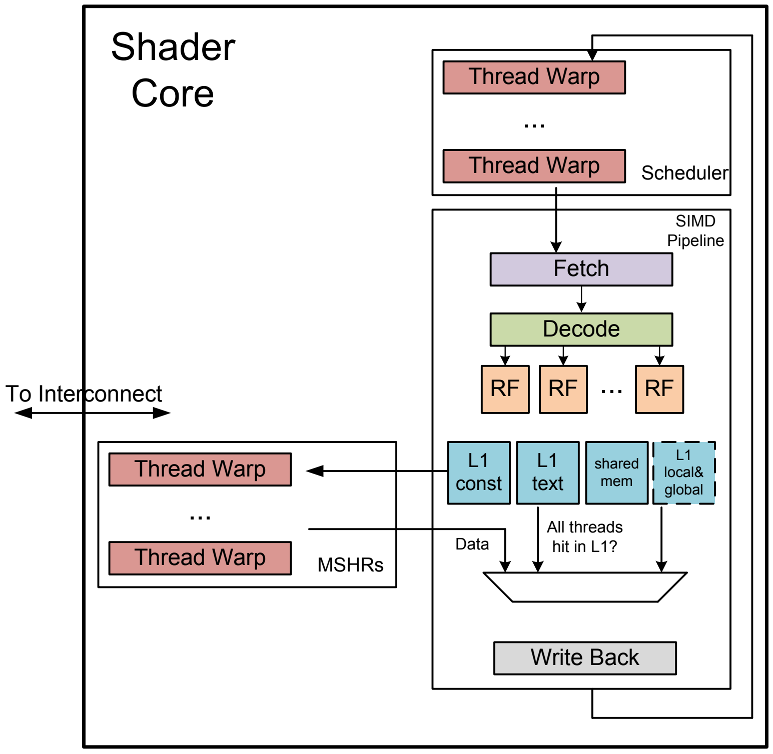

Fig. 4. Details of Shader Core.

Figure 4 illustrates the detailed implementation of a single shader core. In our research, each shader core has a SIMD width of 8 and uses a 24-stage pipeline without forwarding. The 24-stage pipelines are motivated by details in the CUDA programming guide [2], which denotes that at least 192 active threads are needed to avoid stalling for data dependencies between consecutive instructions from a single thread. Warp [24] schedules threads to the SIMD pipeline in a fixed group of thirty-two threads. All the thirty-two threads in a given warp execute the same instruction with different data values over 4 consecutive clock cycles in all pipelines.

With no overhead on a fine-grained basis, threads scheduling inside a shader core is performed. Every four cycles, warps available for execution are chosen by the warp scheduler and issued to the SIMD pipelines in a loose round-robin fashion which skips non-available warps (e.g. these waiting on global memory accesses). This means whenever any thread inside a warp are taken out of the scheduling pool until the long-delay operation is over. At the same time, others warps which are not waiting are issued to the pipeline for execution in a round-robin manner. The multiple threads running on different shader core thus permit a shader core to tolerate long-delay operations without degrading throughput.

To access global memory, the requests should be sent via an interconnection network to the corresponding Memory Controllers (MCs), which are physically distributed over the chip. Every on-chip MC then connects to 2 off-chip GDDR3 DRAM chips. Figure 5 illustrates the physical layout of the MCs in 8X8 mesh configuration as blue areas. The address decoding method is implemented in a way such that successive 2KB pages [27] are spread across various banks and different chips to maximize row locality while distributing the load among the MC.

B. CUDA Programming Model and Flow

At the high level, CUDA can be considered as a set of extensions to the C programming language which allows developers to distinguish between highly multithreaded GPU functions, and native host functions. Kernels are presented as SIMD programs and explicitly control the GPU memory hierarchies. Kernels are managed by a series of API calls implemented by the CUDA runtime that allocate GPU memory, launch kernels and copy data between the host and GPGPU memories, etc. The CUDA compiler compiles kernels to PTX that is an intermediate virtual ISA that is translated to the native ISA at load time. The compiled PTX kernels are packed into a

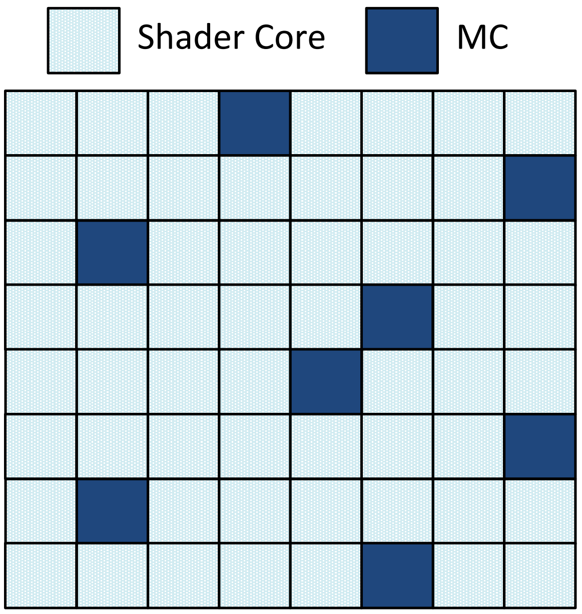

Fig. 5. MC Layout in 8X8 Mesh Network.

fat binary structure contains separate entries for each kernel and is stored as a static array along with the C code to be executed in the host. The resulting source file is a pure C source file, which is then propagated to a native compiler such as msvcc or gcc. The final source file includes references to the CUDA API functions, and must be linked against an implementation of the CUDA runtime to be executed.

Figure 6 illustrates how a CUDA application can be compiled to run on GPGPUs. The compilation flow uses cudafe to translate the source code of CUDA applications into host code running on CPUs and device code running on GPGPUs. The GPGPU C code is then compiled into PTX assembly (marked as ".ptx" in Figure 6) by nvopencc, which is an open source compiler provided by NVIDIA based on Open64 [35]. The PTX assembly code is then assembled by the PTX assembler (ptxas) into the target GPGPU's native ISA (marked as "cubin.bin" in Figure 6). The assembled code is combined with the host C code and compiled into one executable linked with the CUDA runtime API library (marked as "libcude.a" in Figure 6) by

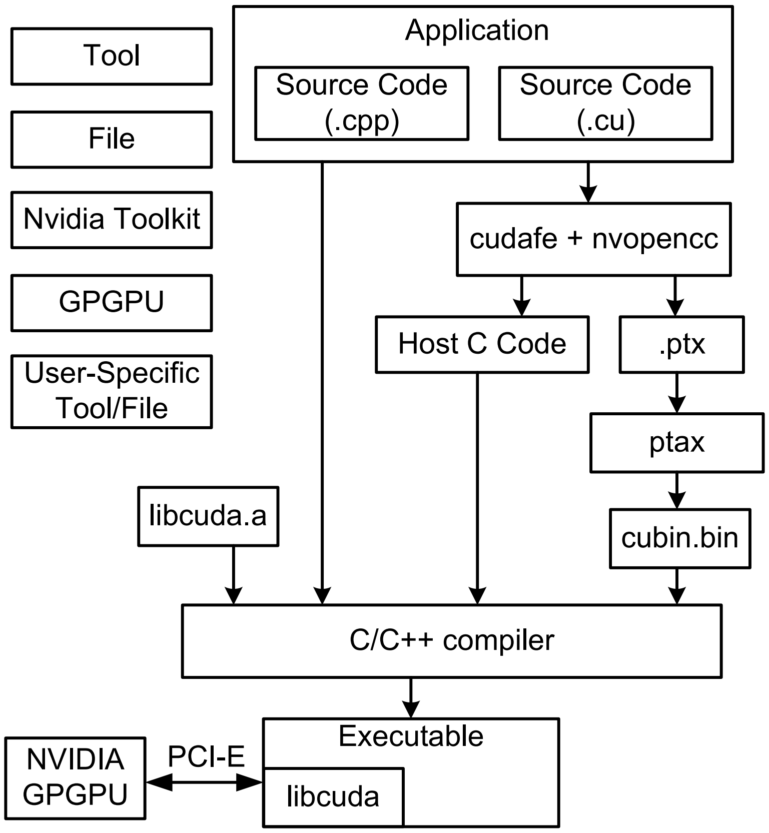

Fig. 6. CUDA Flow with GPU Hardware.

a C compiler such as gcc. In this normal compilation flow, the generated executable calls the CUDA runtime API to launch and invoke compute kernels onto the GPGPU via the NVIDIA CUDA driver.

C. Benchmarks

The benchmarks we used to import to the GPGPU-Sim are listed in Table I together with important application properties, such as the grid dimensions, CTA dimensions, total threads and instruction count. Below, we present some details about these benchmarks we simulated in this research which are not from NVIDIA's CUDA Software Development Kit (SDK). These CUDA applications were developed by the researchers which are cited below and kept the original version on the simulator.

Graph Algorithm: Breadth First Search (BFS) [28] This application, developed by Harish and Narayanan [28], implements breadth first search on a graph. As each thread maps in the graph maps to a different node, the amount of parallelism in this applications scales with the size of the input graph. BFS suffers from performance loss due to branch divergence and heavy global memory traffic. We run breadth first search on a arbitrary graph with 65,536 nodes and an average of 6 edges per node.

AES Encryption (AES) [29] Developed by Manavski [29], this application performs the Advanced Encryption Standard (AES) algorithm in CUDA to encrypt/decrypt files. It is optimized by Manavski so that the input data processed in shared memory, constants are stored in constant memory, and the expanded key stored in the texture memory. In the experiments, we encrypt a 256KB picture using 128-bit encryption.

MUMmerGPU (MUM) [30] This application implements a parallel pairwise local sequence alignment which matches query strings consisting of standard DNA nucleotides to reference string for purposes such as genome re-sequencing, genotyping, and metagenomics [30]. The reference strings have been arranged to exploit the texture cache's optimization for 2D locality and are stored as suffix trees in texture memory. In our experiments, we utilize the first 140000 characters of the Bacillus anthracis string. Ames genomes as the reference string and 50000 25-character queries randomly generated applying the complete genome as the seed.

Ray Tracing (RAY) [32] RAY is a approach of rendering graphics with near photo realism. In this application each rendered pixel corresponds to scalar threads in CUDA. It limits 5 levels of reflections and shadows are taken into account, therefore threads behavior rely on what object the ray hits, forcing the kernels susceptible to branch divergence. In our experiments, we simulate rendering of a 256X256 image.

StoreGPU (STO) [33] This application is a library which speeds up hashing based primitives designed for middleware [33]. In this research we decide to use the sliding window implementation version of the MD5 algorithm on an input file of size 192KB. The off-chip memory traffic is minimized by the developers by applying the fast shared memory.

N-Queens Solver (NQU) [31] This application solves a classic puzzle of placing N queens on a NxN chess-board such that no queen can capture another [31]. The N-Queen solver applies a simple backtracking algorithm to try to decide all possible solutions. The search space denotes that the execution time increases exponentially with N.

Table I. Benchmark Properties.

Benchmark	**Abr.**	**Grid Dim.**	**CTA Dim.**	**Total Threads**	**Inst. Count**
Graph Alg.: Breadth First Search [28]	BFS	(128,1,1)	(512,1,1)	65536	17M
AES Cryptography [29]	AES	(257,1,1)	(256,1,1)	65792	28M
MUMmerGPU [30]	MUM	(782,1,1)	(64,1,1)	50000	77M
N-Qqueens Solver [31]	NQU	(223,1,1)	(96,1,1)	21408	2M
Ray Tracing [32]	RAY	(16,32,1,)	(16,8,1)	65536	71M
StoreGPU [33]	STO	(384,1,1)	(128,1,1)	49152	134M

CHAPTER III

NOC ARCHITECTURE

This chapter describes some of the NoC architectural design options which are related to our research. The on-chip interconnection network can be designed in different ways based on the performance and cost. Performance is determined by by bandwidth, latency and path diversity of the network. Cost depends on number of routers and complexity as well as density and length of wires.

A. A Generic NoC Router Overview

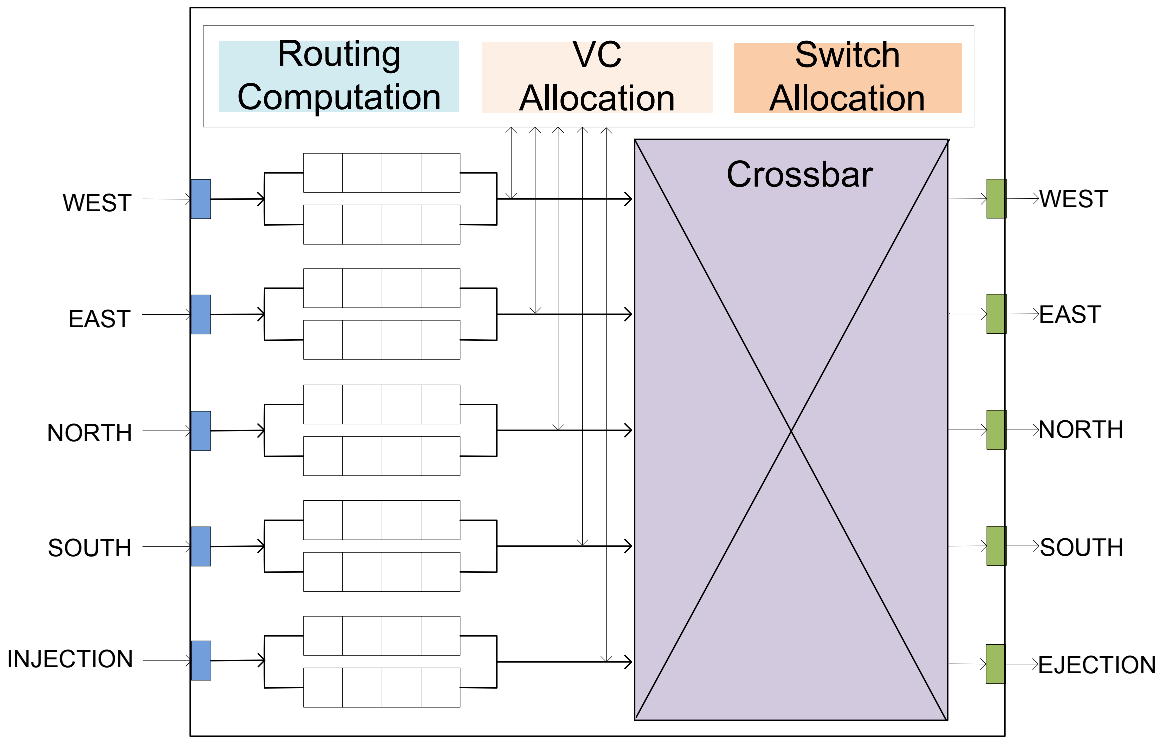

Fig. 7. Generic NoC Router Architecture.

In 2D Mesh topology, there are 5 ports in the router: four from/to the four cardinal directions (NORTH, EAST, SOUTH and WEST) and one from/to local

Processing Element (PE), as depicted in Figure 7. The main building blocks are input buffer, route computation logic, VC allocator, switch allocator, and crossbar. We will describe more in detail of these components in Section B. To achieve high performance, routers process packets with four pipeline stages, which are routing computation (RC), VC allocation (VA), switch allocation (SA), and switch traversal (ST). We will discuss more on pipeline designs in Section C. Due to the stringent area budget of a chip, routers use flit level buffering in a wormhole-switching network as opposed to packet level buffering. Additionally, the buffer is managed with credit-based flow control, where downstream routers provide back-pressure to upstream routers to prevent buffer overflow.

Considering that only the head flit needs routing computation and middle flits always have to stall at the RC stage, low latency router designs parallelize RC, VA and SA using lookahead routing [22] and speculative switch allocation [23]. The functionality of lookahead routing is the same as the normal RC stage, calculating the output ports. However, instead of calculating routing information for the current router, lookahead routing does the same for the downstream router and stores the routing information in the head flit. In this way the RC and VA stages can be overlapped because the VC allocator does not need to wait for the output of the RC logic. Speculative switch allocation [23] predicts the winner of the VA stage and performs SA based on the prediction. If the packet fails to allocate a VC, the pipeline stalls and both the VA and SA stages will be repeated in the next cycle. These two modifications lead to two-stage and even single-stage routers, which parallelize the various stages in the router.

B. Router Component

A NoC router is composed of a set of registers, function units, switches, and control logic that logically and physically implement the routing and flow control functions required to buffer and forward flits to their destinations. Although various router implementation exist, in this section, we examine the components of a typical virtual channel router.

To begin advancing a packet, component *route computation* (marked as Routing Computation in Figure 7) must first be performed to determine the output port to which the packet can be forwarded. Given the output port which obtains from route computation, the packet requests an output Virtual Channel (VC) from the *VC allocator* (marked as VC Allocation in Figure 7). Once a route has been determined and a virtual channel allocated, every flit belongs the packet is relayed over this virtual channel by allocating a timeslot on the switch and output port using the *switch allocator* (marked as Switch Allocation in Figure 7) and forwarding the flit to the corresponding output port during the timeslot. After granting the timeslot during the switch allocation, the flit pass through the *crossbar* (marked as Crossbar in Figure 7) Finally, the output unit forwards the flit to the next router in the packet's route.

C. Router Pipeline

Based on the overview of the Section A, we describe the on-chip router pipeline more in detail in this section. Modern routers are pipelined at the flit level. Every head flit proceed through pipeline stages which perform routing computation and VC allocation (VA) and all flits pass through switch allocation and switch traversal stages. Pipeline stalls occur if a specific pipeline stage cannot be completed in the current cycle time. These stalls halt operation of all pipeline stages because flits must remain proceeding in order.

The routing process begins when the head flit of a packet arrives at the router. Then, the RC stage directs a packet to a proper output port by looking up its destination address. In parallel with routing computation, the first body flit arrives at the router. Next, the VA stage allocates one available VC of the downstream router determined by RC. During this stage, the head flit enters the VA stage, the first body flit passes through the RC stage, and the second body flit arrives at the router. The result of the RC, is input to the VC allocator. If it is successful, the allocator grants a single output VC on the output channel. From this time until the release of the VC by the tail flit, this VC is reserved by this packet. For the next stage, all of the packet-based processing is finished and all remaining control is the flit by flit switch allocation. The SA stage arbitrates input and output ports of the crossbar, and successfully granted flits traverse the crossbar during the ST stage. The head flit starts this SA stage, but it is handled no difference with any other flit. After SA stage, all the flits of the packets pass through the crossbar one by one at the current cycle during ST stage and pass the LT stage to the next router.

D. Topology

Butterfly network topology offers minimal hop count for a arbitrary router radix however it does not have path diversity and requiring very long distance wires. A crossbar interconnect can be considered one stage butterfly and scales quadratically in area as the number of ports grows. From the perspective of a single input port, the butterfly looks like a tree [3]. Each level of the tree contains routing nodes, which unlike the terminal nodes, do not receive or send packets, but merely pass packets along the way. Additionally, each of the channels is unidirectional, flowing from the input to the output nodes.

A 2D torus topology can be fabricated on chip with almost uniformly short wires and provides good path diversity which may lead to a more load balanced design [3]. Mesh interconnects is a special type of torus interconnects. These topologies are attractive for various reasons. These normal physical arrangements are well matched to packaging constraints. At low dimensions, torus have uniformly short wires allowing high frequency operations without repeaters. Logically minimal paths in torus are almost usually physically minimal as well. This physical conservation allows mesh and torus topologies to exploit physical locality between computation nodes. For local traffic patterns, such as each node sending to its neighbor in the first dimension, throughput is much higher and latency is much lower than for other random traffic patterns. Butterfly topology, on the other hand, is unable to exploit this locality. Torus have good path diversity and can have good load balance even on permutation traffic which always has poor performance on other topologies. Also, since all channels in a mesh or torus are bidirectional, bidirectional signaling can be exploited, making more efficient use of wires and pins. The main disadvantage of the mesh topology is its own relatively high latency due to more hop counts. As we will show in Section VII, the benchmarks are not sensitive to latency so that we use mesh network as our baseline while enjoying the simple design of the mesh topology.

CHAPTER IV

NANOPHOTONIC ARCHITECTURE

In this chapter, we present an overview of optical interconnects, including communication components, interconnect patterns, arbitration and flow control.

A. Optical Communication Components

Optical communication structures consist of a laser source (normally located off-chip), waveguides carrying light, and micro-rings or silicon ring resonators that modulate and detect optical signals. Light from the laser source travels unidirectionally in the waveguides with negligible losses. Multiple wavelengths can use the same waveguide with no interference. With dense-wavelength-division-multiplexing (DWDM), up to 128 wavelengths can be generated and carried by the waveguides [41]. Micro-rings are tuned to a particular wavelength and can be used to modulate or detect light of the particular wavelength when placed next to a waveguide. Meanwhile, rings can switch the light from one waveguide to another. The modulation, detection and diversion are controlled by an electrical signal, which tunes the ring between resonance "on" and "off" states. Functioning ring resonators are described in [39] and Figure 8 shows a conceptual optical link.

Ring detection is destructive, which means that an active ring detector removes all the light during the process of detection. Thus, any downstream detectors will not be able to detect the light. In other words, an active detector detects a light signal only when no upstream detector is activated. A ring splitter is used for switching a fraction of light to another waveguide without affecting modulated light signals.

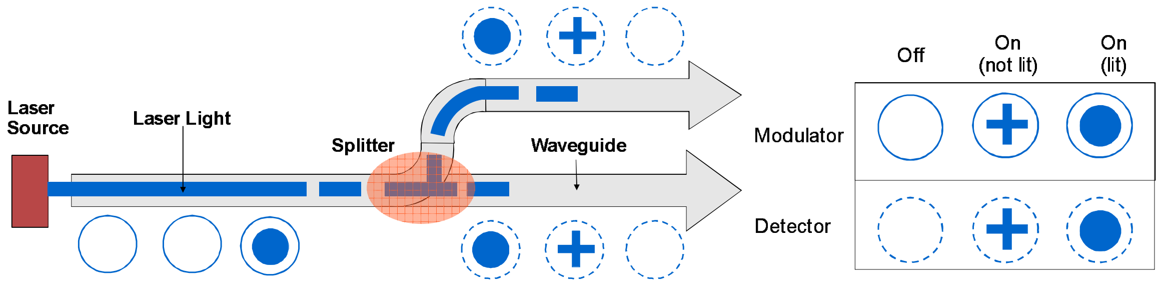

Fig. 8. A Conceptual Optical Link.

B. Interconnect Patterns

In traditional electrical interconnects, each node is connected to its neighboring nodes using separate electrical links, such as a 2D Mesh network, while in optical interconnects nodes are normally attached to a single communication media[1] forming a ring-based network as shown in Figure 9. 64 nodes, each of which contains 4 cores, are connected through unidirectional optical rings. The ring-based optical interconnect falls into two categories: Multiple Write Single Read (MWSR) such as Corona [39], or Single Write Multiple Read (SWMR) such as Firefly [40]. Figure 10 shows these two interconnects. In MWSR, a node can write to all the channels except one specific channel from which the node can read, while in SWMR a node can write to a specific channel from which any other nodes can read. MWSR needs arbitration in the sender side, since a destination node can only receive one light signal at a time. SWMR benefits from not requiring any arbitration in the sender, but introduces extra communication complexity. Considering multiple nodes can read from one given channel in SWMR, a reader should activate its detector. Since ring detection is de-

[1]This single communication media is composed of many separate channels, and each channel consists of a couple of waveguides.

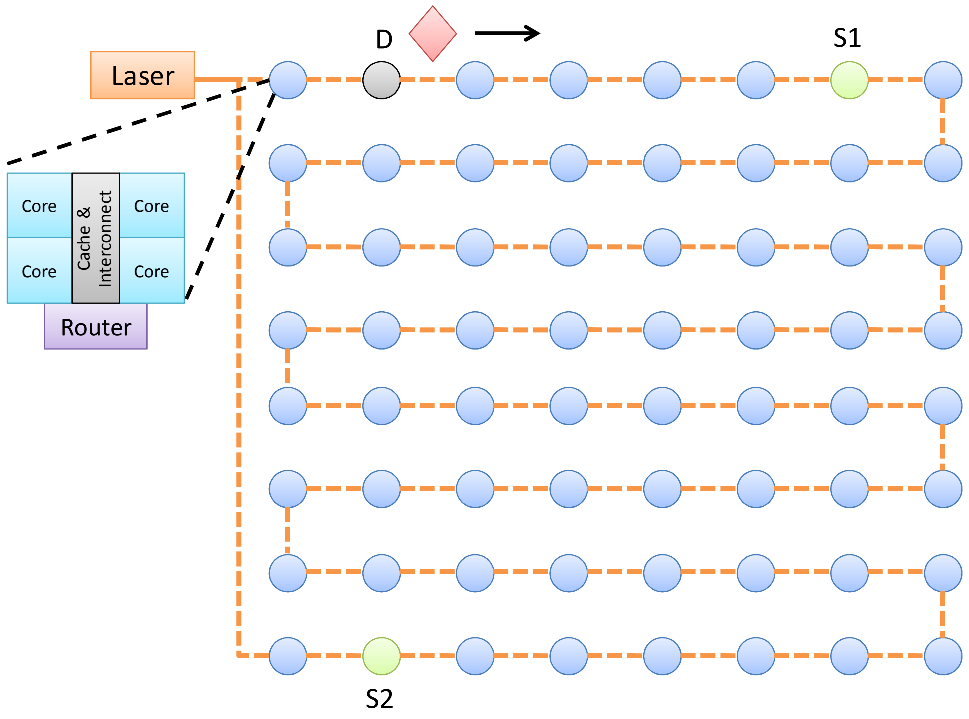

Fig. 9. Ring-Based Network Architecture.

structive, we cannot allow all the nodes to keep their detectors activated all the time. Only the destination node is allowed to open its detector. To handle this situation, before sending data signals, the sender must notify the receiver of the future communication to activate the receiver's detector, which costs extra bandwidth and needs relatively expensive broadcast waveguides. Although our handshake schemes can be applied to both MWSR and SWMR, we choose MWSR as our interconnect pattern for its simplicity and low cost.

C. Arbitration and Flow Control

With limited on-chip channel and buffer resources, arbitration and flow control become the most critical factors in the NoC design. In nanophotonic interconnects,

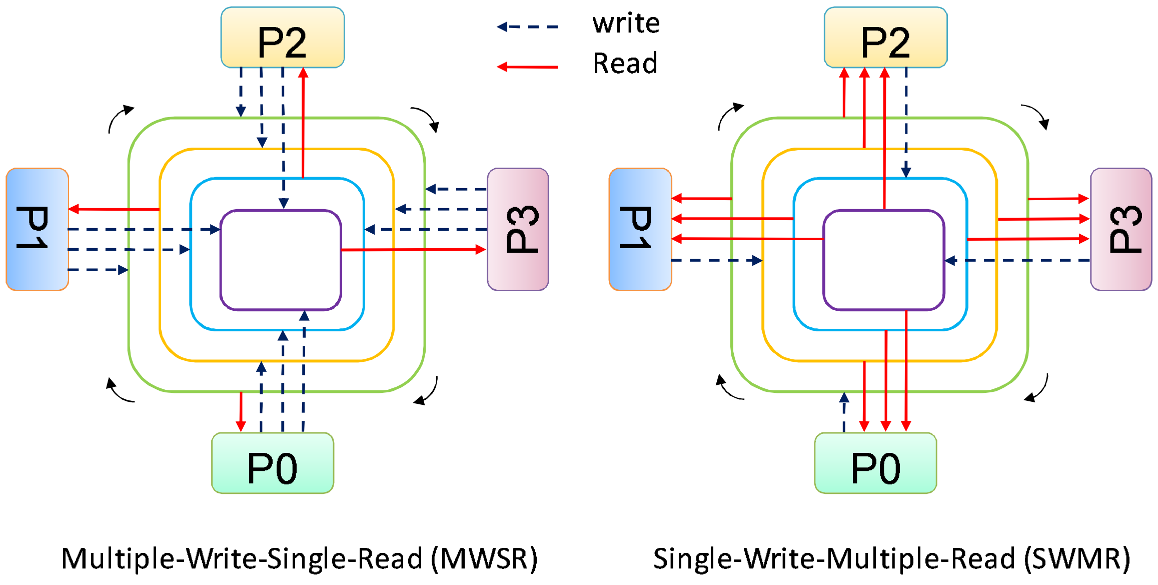
write
Read
P0
P1
P2
P3
Multiple-Write-Single-Read (MWSR)
Single-Write-Multiple-Read (SWMR)

Fig. 10. MWSR and SWMR.

packets traverse through optical channels in a wave-pipelined manner, which allows a single optical channel to be divided into several segments, and each segment is similar to a single-cycle bus. For example, on a 576 mm^2 chip with 64 nodes and a 5GHz clock, the round trip time for an optical channel is 8 cycles [39], so it can be divided into 8 segments. Considering the specific characteristics of optical channels, the arbitration of a shared optical channel can take two methods: global arbitration or distributed arbitration. Global arbitration is like a bus-based interconnect. In the whole round trip time, only one sender and one receiver will use the channel. Distributed arbitration considers the wave-pipelined manner of packet transmission. If two packets are not overlapped in the same segment at the same time, they can traverse in the same optical channel. Prior work [41, 40] adopts token-based arbitration, in which a photonic token represents the right of transmitting packets on a channel. Token channel is proposed for global arbitration, while token slot and token stream are designed as distributed arbitration. Traditional electrical on-chip interconnects hire credit-based flow control, in which upstream routers keep a record of the number of free buffers in downstream routers. When a router forwards a flit to the next hop, it sends a credit backward to its upstream router. Inherited from credit-based flow control, all the above token-based arbitration schemes integrate the credit information into the arbitration token.

CHAPTER V

GPGPU-SIM ARCHITECTURE

In this chapter, we present a detail illustration of the three-layer GPGPU-Sim infrastructure we simulated.

A. CUDA-Sim

CUDA-Sim is a functional simulator that executes PTX kernels generated by NVCC or OpenCL compiler. This is the first layer of the GPGPU-Sim, which runs CUDA applications and makes the API calls. The API calls which is in the *libcuda*, as we stated in Section B. It implements the PTX functional simulation engine for GPGPU-Sim. The interface between CUDA-Sim and GPGPU-Sim is contained in the *libcuda*. In the CUDA-Sim, instructions used for functional execution are represented as an instruction object contained within a function object. Instructions are decoded by calling appropriate functions and passing correct information, which provides basic information to the timing model about the next instruction a thread will execute.

B. GPGPU-Sim

Figure 3 also depicts an overview of the system we simulate in this research. The GPGPU-Sim is a performance simulator that simulates the timing behavior of a GPGPU. In the CUDA programing model, we consider GPGPU as a co-processor onto which an application running on a CPU can generate a massively parallel compute kernel. The kernel is comprised of a grid of scalar threads. Each thread is marked a unique ID, which can be used to help divide up work among all threads. Within a single grid, threads are grouped into blocks, which are also called Cooperative Thread

Arrays (CTA). Threads have access to common fast memories, which are also referred to as shared memory within a grid.

The GPGPU-Sim models the pipeline stage with 6 logical stages (fetch, decode, execute, mem1, mem2 and write-back). The GPGPU-Sim employs the immediate post-dominator reconvergence where several scalar threads within a warp evaluate a branch as “taken” and others are “not taken”. Threads running on the GPGPU in the CUDA programming models can access several memory regions, such as local, global, texture, constant and shared [2]. The GPGPU-Sim models accesses to each of those memory regions. Global and local memory accesses usually require off-chip memory accesses in the baseline configuration. For the constant cache, the GPGPU-Sim only allows single cycle access if every thread in a warp is requesting the same memory address. The GPGPU-Sim implements a 4D blocking address scheme as described in [25] for the per-core texture cache, which basically permutes the bits in requested addresses to improve spatial locality. Concurrent memory accesses from threads, which belongs to a single warp to a localized region, are coalesced into fewer wide memory accesses to promote DRAM efficiency. In order to alleviate the DRAM bandwidth bottleneck, which most applications face, a common technique applied by CUDA programmers is to load frequently accessed data into the fast on-chip shared memory [26].

C. Intersim

The Intersim is the interconnection network simulator adopted from Bill Dally’s BookSim [14]. This simulator is developed using C and C++, and it can be downloaded from Stanford University [36]. It is a cycle-accurate network simulator that models all NoC router pipeline delays, components, topologies and wire latencies, which we presented in Section III.

The Intersim has a command line interface for defining various parameters of an interconnection network. In other words, the user can customize the network size, packet size distribution, buffer size, routing algorithm, scheduling strategy, packet injection rate, traffic time distribution, traffic pattern, and hot-spot traffic distribution. The simulator allows NoC evaluation in terms of latency, throughput, chip area and power consumption. This information is delivered to the user both in terms of average and per-communication-basis results. Additionally, the user is allowed to collect different evaluation metrics including global average throughput, the total number of received packets/flits, total energy consumption, max/min global delay, per-communications delay/throughput/energy, and etc.

CHAPTER VI

DESIGN AND IMPLEMENTATION

In this chapter, we first present the drawback of conventional mesh network in GPGPU architecture design. Then, we propose adding nanophotonic ring network to mesh network design scheme.

A. Conventional Mesh Drawback

As we discussed in Section D, mesh network is common in NoC design because this regular physical arrangement is well matched to packaging constraints [3]. One drawback of mesh network is that it has a larger hop count than other logarithmic networks. This increases the pin cost of the network and gives it a slightly higher latency than the minimum bound. One should note, in exchange, that some increase in hop count is required for path diversity.

Due to the physical nature of mesh network, the nodes placed in the center of the mesh have higher workload rates because more packets are forwarded through these "central" nodes. This causes workload in mesh network to be slightly unbalanced. On the other hand, because the special communications in GPGPU architecture is not considered, the "pure" mesh network is not quite suitable for a throughput friendly design.

B. Nanophotonic Ring Network

In Section II we discussed the traffic pattern in GPGPU architecture. The most communication is the traffic between memory controllers and shader cores while cores to cores communication is rare. Due to the physical nature of mesh network, the interconnection network design, which only employs mesh network, does not consider these traffic patterns. Due to the inefficiency of mesh network, we propose to add a nanophotonic ring to the mesh network to improve throughput. This nanophotonic ring connects all memory controllers in the mesh network, as depicted in Figure 11. The mesh network is the original topology we used in this research. The orange curve is the nanophotonic ring which connects all memory controllers.

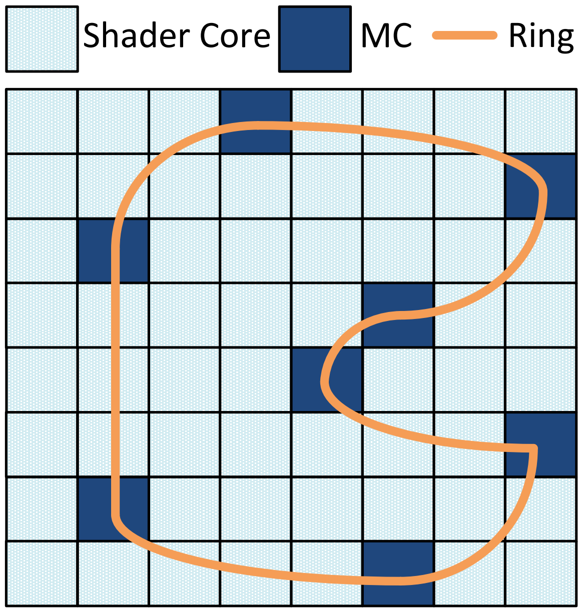

Fig. 11. Nanophotonic Ring Added to Mesh Network.

Because every communication in nanophotonic ring network is only one hop away, as we mentioned in Section B, the hop count of every communication between memory controller and shader core is dramatically decreased. By adding the nanophotonic ring we can reduce the average hop count for routing a packet. Therefore we improve the network throughput.

There are four different types of traffics in detail implementation. Shader core to shader core traffic, though it is rare, employs normal Dimension Order Routing (DOR) algorithm such as XY routing. For shader core to memory controller traffic, it should be first sent to the nearest memory controller and then relays the traffic to the destination through the nanophotonic ring network. It is a similar case if the source is memory controller and destination is shader core. However, this is in a reversed way which first routes the traffic through the nanophotonic ring network to the nearest memory controller. Then we employ normal DOR routing to send the traffic to shader core. If both the source and destination are memory controllers, we directly send the packet to the destination through the ring within one hop.

CHAPTER VII

EXPERIMENTAL RESULTS

In this chapter, we present experimental results for our throughput-effective on-chip network for GPGPU architecture. We begin with showing our simulation configuration, then analyze several important metrics of six CUDA benchmarks, and finally present the impact of adding a nanophotonic ring network.

A. Methodology

Table II describes the GPGPU-Sim simulator configuration we simulated in the experiments. We use a modified version of GPGPU-Sim which was presented in Section V. The modification is to bring a nanophotonic ring network into the interconnection network. To simulate the mesh interconnection network, we use a detailed cycle-accurate network simulator, Intersim Section C. We show the interconnection configuration used in our simulations in Table III.

B. CUDA Workloads Analysis

In this section we analyze several important characteristics of the six CUDA benchmarks used in this research which was introduced in Section C. Figure 12 illustrates the classification of each benchmark's memory access. As we mentioned before in Section I, a thread can access shared memory, constant memory, texture memory, local memory and global memory. Note that "parameter" memory denotes to parameters that pass through the GPU kernel call. These are considered as cache hits. There is a huge variation in different types of memory instructions executed among benchmarks: for AES nearly 90% of accesses are to shared memory while for BFS half of accesses

Table II. Hardware Configuration.

Number of Shader Cores	56
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of CTAs / Core	8
Shared Memory / Core (KB)	16 (16 banks, 1 access/cycle/bank)
Constant Cache Size / Core (KB)	8 (2-way set associate. 64B lines LRU)
Texture Cache Size / Core (KB)	64 (2-way set associate. 64B lines LRU)
Number of Memory Channels	8
Bandwidth per Memory Module	8 (Bytes/Cycle)
Average Memory Fetch Latency	16
DRAM Request Queue Capacity	32
Memory Controller	Out of Order (FR-FCFS)
Branch Divergence Method	Immediate Post Dominator
Warp Scheduling Policy	Round-Robin among Ready Warps

are to global and the rest are to parameter. For RAY more than 80% are accesses to constant memory while for MUM over 50% are accesses to texture memory.

Figure 13 shows the ratio of load/store instructions executed in each benchmark. The variation of the ratio of each benchmark is not that significant as “memory instructions breakdown”. High ratio of load instruction benchmarks, namely BFS, MUM, and STO, are more sensitive to memory bandwidth because load instructions usually lie in the critical path. We can also observe this phenomena reflected in Figure 14.

Table III. Interconnect Configuration.

Topology	2D Mesh
Routing Algorithm	Dimension Order
Routing Delay	4 cycles
Virtual Channels	4
Virtual Channel buffers	8
SW Allocator	iSLIP
Allocation Iteration	1
Input Speedup	1
Flit Size (Bytes)	16

Figure 14 shows the performance of our baseline configuration which was described in Table II and Table III. The metrics are measured in terms of scalar instructions per cycle (IPC) which is also normalized for comparison purposes. In terms of IPC, the higher value is better. As a comparison, we also show the performance assuming a perfect memory system that has no memory access latency. Note that the maximum achievable IPC for the configuration is 448 (56 shader core X 8 wide pipelines).

We also show the effects on IPC of adding caches to the whole system in Figure 15. The first two bars denote the relative normalized IPC of adding 32KB or 64KB L1 cache to each shader core. The last two bars illustrate the effects of adding 128KB or 256KB L2 cache to each memory controller while keeping the 64KB L1 cache in shader cores. Among six benchmarks, we can see MUM, RAY and STO obtain significant IPC improvement due to addition of extra cache (37%, 10% and 11%, respectively). This is because these benchmarks have the highest portion of global

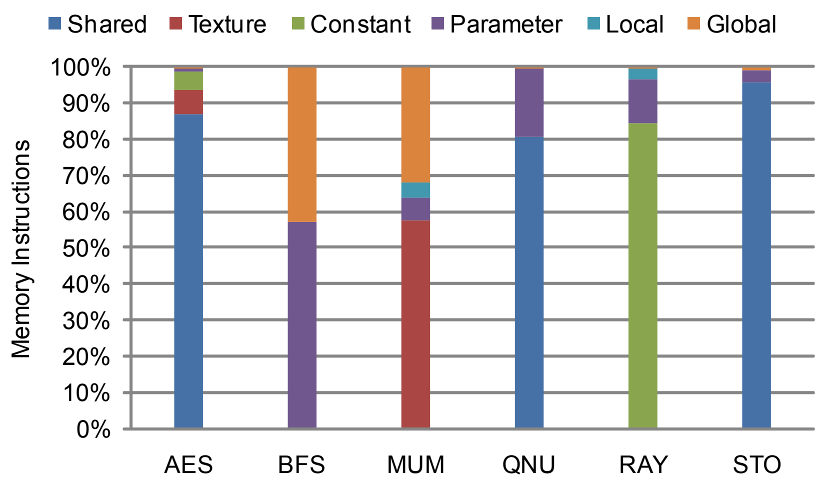

Fig. 12. Memory Instructions Breakdown.

memory instructions among all of them. However, only BFS experiences a slowdown due to the the method write misses, and evictions of dirty lines are resolved. The baseline writes to memory merely causes the memory controller to read data out of off-chip memory if a part of cache block is modified. When we bring caches for global and local memory accesses, a write miss prevents a warp from being scheduled until the 16B is read from off-chip memory. Moreover, the whole cache line is written to off-chip memory when a dirty line is evicted even if only a single word of that cache line is changed. These benchmarks which make extensive use of shared memory, such as AES and QNU, do not access significantly to caches.

C. Nanophotonic Ring Network

In this section, we evaluate the design proposed in Section B. Figure 16 illustrates the normalized interconnection network latencies of six CUDA benchmarks. In terms of network latency, the less value is better. As we mentioned in Section B, the dominant part of communication in GPGPU architecture is core-to-memory-to-core traffic. Since we connect all memory controllers with a nanophotonic ring, which

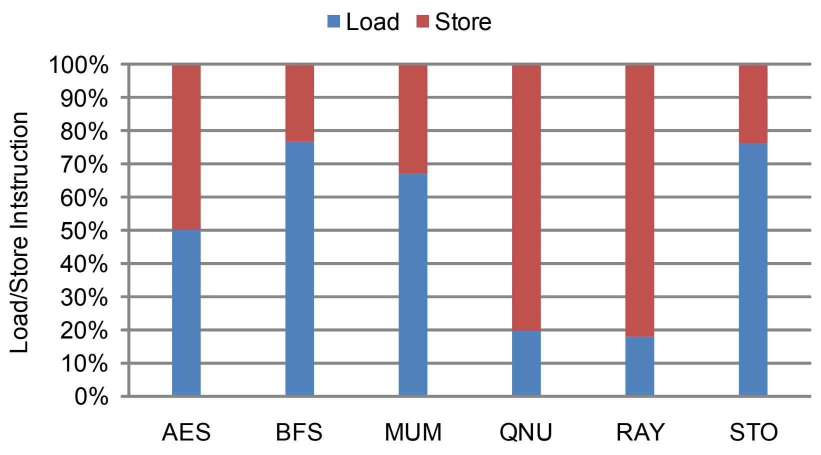

Fig. 13. Load/Store Instruction Analysis.

means every memory controller to memory controller is one single hop away, we can achieve significant throughput improvement. The more load instructions access the DRAM, the more the nanophotonic ring network is improved. The simulation results we can see here meet the expectation in Figure 13. For BFS, MUM and STO, we achieved 27%, 32% and 18% throughput improvement, respectively.

Figure 17 shows the normalized IPC values of proposed nanophotonic ring network compares to baseline GPU architecture. Simulation results demonstrate that employing nanophotonic ring network improves system performance by 5% in average and up to 11.5% (MUM). This meets our expectation from previous experiment results in Figure 16 We obtain much improvement in system performance due to dramatic reduction of the interconnection network latencies by adding a nanophotonic ring network. This is because a request always sends to the nearest MC then relays to the destination MC in one cycle. Therefore, we physically reduce the average hop count. Since requests can arrive to destinations sooner, the GPU will not waste much execution time to wait for the data to be replied back.

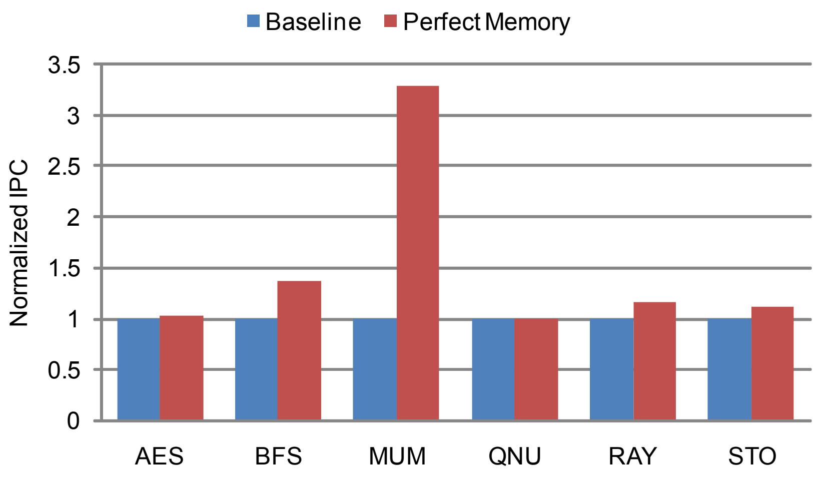

Fig. 14. Baseline Performance.

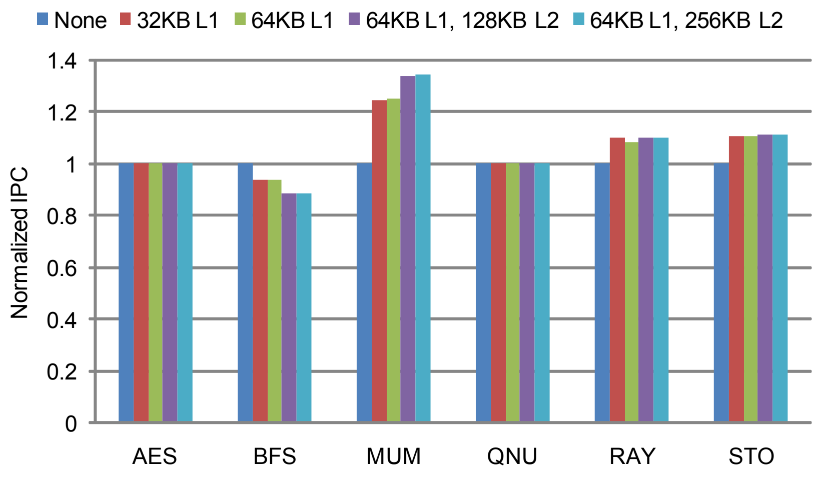

Fig. 15. Effects of Adding An L1 or L2.

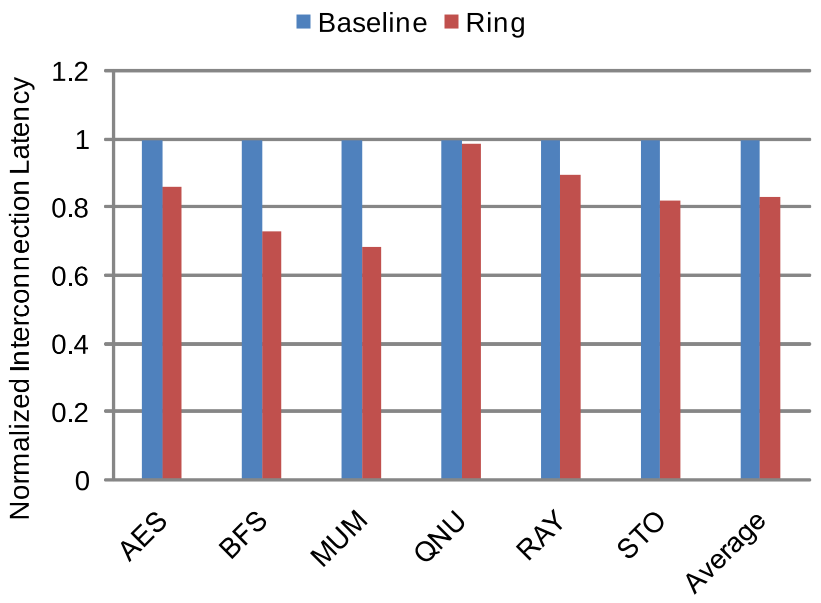

Fig. 16. Normalized Interconnection Latencies.

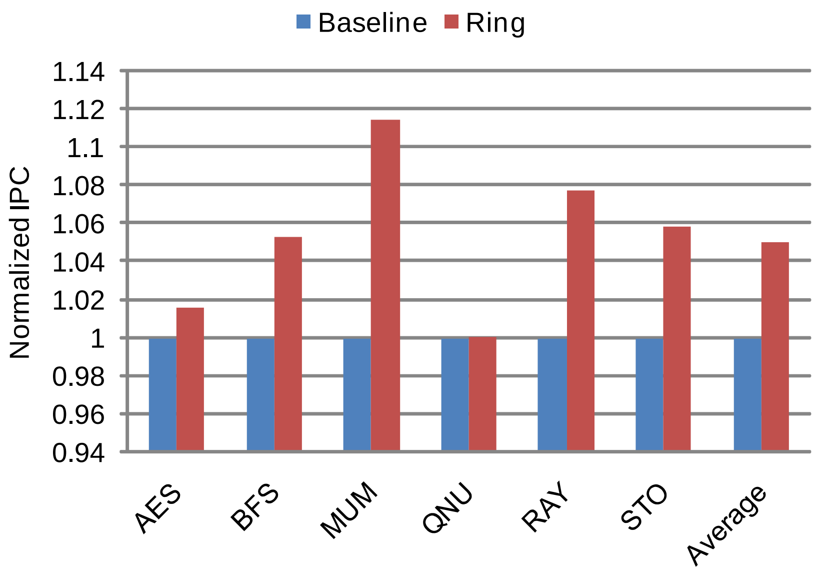

Fig. 17. Normalized IPC.

CHAPTER VIII

RELATED WORK

Rigel [6] is an accelerator which provides a more flexible programming model compared to CUDA and chooses a MIMD model rather than SIMT. The Cell [7] architecture's NoC design is an example of making trade-offs between network's area and latency. It consists of a controlling processor and a set of SIMD co-processors each with independent instruction memory and program counters. The Cell researchers chose a ring over a crossbar to meet their area and power constraints [8]. Merrimac [9] and Imagine [10] are both streaming processor architectures provided by Stanford University. Bakhoda et al. analyze popular CUDA workloads using GPGPU-Sim [11]. In our research, we use GPGPU-Sim to simulate the GPU behavior and try to improve throughput of the GPU architecture.

Bakhoda et al. propose half router to reduce the fabricate chip area while maintaining comparable performance for the GPGPU on-chip networks in [5]. Their idea is to provide some simpler routers among all routers and MCs to reduce the chip area. Area savings is 10% if we assume 25% reduction in full router and 75% of half router. Existing graphics-oriented GPU simulators include Qsilver [12], which does not model programmable shaders, and ATTILLA [13], which focuses on graphics specific features. The [14] uses CUDA to speed up a variety of relatively easily parallelizable scientific applications. They explore the use of conventional code optimization techniques and take advantage of the various memory types available on NVIDIA's 8800GTX to achieve the speedup.

Khailany et al. examine VLSI costs and performance of a stream processor as the number of streaming clusters and ALUs per cluster scales [20]. The benchmarks they use also have a huge portion of ALU operations to memory references which is a property that eases memory reqirements of streaming applications. The UltraSPARC T2 microprocessor [21] is a multi-threading, multi-core CPU which belongs to the SPARC family, with each core capable of running 8 threads concurrently. They have 4, 6 and 8 core variation versions and all have a crossbar between the L2 and the processor cores. Although the T1 and T2 support many concurrent threads compared to other contemporary CPUs, this number is very small compared to the number on a high end contemporary GPU (for example, the NVIDIA Geforce 8800 GTX supports 12,288 threads per chip).

Significant research in the Network-on-Chip (NoC) and application specific design communities addresses the challenge of how best to map tasks to physical cores on-chip [15]. There are also much research has focused on the impact of different memory technologies and their respective trade-offs in providing adequate off-chip latency and bandwidth [16, 17]. [18] gives significant research on different memory controller placement in many-core CMPs. They propose various memory controller placements and routing of many-to-few-to-many traffic in on-chip networks. Prior work [19] illustrated that if the pipeline delay of adaptive routing is considered, then the O1Turn routing algorithm outperformed adaptive routing algorithm. Moreover, adaptive routing algorithm can only be used for response packets - deterministic routes are necessary to preserve order of the read/write request packets to the memory controllers.

CHAPTER IX

CONCLUSIONS

In this research, we exploit the many-to-few-to-many traffic behavior of GPGPU architecture and attempt to optimize the design of on-chip interconnection network for GPGPU architecture. We first analyze the application characteristics of six CUDA benchmarks. Because different applications vary in load/store ratio, read request/write request ratios and number of accesses to shared memory, constant memory, texture memory, local memory and global memory, it is important to know these key metrics to further improve the GPGPU design. Besides that, we propose to add a nanophotonic ring network to the conventional mesh network to further explore GPGPU performance. Our experiment results show that the proposed scheme reduces the interconnection network latency and improves system IPC for the six CUDA benchmarks. We believe the observations made in this research provides meaningful guidance for indicating future GPGPU architecture and CUDA research.

REFERENCES

[1] Advanced Micro Devices, Inc. Press Release: *AMD Delivers Enthusiast Performance Leadership with the Introduction of the ATI Radeon HD 3870 X2*, 28 January 2008.

[2] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2.3 edition, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIAOpenCLProgrammingGuide.pdf, 2007.

[3] W. J. Dally and B. Towles, Route Packets, Not Wires: On-Chip Interconnection Networks, in *Proceedings of DAC*, 2001, pp. 684-689.

[4] J. D. Owens, W. J. Dally, R. Ho, D. N. Jayasimha, S. W. Keckler, and L. S. Peh, Research Challenges for On-Chip Interconnection Networks, *IEEE Micro*, vol. 27, no. 5, pp. 96-108, 2007.

[5] Bakhoda, A., Kim, J., Aamodt, T.M., Throughput-Effective On-Chip Networks for Manycore Accelerators, in *Proc. MICRO*, 2010.

[6] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. Patel. A task-centric memory model for scalable accelerator architectures. *IEEE Micro Special Issue: Top Picks 2010*, Jan./Feb. 2010.

[7] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, vol. 26, pp. 10-23, 2006.

[8] D. Krolak. Cell Broadband Engine EIB bus. http://www.ibm.com/developerworks/power/library/paexpert9/, Retrieved Sept. 2010.

[9] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proc.* 2003 ACM/IEEE Conf. on Supercomputing, page 35, 2003.

[10] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine stream architecture. In *Proc. 31st Int'l Symp. on Computer Architecture*, page 14, 2004.

[11] A. Bakhoda., Analyzing cuda workloads using a detailed GPU simulator, in *Proc. ISPASS*, 2009.

[12] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 85-94, 2004.

[13] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. *Int'l Symp. on Performance Analysis of Systems and Software*, pages 231-241, March 2006.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 73-82, 2008.

[15] K. Srinivasan and K. Chatha. A technique for low energy mapping and rougin in network-on-chip architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005

[16] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the International Symposium on Computer Architecture*, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[17] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *International Symposium on High Performance Computer Architecture*, pages 109-120, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[18] D. Abts, N.E. Jerger, J. Kim, D. Gibson, and M. Lipasti, Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs, in *International Symposium on Computer Architecture*, 2009.

[19] D. Seo, A. Ali, W. T. Lim, N. Rafique, and M. Thottethodi. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proc. of the International Symposium on Computer Architecture(ISCA)*, pages 432-443, 2005.

[20] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens, and B. Towles. Exploring the VLSI scalability of stream processors. In *Proc. 9th Int'l Symp. on High Performance Computer Architecture*, page 153, 2003.

[21] Sun Microsystems, Inc. *OpenSPARCTM T2 Core Microarchitecture Specification*, 2007.

[22] M. Galles, Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER chip, in *Proceedings of Hot Interconnect* 4, 2009, pp. 141-146.

[23] L. S. Peh and W. J. Dally, A Delay Model and Speculative Architecture for Pipelined Routers, in *Proceedings of HPCA*, 2001, pp. 255-266.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39-45, 2008.

[25] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Proc. 24th Int'l Symp. on Computer Architecture*, pages 108-120, 1997.

[26] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Z. Ueng, J. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc. 6th Int'l Symp. on Code Generation and Optimization (CGO)*, pages 195-204, April 2008.

[27] Infineon. 256Mbit GDDR3 DRAM, Revision 1.03 (Part No. HYB18H256321AF). http://www.infineon.com, December 2005.

[28] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, pages 197-208, 2007.

[29] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC 2007: Proc. of IEEE Int'l Conf. on Signal Processing and Communication*, pages 65-78, 2007.

[30] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[31] Pcchen. N-Queens Solver. http://forums.nvidia.com/index.php?showtopic=76893.

[32] Maxime. Ray tracing. http://www.nvidia.com/cuda.

[33] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage

systems. In *Proc. 17th Int'l Symp. on High Performance Distributed Computing*, pages 165-174, 2008.

[34] Int'l Technology Roadmap for Semiconductors. 2008 Update. http://www.itrs.net/Links/2008ITRS/Home2008.htm.

[35] Open64. The open research compiler. http://www.open64.net/.

[36] Booksim interconnection network simulator. http://nocs.stanford.edu/booksim.-html.

[37] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6Tbits/s 3.6GHz Single-Cycle NoC Router with a Novel Switch Allocator in 65nm CMOS. In *Proceedings of ICCD*, pages 63-70, 2007.

[38] H. S. Wang, L. S. Peh, and S. Malik. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *Proceedings of MICRO*, 2003.

[39] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. L. Binkert, R. G. Beausoleil, and J. H. Ahn. Corona: System Implicationsof Emerging Nanophotonic Technology. In *ISCA*, pages 153-164, 2008.

[40] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang, and A. N. Choudhary. Firefly: Illuminating Future Network-On-Chip with Nanophotonics. In *ISCA*, pages 429-440, 2009.

[41] D. Vantrease, N. L. Binkert, R. Schreiber, and M. H. Lipasti. Light Speed Arbitration and Flow Control for Nanophotonic Interconnects. In *MICRO*, pages 304-315, 2009.

VITA

Name:	Wen Yuan
Address:	Texas A&M University
	H.R. Bright Building, Room 336
	College Station, TX 77843
Email Address:	yuanw@cse.tamu.edu
Education:	B.S., Computer Science, National University of Defense Technology, 2009
	M.S., Computer Science, Texas A&M University, 2012

The typist for this thesis was Wen Yuan.