Improving the Global Memory Efficiency in GPU-Based Systems

A Dissertation Presented
by
Amir Kavyan Ziabari
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

December 2016
# Contents

List of Figures iv
List of Tables viii
List of Acronyms ix
Acknowledgments xi
Abstract of the Dissertation xii

## 1 Introduction

1.1 Background and Motivation ........................................ 3
  1.1.1 Existing GPU Architectures .................................. 3
  1.1.2 Future Scalable GPU Architectures ......................... 7
  1.1.3 MultiGPU Systems ........................................... 8
1.2 Contributions .................................................... 12
1.3 Thesis Outline .................................................... 13

## 2 Background

2.1 OpenCL Runtime and Programming Model ........................... 15
  2.1.1 Execution Model on GPU architecture ....................... 15
  2.1.2 OpenCL Memory and Consistency Model ..................... 17
2.2 GPU Organization and Design .................................... 18
  2.2.1 Command Processor and Dispatcher ......................... 19
  2.2.2 Compute Unit Design ...................................... 19
  2.2.3 Global Memory ............................................ 21
2.3 Interconnection Networks ........................................ 22
  2.3.1 Network Design Considerations ............................. 22
  2.3.2 Building Blocks ........................................... 23
  2.3.3 Topology and Routing .................................... 27
2.4 Silicon-Photonic Technology ..................................... 29
  2.4.1 Silicon-Photonic Links .................................... 30
  2.4.2 Silicon-Photonic Integration ............................... 31
2.5 Existing Cross-Device Memory Management Options ............. 31
List of Figures

1.1 The addition of two vectors using GPUs. Each GPU threads calculates one element of the output vector at a time .................................................. 2
1.2 High level overview of the contemporary GPU accelerator. This design is based on AMD Southern Islands 7970 HD GPU .................................................. 4
1.3 Potential improvements for various workloads running on a 32-CU GPU, with an ideal NoC against an electrical NoC .................................................. 5
1.4 High-level with of a multi-GPU system design, leveraging a conventional PCIe off-chip interconnect ................................................................. 9

2.1 Software entities defined in the OpenCL programming model. An ND-Range is formed of workgroups, which are, in turn, sets of work-items executing the same OpenCL C kernel code ................................................................. 16
2.2 A multi-GPU system with an original ND-range is partitioned into two separate ND-ranges for execution on two GPU devices ......................................... 17
2.3 The design of a AMD’s GCN compute unit [1]. The compute unit contains five processing pipelines. The vector memory pipeline, has 4 SIMD units, capable of processing one instruction of 64 work-items at each cycle. The colored pipelines, scalar and vector memory (against the gray-scale units), are involved in the processes that lead to a global memory access .................................................. 20
2.4 A high-level view of a bus-based shared-medium network with centralized arbitration ................................................................. 23
2.5 Two virtual channels multiplexed across a physical channel ................................................................. 24
2.6 Baseline microarchitecture of a classical router [2] .................................................................................... 26
2.7 Logical layout of different topologies. Routing determines which path a message takes from source to destination within the topology. One example of a route is identified in each topology (in red) ................................................................. 28
2.8 Photonic Link Components – Two point-to-point photonic links implemented with Wavelength-Division Multiplexing (WDM) ................................................................. 30
2.9 Memory-copy based approach used to manage memory for GPU applications ................................................................. 32
2.10 Remote access approaches used to manage memory for GPU applications ................................................................. 33
2.11 Different commercial memory technologies available on current GPU devices ................................................................. 36
2.12 The Multi2sim simulation philosophy for architectural modeling. The framework provides disassemblers, emulators, and detailed timing simulators, which are used to model different architectures (including x86 CPU, and AMD Southern Islands GPU). The visual tool is an additional feature used for detailed presentation of the simulation status/results. 38

2.13 Extracting the detail of the GPU device based on the chip die. 40

3.1 The use of repeaters along the wire to pipeline the signal transfer to avoid signal loss. 49

3.2 Design of a crossbar NoC topology, using global MWSR buses for the GPU architecture. 50

3.3 The Clos topology used as the interconnection network in the GPU architecture. 51

3.4 Layout of a Clos network with an intermediate set of routers placed in the center of the chip. All bidirectional links associated with the highlighted routers are shown. 52

3.5 A 6-ary 4-fly conventional butterfly topology for the GPU architecture. 53

3.6 Design of a Mesh topology, with diagonal L2 cache placement. 54

3.7 Deadlock, and deadlock avoidance using XY routing in 2D mesh [3]. 55

3.8 The logical layout of the Cmesh topology for the GPU architecture. 56

3.9 Performance Comparison of various NoC designs. 59

3.10 Power breakdown of various NoC designs. 61

3.11 Energy-per-Delay Product for each NoC design. 62

3.12 Breakdown of the different types of communications between cache units. 63

3.13 (a) The L1-to-L2 network for MeshX2, (b) The L2-to-L1 network for MeshX2. The use of XY routing leads to several unused links. These links were removed to reduce the static power consumption and switch radices. 64

3.14 The network for both directions in a ButterflyX2. 65

3.15 Performance comparison of various parallel designs against baseline NoCs. 67

3.16 Dynamic and static power breakdown for various parallel symmetric and asymmetric designs. 68

3.17 Design comparison based on the Energy-Delay Product metric. 69

3.18 Overall comparison of baseline and proposed designs using the EDP metric. 70

4.1 SWMR and MWSR busses as building blocks for photonic crossbar topology. 76

4.2 Use of Fast-Forwarding for arbitration in photonic links. 78

4.3 Physical layout of the silicon-photonic crossbar NoCs for the target GPU chips. 80

4.4 Speedup and bandwidth for a 32-CU GPU with different crossbars. Ticks on the x-axis follow the pattern $T$-X-$N$, where $T$ refers to the type of technology (electrical = E; photonic = P), X is the type of crossbar (MWSR or SWMR), and $N$ refers to the channel width in bytes. For electrical NoCs three different link bandwidths (from left to right 16, 32 and 72) are considered. 83

4.5 Breakdown of the total power and energy-delay$^2$ product of NoCs in a 32-CU GPU. The ED$^2$P is normalized to an electrical MWSR crossbar NoC with a 32-byte channel width ($E$ – $MWSR$ – 32). 84

4.6 The breakdown of the ratio of different types of data that traverse the network in the L1-to-L2 direction. More than 80% of the bytes in the L1-to-L2 traffic are due to small 8-byte control messages. 86
5.9 The size of each HMD and SMD for different stack sizes, in a system with 2 HMDs and 4 GPUs. .................................................. 122
5.10 The physical placement of the SMD on HBM. .................................................. 124
5.11 Breakdown of runtime in a system with 1, 2, and 4 GPU devices, when using PCIe
or NVLink peer-to-peer connections (P2P) for mem-copy (MC), zero-copy (ZC),
or Unified Memory Hierarchy (UMH) approach. The results for each system is
normalized to the execution time of application on a system that uses MC and PCIe. 126
5.12 The impact of UMH with peer-transfer in a system with 4 GPUs .......................... 127
5.13 The amount of data that is flushed from caches to the GPU memories during Fi-
nal Flush (FF), and written back to the host memory. (R) denotes the data struc-
tures only read by GPU, and (W) denotes the data structures that GPU writes into.
Software-based UM, memcpy, and zero-copy synchronize the CPU by performing
FF+WR, FF+W, and FF, respectively. UMH performs dynamic synchronization. 129
List of Tables

2.1 **AMD Radeon HD 7970 GPU specification.** ........................................... 39

3.1 **Network Configuration** - Networks are sized to support two types of messages that create the GPU traffic. \( N_C \) = number of channels, \( b_C \) = bits/channel, \( N_{BC} \) = number of bisection channels, \( N_R \) = number of routers, \( H \) = number of hops along data path, \( T_R \) = router latency, \( T_{GR} \) = average latency of global connections (if used), \( T_C \) = channel latency, \( T_{TC} \) = latency from cache units to the concentration switch, \( T_S \) = serialization latency, \( T_0 \) = zero-load latency. Both types of messages are considered in the latency calculations. The latency values are separated by ",". ........................................... 57

3.2 **Workloads from the AMD APP SDK.** .................................................. 58

3.3 Bisection bandwidth for symmetric and asymmetric designs for the targeted topologies, butterfly, mesh and Cmesh. ......................................................................................... 66

4.1 **Energy Projections for Photonic links based on prior work** [4, 5, 6]. – \( Tx = \) Modulator driver circuits, \( Rx = \) Receiver circuits, Dynamic energy = Data-traffic dependent energy, Fixed energy = clock and leakage We consider 20% (conservative projection) and 30% (aggressive projection) laser efficiency. ........................................... 79

4.2 **Projected/Measured Optical Loss per Component** [4, 5, 6]. We consider -17 dBm (a conservative projection) and -20 dBm (an aggressive projection) for the photodetector sensitivity. .................................................. 80

4.3 **The workloads selected from the AMD APP SDK.** ................................. 81

4.4 The number of global buses required for topologies with 32-CU GPUs – DOWN = home node is a L1 unit, and UP = home node is a L2 unit. The channel width is 72 bytes. ......................................................................................... 82

4.5 Number of global buses required for the hybrid topology support 32-CU GPUs – DOWN = home node is a L1 unit, and UP = home node is a L2 unit. The channel width is 72 bytes. ......................................................................................... 87

4.6 Number of global buses required for Hybrid topology in 128-CUs GPU – DOWN = home node is a L1 unit, and UP = home node is a L2 unit. The channel width is 72 bytes. ......................................................................................... 93

5.1 **State transitions defined by the NMOESI protocol.** ............................... 111

5.2 **Workloads from the AMD APP SDK. The number in the parentheses indicates the input argument to the benchmark.** .............................................. 121

5.3 **Simulation Parameters.** ......................................................................... 125
List of Acronyms

API  Application Programming Interface.
APU  Accelerated Processing Units.
CMPS  Chip Multi-Processors.
CUDA  Compute Unified Device Architecture.
CUs  Compute Units.
DDR  Double Data Rate.
DOR  Dimension-Order Routing.
DRAM  Dynamic Random-Access Memory.
EDP  Energy-Delay-Product.
FIFO  First-In, First-Out.
GCN  Graphics Core Next.
GDDR  Graphics Double Data Rate.
GPUs  Graphics Processing Units.
HBM  High-Bandwidth Memory Technology.
HPC  High-Performance Computing.
HSA  Heterogeneous System Architecture.
ICs  Integrated Chips.
ITRS  International Technology Roadmap for Semiconductors.
LDS  Local Data Share.
LLC  Last Level Cache.
MPI  Message Passing Interface.
MWSR  Multiple-Write, Single-Read.

NoC  Network-on-Chip.

OpenCL  Open Computing Language.

OS  Operating System.

PCIe  Peripheral Component Interconnect Express.

QoS  Quality of Service.

SIMD  Single Instruction, Multiple Data.

SIMT  Single Instruction, Multiple Thread.

SKE  Scalable Kernel Execution.

SoC  System on Chip.

SPMD  Single Program, Multiple Data.

SMD  Stacked Memory Directory.

SWMR  Single-Write, Multiple-Read.

SVM  Shared Virtual Memory.

TSVs  Through-Silicon Vias.

UM  Unified Memory.

UMH  Unified Memory Hierarchy.

UVA  Unified Virtual Addressing.
Acknowledgments

"Life is not a one-man job” One of the times that I had this reoccurring realization, I wrote a story about a magician, in the wide west. I don’t know why but, at the time, the story felt needed. A story about the goodness of people. About a magician that could have had the world, but grew old silently and generously in a remote city in the middle of nowhere, just keeping his people happy. Made them excited, made them joyous, made them belong.

Looking back today, I see many of those magicians in my life, making me “me”, making me joyous, making me belong. It should go without saying, but I suppose it is much nicer if it is said. Maman Sonia, and Baba Farokh are my basic blocks. The reasons for my good life. They are my fortune. As I grow old, I will become them more and more, and I am content with that.

Another magician is Dave Kaeli who is far more than what I could have wished for an advisor, and is and will always be more than one. He is a good man. A known good man. He was to me and my academic career, an eye to a blind. He reshaped my academic persona patiently, optimistically, and gleefully. He has done the same for many others. One calls him the magician, another Gandalf. That is mark of a good man.

Life is much harder without a few good friends. Growing up my brothers Amir and Kamyar were my best friends, but as geographical separation came into play, I was lucky enough to have new friends, carrying that torch, and carrying me once in a while. This is a nod to them. To Rafa for making me a better man every day, to Dana and Jenny who I am the best version of me when I am around them, to Asieh for her contagious excitement for a better tomorrow and her bitter grumpiness toward the imperfect past, to Jaume for his emotional availability, to Sherman who is the best dad-to-be, to Jimmy who is the best Irish man to grab a drink with, to Lika for her goodness and innocence, to Nick and Catherine for being my social teachers (and patient ones at that), and to Hamid who listens.
Abstract of the Dissertation

Improving the Global Memory Efficiency in GPU-Based Systems

by

Amir Kavyan Ziabari

Doctor of Philosophy in Electrical and Computer Engineering
Northeastern University, December 2016
Dr. David Kaeli, Adviser

Graphics Processing Units (GPUs) have been used in a wide range of high performance computing domains. Unfortunately, computing with GPU devices presents its own challenges, including inefficiencies in the global memory system. With today’s growing demand for Big Data processing, the need to leverage larger-scale GPUs or multiple GPUs becomes the natural next step. Big Data applications magnify the current limitations of global memory on GPU-based systems. A major source of this global memory inefficiency is due to bottlenecks in the on-chip network associated with this memory.

In this dissertation, we describe how to optimize the performance and power efficiency of an on-chip network used on a GPU. We explore the GPU-based Network-on-Chip (NoC) design space, develop execution-driven simulation models, and analyze a range of parallelized applications. We evaluate a number of conventional network topologies, and their impact on performance of a GPU system. We use detailed simulation to characterize memory access patterns present in the GPU applications, and explore electrical on-chip networks that best match the needs of these applications. We incorporate asymmetry into the NoC design as a solution to reduce the power consumption of a network, while providing comparable performance to the best conventional topology. Our solution reduces the Energy-Delay Product (EDP) by as much as 88%.

In order to improve the performance of current and future GPUs, we explore the use of silicon-photonic link technology when constructing the NoC. This emerging, low-latency, high-bandwidth technology has been incorporated in chip multiprocessors (CMPs). By introducing a hybrid silicon-photonic NoC in the GPU memory system, we are able to improve performance of memory-intensive applications by $3.43 \times$, as compared with the best alternative electrical NoC.

Finally, we conduct a thorough analysis of global memory management schemes for multi-GPU systems. We identify limitations of the global memory present in previously proposed
memory management schemes, and propose an alternative, Unified Memory Hierarchy (UMH). Our goal with UMH is to reduce the communication between multiple GPU devices. Our solution supports coherency and cooperative execution between a CPU and multiple GPUs through a single, shared, memory hierarchy. From the GPU’s perspective, the host/CPU memory now serves the role of main memory for both the CPU and the GPU. Adopting this design, a GPU accesses data which resides in CPU memory only if it does not find the data in its own high-bandwidth memory. The UMH design includes the addition of a memory directory for inter-device memory management and coherency. The proposed coherency protocol allows coherent access between a CPU and multiple GPU devices, while relaxing coherency constraints on the GPU when coherency does not need to be enforced. Adding a proper memory coherency protocol to our UMH design reduces the overhead of synchronization between devices by as much as 13×. Additionally, UMH enhances the performance of multi-GPU systems by 1.92× and 5.38× (on average) over alternative memcpy and zero-copy approaches, respectively, for a system with 4 discrete GPU devices.
Chapter 1

Introduction

A little over a decade ago, Graphics Processing Units (GPUs) were fixed-function processors built around a pipeline that was dedicated to rendering 3-D graphics. In the past decade, as the potential for GPUs to provide massive data-level parallelism became apparent, the software community developed programming environments to leverage these massively-parallel architectures. Vendors facilitated this move toward massive parallelism by incorporating programmable graphic pipelines. Parallel programming frameworks such as Compute Unified Device Architecture (CUDA) [7] and Open Computing Language (OpenCL) [8] were introduced, and tools were developed to explore GPU architectures. NVIDIA and AMD, two of the largest graphics vendors, tailored their GPU designs for general purpose high-performance computing, to provide higher compute throughput and memory bandwidth [1, 9, 10].

In contrast to CPUs, graphics processing units (GPUs) exploit fine-grained parallelism. GPU workloads are designed to be processed in a Single Instruction, Multiple Thread (SIMT) fashion. SIMT is a term coined by NVIDIA to refer to the mapping of the Single Program, Multiple Data (SPMD) programming model to Single Instruction, Multiple Data (SIMD) hardware. Application developers are responsible for optimizing their applications in such a way to best leverage the SIMT nature of the GPU hardware. CUDA and OpenCL provide language and runtime support that allows the developer to rewrite the data parallel portions of their application, using C/C++-like syntax, to take advantage of the GPU device. In the past decade, countless papers have been published on mapping and optimizing general-purpose algorithms on the GPU [11, 12, 13]. It has also encouraged researchers to use GPUs to explore a range of challenging problems in various domains (e.g., chemical-physics [14], genetics [15], deep learning [16]). An application that is written for execution on GPU hardware is called a kernel. Kernels exploit large-scale data-level parallelism.
CHAPTER 1. INTRODUCTION

Figure 1.1: The addition of two vectors using GPUs. Each GPU threads calculates one element of the output vector at a time.

Each instance of kernel application (known as *work-item*) is assigned to a single hardware thread on the GPU. A GPU device consists of many hardware threads that can execute in parallel, allowing many instances of an application to be executed together at the same time. GPU kernels typically consume an initial input data. The application loads this data from the dedicated memory of the GPU device, known as the global memory. This means every hardware thread is assigned an instance of the kernel. Every instance will need to access global memory to retrieve the data required by that instance to complete its execution. Additionally, applications are usually required to store their computed data to the global memory as well. So every hardware threads need to access global memory again in order to store the computed data in global memory.

Figure 1.1 shows how a typical GPU application performs a vector addition of two vectors of the same size, $A$ and $B$, and stores the results in a third vector, $C$. For this application, each hardware thread adds the elements with the same index from $A$ and $B$, and stores the resulting element in the same index in the vector $C$. This requires each hardware thread to perform two load operations and one store operation to the global memory. However as the number of elements in the vectors exceeds the number of available hardware threads, each hardware thread has to perform the addition for multiple indices. This means more loads and stores should be performed by the thread. Therefore, the speed at which the global memory can address these growing number of load and store requests has a direct impact on the time it takes an application to finish its execution on the GPU. If the global memory is incapable of providing the data as fast as the GPU hardware is consuming the data (e.g., performing the addition) it can potentially become a bottleneck for application execution.

In this chapter we introduce contemporary GPUs, future scalable GPUs (which are likely to have more computing power), and multi-GPU systems (which leverage multiple discrete GPU
systems for application execution), and will discuss how global memory can quickly become a bottleneck for application execution on these systems.

1.1 Background and Motivation

1.1.1 Existing GPU Architectures

Contemporary GPUs are composed of tens of Compute Units (CUs); for example, 32 CUs in case of AMD Radeon HD 7970, or 15 streaming multiprocessors (the NVIDIA’s term for compute units) in case of NVIDIA Kepler GK110. CUs are composed of separate pipelines designed to execute a massive number of concurrent threads (e.g. 2048 threads in AMD Radeon HD 7970). When using a large number of GPU threads per CU, application developers can overlap both computation and memory operations, to minimize the impact of potential global memory latencies on the computation.

To decrease the impact of global memory on application performance, contemporary GPUs come equipped with a cache hierarchy that comprises two levels of caching (e.g., 40 L1 caches and 6 L2 caches in the AMD Radeon HD 7970 GPU). These different cache levels (L1s and L2s) in the GPU are interconnected through a network.

Figure 1.2 shows a high level view of the contemporary GPU device we model in this dissertation. A CPU host device is in charge of starting the execution of a GPU application by first setting up the required data of that application, and offloading the parallel kernel(s) to the GPU device. The program that runs on the CPU and is in charge of setting up the GPU device for execution is known as host program. The setup step of the host program typically involves copying the entire data range that is required by all the GPU threads to the main memory of the GPU. During kernel execution, the threads that are scheduled on different compute units perform load and store operations to global memory through the cache hierarchy. These load and store requests and responses have to traverse an Network-On-Chip within the cache hierarchy of the global memory.

1.1.1.1 Network-on-Chip for GPU Architecture

Interconnection networks is defined as a system that enables fast and reliable data communication between digital components of a system. Any system that contains more than one component requires some sort of interconnect that allows these components to interact with each
CHAPTER 1. INTRODUCTION

Figure 1.2: High level overview of the contemporary GPU accelerator. This design is based on AMD Southern Islands 7970 HD GPU.

other. The simplest form of a interconnect is a wire. An interconnection networks makes it possible for different components to share resources, and communication in an organized fashion.

The interconnect can be of any scale. It can connect components within a computer device, multiple computers together, or cluster of computers to each other. The term node refers to the module that leverages the interconnect for communication with other modules (or nodes). Each node in the interconnect requires an interface to communicate with other nodes through the interface.

Parallel computers with multiple computing components strongly leverage from interconnection networks, and can also be affected by the design of these interconnects. We refer to the interconnection network that is responsible for connecting different L1 cache units (and their associated compute units) to the L2 cache units (and their associated memory controllers), as the Network-on-Chip (NoC) of the GPU device. The topology of the NoC is the graph that connects different nodes (L1s and L2s) to each other, using different basic blocks (i.e., routers, links, and buses). The messages initiated by the L1 caches toward the L2 caches such as load and store requests (the L1-to-L2 traffic), and generated by the L2 caches toward the L1 caches such as load and store responses (L2-to-L1 traffic) traverse this network. The design of the network should consider
CHAPTER 1. INTRODUCTION

![Graph showing execution time and bandwidth comparison between ideal and electrical NoCs for various workloads on a 32-CU GPU.]

Figure 1.3: Potential improvements for various workloads running on a 32-CU GPU, with an ideal NoC against an electrical NoC.

the types of traffic that is placed on that network (i.e., the number of messages that traverse the network at the same time, the path in the graph that the messages traverse, etc.). Network Latency is the average time it takes for a message to traverse the network from its source to its destination, and can be impacted by different factors such as the topology of the NoC, and the amount of the network traffic.

1.1.1.2 Challenges

There is limited communication between different threads that are executed on different compute units of a GPU. Since GPU applications generally exhibit little data reuse, accesses to global memory usually result in a request that traverses the network between the compute units and their associated L1 caches, to the memory controllers and their associated L2 caches [17]. The requests from the L1 to L2 caches can generate a contention in this network. This is mainly due to a large number of threads being executed in parallel. Additionally, L2 caches have to respond to these parallel requests as fast as possible in order to allow the compute units to continue the execution of the threads. Abts et. al. [18] define this traffic pattern as “many-to-few-to-many”. That is, many L1 caches send requests to few L2 caches, and the few L2 caches respond to those many L1 caches.

Since the network between L1 to L2 caches carry data in both directions (L1 to L2 for read requests and stores, and L2-to-L1 for data transfers on loads), they have to be carefully designed to avoid long latencies in the network. To evaluate the potential benefits of employing a low-latency Network-on-Chip (NoC) for GPUs, we examine a GPU system with 32 compute units (similar to AMD Southern Islands 7970 – see a more detailed description in Chapter 2).
CHAPTER 1. INTRODUCTION

In our evaluation we consider two separate topologies for the network. The first topology is the *ideal* topology. The ideal topology is the best possible network, since it has a fixed latency between the source and the destination of the message. It makes the network to be a deterministic part of the architecture design. The second considered topology is a crossbar topology. Crossbar is the most well-known interconnect topology since it provides an all-to-all connection between all the nodes (L1 and L2 caches) in the network.

In Figure 1.3 We compare a system with an electrical crossbar (See Chapter 2 for more detail) network against a system deploying an ideal network, with a fixed 3-cycle latency between source and destination nodes. This means every message in the network is received by the destination nodes 3 cycles after it is sent. To evaluate the GPU system performance when using these NoC we utilize applications from the AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) [19].

The most important metric to consider for compute systems is the time it takes for the applications to execute on that system, or application *execution time*. The execution time has a reciprocal relation to the *performance* of the system. If an architectural enhancement reduces the execution time of applications on a system, this means it has improved the system’s performance. In our evaluation, we observe that using an ideal network leads to up to 91% reduction in the execution time of the selected applications, in comparison to using a crossbar network.

The network *bandwidth* is the amount of data that can traverse the network at every unit of time. The bandwidth is usually represented by the number of bytes that completely traverse the network at every second (Bytes per Second), and is another important metric that should be considered in designing the NoC for the computing systems. In the comparison presented in Figure 1.3 we observe that the ideal network provides much higher bandwidth than the crossbar network. In the crossbar network, the nodes compete with each other for the resources (such as competing to transfer data on the links/buses that act as edges of the network graph). Therefore, only a limited number of nodes can send messages on the network at every network cycle (the unit of time on the network). On the other hand, with the ideal network every node can send messages at any desired network cycle (no competition). So the provided bandwidth of the ideal network is up to $10.3 \times$ higher than the crossbar network.

These results clearly motivate the need to explore different NoC topologies, possible novel link technologies, and architectural enhancements to design networks that can improve performance in current and future GPUs. Designing an ideal network is very expensive, since the network should provide dedicated non-competing paths between all the nodes within the network. It is also impor-
CHAPTER 1. INTRODUCTION

tant to note that simply increasing the offered bandwidth of the network is not a sufficient approach to address possible network bottleneck. Increasing the bandwidth of the network is possible by widening the connections between nodes, or providing more direct connections between nodes. Either way will correspondingly increase the power consumption of the network (due to an increase in the number of channels, or increase in the number of required wires to widen the channels). Instead we should design networks that are tailored for the needs of the applications executing on the system (the GPU device), to improve the performance of that device.

1.1.2 Future Scalable GPU Architectures

A major paradigm shift in CPU design occurred in the 2000s, as the power wall stalled the historic increases in CPU clock frequencies. However, as transistor count still continued to increase, computer architects began creating CPUs with multiple cores, each capable of executing its own instruction stream. Still, driven by the need for high single-threaded performance, the number of cores remained relatively small, and thus parallelism in software remained largely coarse-grained. Although SIMD processing has had a long tradition in CPUs, SIMD highly depends on being able to exploit parallel data types in otherwise sequential programs, relying on the compiler to automatically utilize SIMD feature for acceleration [20].

In contrast to CPUs, graphics processing units (GPUs) have had a history of exploiting very fine-grained parallelism. The Arithmetic Logic Units (ALUs) in GPUs are based almost entirely on SIMD hardware (GPUs include scalar execution units that are used to execute only a set of special operations – See Chapter 2). GPUs show a remarkable aptitude for accelerating data-parallel general purpose programs. Inherently parallelism that exists in GPU applications result in highly scalable applications. This means that, unlike CPU architectures, if we scale the number of hardware threads, we can achieve speedup relative to the amount of scaling. For example, the performance of an application running on a GPU system with 128 CUs should approximately be 4 times the performance of the application running on a GPU system with 32 CUs (the ratio of the number of compute units of the two systems).

Additionally, with today’s growing demand for “Big Data” processing, the need to leverage from more GPU processing power becomes the natural next step. A large array of applications in the field of Big Data processing map perfectly to the GPU programming model (e.g., Genomic Sequence mapping and analysis [21], large-scale data mining [22], deep learning using neural networks [23]) By increasing the number of compute units in the GPU, we can potentially increase the
CHAPTER 1. INTRODUCTION

processing speed of these applications and address this growing demand.

1.1.2.1 Challenges

The attempt to increase the speed of GPUs merely by increasing the number of compute units on individual chips does not necessarily improve the performance of many applications, specially those that process a large number of data elements received from the global memory (applications with large memory footprint), and applications that perform many memory operations during the processing of a single data element (i.e., memory-intensive applications).

The global memory (memory system and the interconnect) will eventually become a bottleneck on the most scalable GPU, no matter how efficient global memory is designed. The problem is the lack of memory bandwidth, as well as contention between compute units over the network. Since the amount of caching in the L1 cache units (closest to the compute units) is limited, we require either faster memory or more highly interleaved memory banks to address the large amount of requests from compute units. However, both memory transfer bandwidth and memory area are limited by the power wall [24]. The memory transfer technology simply cannot keep up with the high bandwidth demands made by the processing cores.

For evaluation of the future architectures, we consider a forward-looking GPU design with 128 compute units, and a global memory hierarchy that matches the existing GPU designs in proportion. The main challenge for evaluating the forward-looking GPUs becomes designing the GPU architecture and its global memory, and employing a low-latency NoC topologies that can best service the global memory of such a system under large workloads.

1.1.3 MultiGPU Systems

As the size of the data sets and number of processing steps present in these applications continue to increase, we will quickly outgrow the compute resources provided by contemporary GPU devices. Uniting multiple GPU devices into a single system is an attractive alternative to designing a scalable GPU device for addressing these applications’ processing demands. In comparison to a forward-looking GPU architecture with more number of CUs, the multi-GPU system provides the possibility to leverage current existing GPUs, with small modifications, to process applications with large datasets. Previous work [25] 26 27] has shown that we can significantly improve application throughput when combining multiple GPUs, and therefore, there has been an
CHAPTER 1. INTRODUCTION

Figure 1.4: High-level with of a multi-GPU system design, leveraging a conventional PCIe off-chip interconnect.

increased focus on how best to utilize and program multiple GPUs to address this trend in a range of applications [28, 29, 30, 31].

Figure 1.4 demonstrate a high-level view of a multi-GPU system. The CPU still acts as a host to GPU devices; the CPU is still required to offload the workloads to these GPUs. However utilizing these systems has its own challenges. MultiGPU system require a mechanism to distribute the application workload on multiple devices. Current programming languages (e.g., OpenCL or CUDA) rely on the CPU and their runtime libraries to place the workload on a device, in a single-GPU system. But neither of these popular programming languages provide a robust automated infrastructure that allow the distribution of the workload to multiple GPUs.

A simple, yet reliable, scheduling algorithms to distribute the workload to multiple GPU devices is the Scalable Kernel Execution (SKE), presented by Kim et. al. [28]. SKE is a user-level dynamic library that abstracts all of the GPUs in a system, treating them as a single virtual GPU, to simplify programmability of multi-GPU systems. SKE leverages the same kernel code for a single GPU but takes advantage of the multi-GPU system by distributing the groups of threads (known as thread blocks or workgroups) of a kernel to different GPUs. With SKE, a single kernel can be seamlessly executed across all the GPUs [28]. This scheduling method perceives multiple discrete GPU devices as a single virtual GPU device and assigns the workloads to the GPU devices via a fair distribution. For our evaluations we leverage from SKE to allow multiple GPUs to concurrently process equal portions of the application data. SKE simplifies porting single-GPU workloads to multi-GPU systems, but it requires thorough global memory management, and if not done properly, it will significantly decrease the performance of the system. Next we describe the challenges in the global memory management of such multi-GPU systems.
CHAPTER 1. INTRODUCTION

1.1.3.1 Challenge 1: Cross-device Communication and Data Sharing.

One of the challenges in multi-GPU programming is sharing data across GPUs. Different GPUs may need to read the same data in order to execute their portion of a kernel. Through the support of GPU vendors (such as NVIDIA and AMD), and new features in GPU programming languages (CUDA and OpenCL), CPU and GPU devices are able to share data through shared memory. In CUDA terminology, this shared memory is referred to as Unified Virtual Addressing (UVA) \[32\], while OpenCL terminology uses Shared Virtual Memory (SVM) \[33\] to refer to this architectural enhancement. With shared memory the user can maintain all the data that is required for the application in a central place (such as the memory of the host device). Using a central memory to keep all the data in the memory of the CPU, increase the contention to access that memory, while underutilizing the memory of all the other GPU devices in the system.

The programmer can also distribute the address space of the application to multiple devices and allow all devices to access each other’s memory to retrieve the data they require. This requires manual management of the memory between the CPU and multiple GPU devices through the host program, which can be a hard task, inefficient, and prone to errors. Also maintaining the data across the memories of different GPU devices introduces a large off-chip interconnect latency to every access that is made from one GPU to the memory of another GPU for the required data.

Neither of these approaches are satisfactory for executing an application on multi-GPU systems, therefore the investigation of possible alternative approaches are required in order to utilize the existing resources the best way possible and improve the performance of the executing applications on such systems.

1.1.3.2 Challenge 2: Coherency between CPU and Multiple discrete GPUs.

Coherency between CPU and GPU device is another important issue that has been the center of attention in the recent years \[34\][35][36\]. Usually the GPU’s computed data is not available to the CPU device until the execution of the kernel is completely finished on the GPU. This means, even if part of the data has been already processed by the GPU (e.g., vector addition is performed on the first half of the elements of the two vectors), the CPU still cannot have access to this data, unless the content of the global memory of a GPU (or multiple GPU devices) are transferred to the CPU memory through synchronization. This results in CPU device to remain idle while the GPU device is performing kernel computations.
CHAPTER 1. INTRODUCTION

To resolve this issue, both NVIDIA and AMD have been actively involved in the development of software frameworks that support coherency between the CPU and GPU devices. This has become a primary requirement (even ahead of performance, if required) [32]. In CUDA 6.0, NVIDIA introduced Unified Memory (UM) to enhance Unified Virtual Addressing, using the CUDA runtime to transfer data between the host and devices in a user-transparent fashion.

However, it has been shown that Unified Memory can degrade the performance of applications when both the CPU and the GPU share data in the execution of the application. These performance issues are reported by Li et al. [37] and Pai [38] for the NVIDIA CUDA 6 runtime, and the same performance issues are present in CUDA 7. Of the multiple runtime inefficiencies reported, the following are the most serious:

First, Unified Memory does not check whether the GPU actually needs all the allocated data that is being transferred from the CPU/host memory, resulting in a number of unneeded memory transfers. Second, the current software-based UM mechanism presently does not support multiple GPUs. Data is allocated on only one device at the time of allocation, even if multiple CPU or GPU devices are available [39] [32], leading to slower memory operations and performance degradation. Third, software UM always assumes that the GPU data is modified [38]. This means that on every synchronization between the CPU and the GPU, all the GPU data needs to be copied from the GPU memory to the CPU memory, even if the same data is available on the CPU side. This also leads to redundant data transfers. It is evident that a more reliable mechanism is required to allow the CPU and the GPU devices to manage coherency without involving an excessive number of redundant copies between the two devices.

1.1.3.3 Challenge 3: Synchronization Cost.

Synchronization is a process that occurs at the end of the GPU kernel execution. Synchronization on a GPU is used to help insure computed data is coherent on the CPU device, but only after the application finishes its execution on the GPU devices. Synchronization overhead can become very costly in terms of performance. For example, if the GPU is executing an application that produces a large array as output, the GPU has to store the computed data first into its global memory (typically L2 caches). Then this content should be moved to the memory of the CPU device. The off-chip interconnection can play a very important role in synchronization, especially as the copied data grows in size. The host program has to wait until the synchronization of the entire array is complete.
CHAPTER 1. INTRODUCTION

Synchronization is required at the end of execution of the GPU applications, in systems that do not provide coherency between the CPU and the GPU (or a mechanism for dynamic synchronization of individual elements of the output array during the execution). Therefore, we require a mechanism to provide the coherency between devices (or dynamic synchronization during the execution) in such a way that the data that is required by the CPU instruction at this instance (i.e., just a few elements of the array) become coherent/synchronized between the devices. Meanwhile, the remaining data can be maintained non-coherently on the global memory of the GPU device. With this approach, the cost of synchronization at the end of application execution can be avoided (CPU can start its process as soon as few elements are coherent) and the only additional cost is the hardware (or the alternative technique) that makes only the required data coherent/synchronized between the CPU and GPU devices.

1.2 Contributions

The main objective of this dissertation is to tackle the challenges highlighted in the previous section. Here, we outline the work done for this thesis, which includes the following contributions.

With regard to existing GPU architectures:

• we investigate the design of an energy-efficient NoC that is tailored for GPUs and their underlying programming model. To explore the design space of NoC designs for GPUs, we target a contemporary AMD Southern Islands GPU architecture [1] and choose applications with a range of workload intensities from the AMD SDK benchmark suite [19].

• We compare various electrical NoCs, using performance and power metrics, to identify the best topologies for our target GPU system. We present an analysis of the traffic patterns resulting from executing multiple data-parallel applications on the evaluated NoCs.

• Leveraging our analysis considering both performance and power metrics, we propose an asymmetric NoC design that can achieve comparable performance at reduced power consumption.

For existing and future scalable GPU architectures:

• We leverage emerging silicon-photonic technology, a technology that has been proposed as a potential replacement for electrical link technology for many-core system, and consider this
technology for the NoC for future GPU architectures. We explore the design of Multiple-Write, Single-Read (MWSR), and Single-Write, Multiple-Read (SWMR) electrical and photonic crossbar NoCs for communication between L1 and L2 of a GPU with 32 CUs.

- We propose the design of a GPU-specific hybrid NoC with reduced channel count based on the patterns observed in the communication between L1 and L2 cache units in the GPU systems. We compare the electrical and silicon-photonic implementations of this hybrid NoC architecture in a GPU with 32 CUs running standard GPU benchmarks.

- We carefully design a realistic layout of a forward-looking scalable GPU with 128 compute units that follows the same design consideration, used in the AMD’s Southern Islands GPU architectures. We evaluate the scalability of our silicon-photonic hybrid NoC on this forward-looking GPU system, comparing our proposed hybrid photonic crossbar against a competitive electrical 2D-mesh NoC.

Our work targeting the efficiency of the global memory of multi-GPU systems make the following contributions:

- We identify the limitations of the existing memory management schemes, particularly the zero-copy and memcpy approaches, considering systems with multiple GPUs and Scalable Kernel Execution (SKE) [28].

- We describe a novel hardware-based Unified Memory Hierarchy (UMH) for systems that utilize any number of GPU devices. We show how our UMH scheme keeps the memory management task transparent to the programmer, while reducing the number of redundant data transfers as compared to software UM.

- We explore the benefits of our new memory hierarchy while leveraging the NMOESI coherence protocol to support coherency and fast synchronization between a CPU and multiple discrete GPU devices.

1.3 Thesis Outline

The rest of this dissertation is structured as follows. Chapter 2 presents the necessary background for the thesis which includes an introduction to the GPU architecture, the GPU programming model, and the interconnection networks. In this chapter we also introduce silicon-photonic
technology as an emerging technologies for designing links in the interconnection networks. We also introduce our evaluation methodology in this chapter. We tackle the global memory challenges for existing and forward-looking GPU architectures in Chapters 3 and 4 respectively. Chapter 5 present our vision to undertake the challenges in the multi-GPU system and the design of the Unified Memory Hierarchy, equipped with NMOESI protocol. Finally, in Chapter 6 we present our conclusions and outline our future works.
Chapter 2

Background

In this chapter we provide the necessary background information on the OpenCL GPU programming framework. We provide an overview of the organization of a GPU and its global memory design, using the AMD Southern Island GPU family as our main example. The description of different memory management methods between a CPU and a GPU is also described in this chapter. We briefly describe several fundamental concepts related to interconnection networks, and describe the design consideration for current Network-on-Chip in GPU devices. We introduce the silicon-photonic links, and High-Bandwidth Memory (HBM) as two emerging technologies for designing the memory interconnection for future GPUs, as well as multi-GPU systems. Lastly, we introduce Multi2sim [40] as our simulation framework, and describe the target GPU for our evaluations.

2.1 OpenCL Runtime and Programming Model

OpenCL [8] is a popular standard for programming heterogeneous systems. OpenCL provides a host program the ability to offload computationally-intensive and data-parallel applications to an accelerator device such as a GPU. The OpenCL runtime is a collection of API functions that allow a CPU program to create and manipulate an application, targeted for GPU devices.

2.1.1 Execution Model on GPU architecture

The host terminology in OpenCL refers to the program that runs on the CPU for setting up the GPU device (or other accelerators). The processors that are capable of executing the computational kernels are referred to as devices. When running an OpenCL program, the abstract container
CHAPTER 2. BACKGROUND

Figure 2.1: Software entities defined in the OpenCL programming model. An ND-Range is formed of workgroups, which are, in turn, sets of work-items executing the same OpenCL C kernel code.

for managing objects and devices is referred to as a context. Objects created within a context are only valid within that context. Within a context, buffers are created to hold indexable data structures (similar to C arrays) used during computation, and kernels contain the code of the computation to be carried out (similar to function calls) on the device. The programmer submits commands to a command queue, requesting some type of work to be performed (e.g., transfer data or execute a kernel) [8].

Command queues target individual devices. This means we require multiple command queues to submit work to multiple devices. To execute a kernel, we need to provide to specify an N-dimensional range (ND-range) of work-items (intuitively software threads, that are the instances of the kernel) to be executed.

To increase performance and provide a level of collaborative execution among threads, the programmer can specify the selected work-items to be executed together within a group, known as workgroup. A workgroup provides its work-items the ability to share data with each other and synchronize. In order to have synchronization and data sharing, the work-items of the same workgroup are executed on the same processing unit (i.e., CPU core, GPU compute units, etc.), and out of the same L1 cache. However, workgroups themselves, are independent from each other and can be executed on different processing units. ND-ranges, workgroups and work-items correspond directly to CUDA grids, blocks, and threads, respectively. The relationship between work-items, workgroups, and the ND-Range is shown in Figure 2.1 [8].

Currently, if users require to use multiple GPU devices to execute a kernel, they must
CHAPTER 2. BACKGROUND

Figure 2.2: A multi-GPU system with an original ND-range is partitioned into two separate ND-ranges for execution on two GPU devices.

treat the problem as multiple unique kernel launches and submit separate ND-ranges for each device (Figure 2.2). When dividing the work in this manner, the number of work-items to be instantiated on each GPU must be determined when the kernels are launched, and offsets for the ND-ranges should be calculated. The command to execute the kernel is then submitted to each device. The obvious drawback of this execution model is the challenge of workload balance between devices, as well as dynamic memory management.

2.1.2 OpenCL Memory and Consistency Model

The OpenCL execution model specifies four distinct memory regions on the GPU for executing a kernel. OpenCL has a consistency model associated with each memory region. The memory regions in the OpenCL memory model are Global, Constant, Local, and Private.

The Global Memory, the main focus for this dissertation, is a region that permits read and write accesses to all the work-items in all workgroups of an ND-range. Work-items are allowed to read from or write to any elements of a memory object in this memory region. Global Memory supports weak ordering for work-items within the same workgroup. This means the global memory should guarantee that work-items within the same workgroups become consistent at synchronization points. Synchronization points are provided by barriers and memory fences, and ensures correct ordering of memory accesses between work-items within a single workgroup. The synchronization between work-items in different workgroups is not possible, and not required. If global synchro-
CHAPTER 2. BACKGROUND

Organization is needed for correctness, multiple ND-ranges should be used.

**Constant Memory** is another region of global memory that remains constant during kernel execution, and is used for read-only operations by work-items. Since Constant Memory is a read-only memory region, a consistency model is not required for this memory.

**Local Memory** is a dedicated memory that allows the work-items within the same group to share data with each other. It may be implemented as dedicated memory module, or be mapped onto sections of Global Memory. As with Global Memory, Local Memory supports weak ordering for work-items within the same workgroup. Weak Ordering [41] is the most relaxed consistency model that can support OpenCLs memory model, and thus is capable of providing the highest performance.

Finally, **Private Memory** is the dedicated memory for each work-item (i.e., registers) that are inaccessible to any other work-items. Private memory (memory visible only to a single work-item) maintains load/store consistency.

Since OpenCL 2.0 [33], support for atomic operations has been added to this programming framework. Atomic operations are extensions that can be used to provide ordering in terms of the execution of work-items within different workgroups. Commercial GPU architectures have implemented hardware that is capable of providing acquire and release semantics (supporting Release Consistency – which is a stronger model than Weak Ordering). A read-acquire signals the memory system that the effects of this operation must be visible to all other processing units (cores or CUs), before completing any of the operations that might follow. (i.e., operations may not be reordered). Similarly, a store-release says that all previous memory operations must be visible to all processing units before the effects of this operation is globally visible (i.e., previous operations may not be reordered relative to this memory access).

2.2 GPU Organization and Design

In this section, we describe the general characteristics of discrete GPUs. We will use AMD’s Graphics Core Next (GCN) architecture, and terminology for the reminder of this dissertation.

As can be seen in Figure 1.2 in Chapter 1 three main components shape the high-level design of the GPU architecture. The first major component is the Command Processor, which is in charge of receiving commands from the host device in the form of command queues, and initiates the execution of the kernel based on the commands. The second major component is an array of
CHAPTER 2. BACKGROUND

*Compute Units* which are in charge of performing the parallel computation. The third component is the *Global Memory Hierarchy* that is designed for caching the required data by the work-items, and to provide some order of consistency (i.e., Relaxed, Release) and limited communication.

In this section we delve into the details of each major component, using the design of the AMD GCN GPU architecture as our main example. The GCN is the GPU architecture utilized in design of the AMD’s Southern Islands GPU devices [1].

2.2.1 Command Processor and Dispatcher

The GCN command processor is the unit that is responsible for receiving high-level API commands from the graphics driver, and mapping these commands to different processing elements of the Compute Units within the GPU. The command processor for the general purpose computing in the GCN have been improved to allow for multi-tasking (scheduling commands of multiple contexts to the same GPU), virtual memory (executing kernels of multiple ND-ranges on the compute units), priority-based execution (prioritizing a task to another based on the requirement of the applications), and allocating the resources of compute units to multiple workgroups [1].

The host program performs multiple OpenCL runtime API function calls to set up the GPU device. These API calls are translated to the hardware commands using the graphics driver. The commands are submitted to the command processor. The OpenCL workgroups are divided up by the *ultra-thread dispatcher* into smaller hardware threads, called *wavefronts*, which contain up to 64 work-items each. This dispatching is performed transparently to the user [42].

2.2.2 Compute Unit Design

Compute units are the basic computational blocks of an AMD’s GPU architecture. The main computation engines in a CU are the Single Instruction, Multiple Data (SIMD) units. CUs includes 4 separate SIMD units for vector processing. Each SIMD unit simultaneously executes a single operation across 16 work-items, but each can be working on a separate wavefront. This means, a wavefront is executed on a SIMD pipeline in 4 partitions, taking 4 cycles to complete. This allows many wavefronts to be processed in parallel, on different SIMD pipelines, while allowing 4 wavefronts to be finished in 4 cycles.

Figure 2.3 presents a detailed view of a compute unit. Compute units have 5 separate pipelines for different types of instructions that exist in a workgroup. Work-groups are split into wavefronts, for which the compute unit executes one instruction at a time. The compute unit front-
CHAPTER 2. BACKGROUND

Figure 2.3: The design of a AMD’s GCN compute unit [1]. The compute unit contains five processing pipelines. The vector memory pipeline, has 4 SIMD units, capable of processing one instruction of 64 work-items at each cycle. The colored pipelines, scalar and vector memory (against the grayscale units), are involved in the processes that lead to a global memory access.

The front-end fetches instructions from instruction memory, using an instruction fetch buffer. The front-end is formed of a set of wavefront pools, and an instruction arbitration/issue stage. The number of wavefront pools match exactly the number of SIMD units. Its goal is to fetch instructions from instruction memory for different wavefronts, and to send these instructions to the decoder of the corresponding execution unit.

The execution units present in a compute unit include a set of SIMD units, the scalar unit, the vector memory unit, the branch unit, and the Local Data Share (LDS) unit. Arithmetic vector instructions are sent to the SIMD unit matching the wavefront pool where the instruction was fetched. The scalar unit is responsible for executing scalar arithmetic-logic and scalar memory instructions. The branch unit is responsible for executing certain control flow instructions. The LDS unit is responsible for handling all local memory instructions, and uses local memory to service its instructions. The scalar and vector memory units can access global memory, shared by all compute units. The Vector Memory Unit is responsible for handling all vector global memory operations.
CHAPTER 2. BACKGROUND

2.2.3 Global Memory

The vector memory instructions of a wavefront can generate up to 16 address requests (number of work-items in a wavefront). These requests are sent to a unit responsible for address calculation and coalescing (known as Address Generation Unit). The address generation unit receives these addresses on each clock cycle and coalesces the requests together. Coalescing is possible if multiple address requests are made to the data that resides in a single cache-line (usually 64B).

Each CU is equipped with a L1 vector data cache unit, known as a vector cache, which provides the data for vector-memory operations. The vector cache is 16KB and 4-way set associative, with a 64B line size. The vector cache has a write-through, write-allocate, design. If the requesting cache line exists in this L1 cache (due to a previous read access), the data is returned to the compute unit. Otherwise the request traverses the global memory hierarchy to the appropriate L2 cache unit.

The L2 cache in the Southern Islands architecture is physically partitioned into banks that are coupled with separate memory controllers. Similar to the L2 cache units, memory controllers are partitioned into different memory address spaces. Each L2 cache unit is associated with only one memory controller.

Scalar operations (i.e., computations shared by all threads running as one SIMD unit) use a dedicated set of L1 scalar data cache units (sL1s), known as scalar caches, shared by 4 CUs. The data in the scalar cache unit is read-only, and acts as the constant memory region, defined in the OpenCL programming framework.

All the scalar and vector cache units are connected to the L2 cache units through a Network-on-Chip (NoC). The Network-on-Chip is an interconnection network designed to carry the messages between different cache units of the global memory of the GPU. The type of messages transferred on the NoC include control and data messages. A control message is 8 bytes, containing the type of request (read, write, acknowledge), flow-control data, routing data (if needed), and coherency information (if applicable). The data message on the network is considered to be a header control message (8B) combined with a cache line (64B).

In the next section we go into deeper detail on the topic of interconnection networks, as well as network design considerations. We will also outline network characteristics, discussing the building blocks of a network, routing mechanisms, topologies, and flow control mechanisms.
CHAPTER 2. BACKGROUND

2.3 Interconnection Networks

Among the major contributions of this dissertation, a major focus is the analysis and optimization NoCs for existing and future scalable GPUs. In this section we provide a short background on the basic concepts related to interconnection networks, presenting those that are most relevant to a GPU.

2.3.1 Network Design Considerations

Interconnection networks play a major role in the performance of the modern parallel architectures. There are many factors that may affect the performance of a system. Duato [43] outlines many factors that has to be considered in designing a network for parallel architectures. Network performance, scalability, simplicity, physical constraints, and cost are just a few of these factors.

Performance. Performance is easily the most important factor in the design of a parallel architecture. Nodes in a parallel architecture typically require to communicate with each other. Internode communication is usually performed through explicit messages generated by the processing units. A node can inject messages into the network, targeting a destination node, where the message will be ejected from the network. The time it takes for a message to traverse the network has a direct impact on the system that utilizes this network. Message latency is the time elapsed between the time the message is generated by a source node, to the point in time when it is delivered at the destination node. Message latency may increase the time it takes a processing unit to access its memory, and in turn increase the idle time of that processing unit. The term saturation is an operating point where the network is unable to service any new messages, due either to the high number of injected messages in the network, or the network’s incapability to deliver the in-flight messages as fast as new messages are injected. At this point, the maximum throughput of the network is reached. Throughput is the amount of information delivered by the network per time unit, and is probably the most important metric for evaluating the performance of a network.

Scalability. The term scalability refers to capability of the network for support growth in the intercommunication between nodes. As more nodes are added to the network, network bandwidth should increase proportionally. Otherwise, the lack of network scalability will decrease the overall efficiency of the system. Also, as we scale the system, the network should scale proportional to the amount of the scaling, while maintaining the same performance. Additionally, interconnection networks should provide incremental expandability. This means they should allow the addition
Simplicity. Networks within a parallel architecture are usually designed with a fixed clock frequency. This allows the transactions in the network to be processed in a pipelined fashion. Using a fixed clock frequency reduces the inefficiencies that might result from working with components in different clock domains. The clock frequency of a network defines the duration of a cycle in that network, which is directly related to network latency. A network that leverages a simple design can enjoy higher frequencies (to match or exceed the frequency of its nodes – processing units and memory units), and therefore achieve a higher throughput and performance. Additionally, a network that is simple and easy to understand, can also be easily tailored to the needs of its components, which can be exploited for achieving a higher performance.

Physical Constraints. Some of the most important challenges to be considered in designing a network are physical constraints, such as packaging, wiring, and maintenance. As we increase the number of nodes within a system, the number of wires required to interconnect these nodes increases. Operating temperature, wire-length limits, wire-space limits, and number of pins per board dedicated to communication channels, are among the factors that can be impacted as we increase the number of nodes in a system. Each of these constraints adds to the complexity to the network design.

Cost. An ideal network is typically too expensive to realize in practice. Increases in area and power in the network design also directly impact the cost of the network. Design decisions are commonly a tradeoff between cost and performance. However, these two factor are not always directly proportional. A simple and cost effective network can still achieve performance approaching that of an ideal network.

2.3.2 Building Blocks

The common building blocks of interconnection networks are buses, routers, and links.
CHAPTER 2. BACKGROUND

Figure 2.5: Two virtual channels multiplexed across a physical channel.

2.3.2.1 Bus

Buses are the main components in designing a shared-medium network. A shared-medium network allows all the nodes to share the same communication medium (i.e., bus) in order to communicate with each other. Both data and control information must be carried on the bus. Control signals include bus request signals from the nodes, and request grant signals from the arbiter. The arbiter involves centralized logic, with dedicated control lines on the bus that grants permission to nodes to transmit data on the bus. Specifying the types of permission granted to nodes is yet another design decision, and include release-when-done, release-on-request, and time-share schemes [43].

Figure 2.4 shows a high-level view of a bus-based shared-medium network that uses centralized arbitration. Each node in the network has to listen to the bus to check whether it is the destination of the message that is being transmitted on the bus. A major benefit of the shared-medium network is the support for bus broadcast. During a broadcast, every node sees the data sent on the shared medium, and can receive that data. Multicast is a special case of the broadcast where only a subset of the nodes can receive the data that is carried on the bus.

Shared-medium networks that share a single bus are not scalable because, as the number of nodes in the system grows, the network will become a bottleneck. To increase performance and scalability, a shared-medium network can leverage multiple buses in order to allow multiple devices to communicate with each other concurrently. The network can be broken down into separate paths that provide connectivity between nodes in the network [44]. Alternatively, a bus can have multiple backplane lanes (a multiple-bus organization), and can use a central arbiter to allow multiple messages to traverse the network concurrently [45]. We describe how we use this strategy to design our crossbar topology in Chapter 3.
CHAPTER 2. BACKGROUND

2.3.2.2 Links

Point-to-point links provide physical channels, versus shared-medium buses, allowing nodes to directly connected to each other to communicate. Any communication between non-neighboring nodes requires transmitting the information through several intermediate nodes. Each point-to-point link requires a link controller that handles the flow of messages across its physical channel. A link controller is placed on both side of the physical channel to coordinate transfers of data and control information. Links also leverage buffers, located on the transmitting and receiving nodes, to store messages that will be placed on the physical channel in a First-In, First-Out (FIFO) manner. A link may support several logical or virtual channels multiplexed across the physical channel. Each unidirectional virtual channel is realized by an independently managed pair of message buffers, as illustrated in Figure 2.5.

The performance of both links and buses are affected by the length of the wires that are used to construct these components. A signal will diffuse as it moves along the pulse edge, becoming more dispersed as a function of transmission distance. The time it takes to transfer one strong signal over a wire of the link is known as the time of flight, and any additional cycles resulting from the time of flight are considered Transmission Latency. If the signal loses its strength after a certain distance (i.e., due to resistance), then it should be strengthened using a repeater to restore the original signal for the receiver. Repeaters can help to avoid signal transmission errors.

2.3.2.3 Routers

A network that uses solely point-to-point connections between all nodes is not scalable solution. This is mainly due to the excessive use of wires for creating physical links, which results in high power dissipation and space issues. Also, point-to-point networks usually result in a waste of resources since the nodes usually have one target to talk to, while physical channels to other nodes are remain unused.

Router-based networks are commonly used to avoid the high cost of the network, while providing stability and expandability. Routers are components within the network that handle the message passing between nodes by forwarding the messages on the physical links along the path between two nodes. Each router is connected to neighboring routers usually through two unidirectional physical channels. Alternatively, a bi-directional channel may be used to connect two neighboring nodes. In router-based networks, each node is usually connected to a router. As the number of nodes in the system increases, the number of routers in the system increases as well.
Figure 2.6 shows a detailed design of a baseline router microarchitecture. A baseline routers have four separate stages: i) Virtual Channel Allocation (VA), ii) Switch Allocation (SA), iii) Switch Crossbar Traversal (ST), and iv) Link Traversal (LT).

Two tasks are performed in the VA stage. First, the data received by the input ports of the router is buffered. Second, the buffer lets the neighboring router know whether it can receive new messages. The latter is part of flow-control mechanism of the network. The flow control mechanism establishes a dialog between sender and receiver routers (two neighboring routers), allowing and stopping the advance of messages. If a message is blocked, it requires the some buffer space to be stored. When we run out of buffer space, the flow control mechanism stops information transmission. Credit-based flow control uses credits (as shown in the figure). When the messages advances and buffer space is available in the current router, a credit is send to the previous router. In credit-based flow control, each router has counters that indicate the number of empty slots in the downstream routers directly attached to it. If there is no space left in the buffers of the attached router, the initial router cannot send the message forward until buffer space becomes available (a
CHAPTER 2. BACKGROUND

credit is received).

The SA stage, implements the routing algorithms, selects the output link for an incoming message, and sets the switches within the crossbar accordingly. If multiple messages simultaneously request the same output link, this stage must provide for arbitration between them. The router, shown in Figure 2.6 has 5 output links (one in each direction, and a link to a receiver node).

In the ST and LT stages the message traverses a switch crossbar to the destination link, and traverses that link to the next router. These two stages are the only ones that carry the actual data (identified by green in the figure), and all the remaining stages are considered router overhead. Therefore, they can be customized based on the needs of the network. One optimization is to remove the VA stage when virtual channels are not being used (highlighted in the figure by orange). Additionally we can combine the Switch Allocation and Switch Traversal stages by using lookahead signals that are sent one cycle ahead of the data messages [2]. The time it takes for a message to traverse the switch stages and reach the link is known as the switch latency.

2.3.3 Topology and Routing

A topology is a graph with nodes and routers as vertices, and buses and links as edges. It highlights different paths between different nodes in a network. Topology is strongly coupled with the concept of routing. Given the topology of an interconnection network, multiple paths can exist between two nodes. Routing determines which path a message takes from the source to its destination. An efficient routing algorithm is critical for load balancing on a topology under different traffic loads, and to produce the maximum performance of a topology. Figure 2.7 outlines a number of different network topologies. It is important to note that the topology is a logical concept. This means it is not necessary for the network to have the same physical layout as the view of the graph provided by the topology. We will discuss several topologies for the interconnection network in more detail in Chapter 3.

Figure 2.7(a) illustrates a peer-to-peer ideal topology. The ideal topology is a network in which every node is directly connected to every other node. In this network, no message would ever pass through an intermediate node before reaching its destination. The number of links required to create this fully-connected graph is N-1, where N is equal to the number of nodes in the network. Clearly, the cost of such a network is prohibitive as the number of nodes grow to moderate and large sizes. The ideal network can also become impractical, since the number of physical connections of a node is limited by hardware constraints, such as the number of available pins and the available
Figure 2.7: Logical layout of different topologies. Routing determines which path a message takes from source to destination within the topology. One example of a route is identified in each topology (in red).

wiring area. Therefore, other more practical topologies have been proposed to balance the cost and performance of the network.

Figure 2.7(b) is a crossbar topology that is designed with a single router. Crossbars are among the most popular topologies, since they provide a minimal hop (intermediate node) count between nodes. However, they offer no path diversity and require very long wires to implement. Additionally, the crossbar presented in the figure is limited by the number of nodes that it can support, since the router radix is limited by the physical constraints as well. Radix describes the number of possible connections a router can provide between its neighboring nodes. Conventional microarchitectures do not scale to high radices since the complexity of the allocators in the routers scale quadratically with the radix [46]. Therefore, for larger networks, it is advisable to use a bus-based crossbar topology, as shown in Figure 2.7(c).

The bus-based crossbar uses multiple buses to provide the all-to-all connectivity between
CHAPTER 2. BACKGROUND

the nodes in a system. In the figure, many Multiple-Write, Single-Read (MWSR) buses are used to create this topology. The MWSR bus allows multiple nodes to inject messages to the bus, while only one node is the recipient of the injected messages. At any given time, only one node is allowed to inject messages to the bus. The injecting node and the duration of the injection is decided by the arbitration policy. Time-share, Release-When-Done, and Release-On-Request are among these arbitration policies [43]. The bus based crossbar still requires a centralized arbitration logic and long wires (which adds transmission latency), and can have a large arbitration latency (the time spent by the message waiting for its turn). In Chapter 3 we will discuss this topology in more detail.

The use of router-based networks allows the creation of, virtually, any topology. Mesh (Figure 2.7), Clos, and Butterfly, are among well-known interconnection network topologies. They exhibit different characteristics in terms of router degree, diameters (the maximum distance between two nodes in the network), regularity (a network is regular when all the routers have the same degree), and symmetry (a network is symmetric when it looks alike from every node). We will discuss some of the more suitable topologies for the GPUs later in Chapter 3.

2.4 Silicon-Photonic Technology

With the growing number of computing cores on a single die, the global interconnect between these cores is becoming a critical component, affecting both performance and power consumption of the system. In 2008, the International Technology Roadmap for Semiconductors (ITRS) highlighted that electrical interconnects will become inadequate to meet the speed and power dissipation requirements of a highly-scaled Integrated Chips (ICs) [47].

Additionally, the global interconnect has also started to affect the programmability of these systems. Electrical interconnect topologies such as Crossbar, Torus, and Clos rely on wires of differing lengths in order to map to physical layouts of a system. This means the physical distance between two cores can impact the performance of the system as a whole, and introduce latency non-uniformity [48]. Depending on the topology used, if a program uses two neighboring cores it may have a better performance than if it used two cores on opposite sides of the chip. However, choosing which cores to use is usually not under programmer control.

ITRS has identified CMOS-compatible optical interconnect technology as an alternative to metal-based electrical interconnects [47]. Ever since, silicon-photonic link technology has been studied as a potential replacement for electrical link technology for NoC designs on manycore systems [49, 48, 50, 51, 52].
CHAPTER 2. BACKGROUND

Figure 2.8: Photonic Link Components – Two point-to-point photonic links implemented with Wavelength-Division Multiplexing (WDM).

Silicon-photonic links can provide significantly lower latency, and an order of magnitude higher bandwidth density, for manycore systems, as compared to electrical links. The use of optical links solves the programmability problem of global interconnects, by equalizing the latency of short and long links (up to a much greater distance) [43].

2.4.1 Silicon-Photonic Links

Figure 2.8 shows a generic silicon-photonic channel with two links multiplexed onto the same waveguide. These two silicon-photonic links are powered using a laser source. The output of the laser source is coupled into the planar waveguides using vertical grating couplers. Each light wave is modulated by the respective ring modulator that is controlled by a modulator driver. During this modulation step, the data is converted from the electrical domain to the photonic domain. The modulated light waves propagate along the waveguide and can pass through zero or more ring filters. On the receiver side, the ring filter whose resonant wavelength matches the wavelength of the light wave “drops” the light wave onto a photodetector. The resulting photodetector current is sensed by an electrical receiver. At this stage, data is converted back into the electrical domain.

Each Electrical-to-Optical (E/O) and Optical-to-Electrical (O/E) conversion is a single independent stage, and each affects the fixed cycle duration of the network. Hence, by accounting for these conversions between domains when calculating the frequency of the network, we can make sure that each stage takes a single cycle. The time-of-flight for the data on the photonic link is always less than the frequency of the network, since the data travels at the speed of light. However, the serialization latency depends on the bandwidth of the photonic link. Arbitration, and queuing delays (the time spent in the buffers waiting to reach the E/O stage) also contribute to the overall latency of a transmission.
CHAPTER 2. BACKGROUND

2.4.2 Silicon-Photonic Integration

In terms of realizing silicon-photonic NoCs, two integration approaches have been proposed: i) 3D integration and ii) monolithic integration [4, 5, 53]. The 3D integration approach requires a specialized die or layer in the chip substrate. Different implementations of the 3D integration use different chemical materials such as silicon nitride [53], or buried oxide [54], to construct this separate layer or die, respectively. While a separate layer or die allows easier customization of the silicon-photonic links (to improve performance), it also adds to the number of processing and manufacturing steps. In the case of introducing a separate die, the IC needs to interface to two separate chips, which can lead to significant power consumption and area overhead [48].

On the other hand, using monolithic integration leverages the existing process layers of a standard logic process for photonic devices. Although monolithic integration may require some post-processing, its manufacturing costs can be lower than using 3D integration. Monolithic integration decreases the area and energy required to interface electrical and photonic devices, but it requires active area for waveguides and other photonic devices.

The two integration methodologies have many similar optical loss components. Optical loss is the gradual loss of intensity in the optical signal, and can impact system design. Optical loss limits set the required optical laser power, and correspondingly, the electrical laser power. In the critical path of the silicon-photonic links, some losses such as coupler loss, photodetector loss, and filter drop loss are relatively independent of the network layout, size, and topology. In addition to optical loss, ring filters and modulators have to be thermally tuned to maintain their resonance under on-die temperature variations. We will discuss the parameters chosen for the evaluation of silicon-photonic based networks, later in Chapter [4].

2.5 Existing Cross-Device Memory Management Options

In this section, we describe different methods used for the memory management of multi-GPU systems. Memory management is a fundamental part of application execution on the systems with GPUs. These approaches are either utilized by the user or by the runtime (and transparent to the user). Each approach emphasizes specific features that an executing application should receive benefits from.
CHAPTER 2. BACKGROUND

Figure 2.9: Memory-copy based approach used to manage memory for GPU applications.

2.5.1 memcpy Approach

As mentioned earlier, the traditional way for executing a GPU application on the GPU device is to initialize the data on the CPU or the host device, then copy that data to the memory of the GPU device, where the GPU can access data, and begin the application’s execution. The communication between CPU and GPU in this instance is a copy of the applications entire data address space. The process of copying the data between the CPU and the GPU, as shown in Figure 2.9, is performed by the user using Application Programming Interface (API) calls (for CUDA, the cudaMemcpy() is used). We refer to this method the memcpy approach, following the terminology used by Kim et al. [28]. This method allows the GPU to exploit data locality that may be present in an application, leveraging its own high-bandwidth memory, and accessing the data through its memory hierarchy. However, this method has been shown to be inefficient since the data copy process can consume a large number of cycles to complete, during which time the GPU cannot execute the application.

Moreover, these two traditional approaches are not easily extendible to multi-GPU systems. In Section 2.2 we talked about one possible way to divide an original ND-range, partitioning it into multiple separate ND-ranges for execution on multiple GPUs. Using traditional approaches, we would need to copy the entire data for the application to multiple GPUs. In this case, the main issue becomes the way in which coherency between the resulting data on the multiple GPUs is maintained. If we simply copy all of the data back to the CPU, data transferred from one GPU could be overwritten by another GPU [55].
CHAPTER 2. BACKGROUND

Figure 2.10: Remote access approaches used to manage memory for GPU applications.

2.5.2 zero-copy and GPUDirect

To avoid the complexities of the memcpy approach, a number of popular GPU programming frameworks have introduced the notion of a shared address space across the host (CPU) and devices (GPUs). This notion was first introduced by NVIDIA, through Unified Virtual Addressing [32]. This solution provides API calls that allow the user to allocate data on the dedicated memory of the CPU or the host memory. As shown in Figure 2.10(a), the user can pin data used by the GPU in the host (CPU) memory. This methodology is called the zero-copy approach. Any GPU access to that data is performed using a zero-copy operation [56]. This means the L2 cache units in the GPU can directly read/write a single cache line from/to the host memory without involving the GPU’s main memory.

While this method provides support for coherency between CPU and one or more GPUs (since the data is maintained in a fix location), it has its own drawbacks. The major drawback is the underutilization of the GPU’s high bandwidth memory. When using zero-copy, accesses are made to the host memory. Today’s GPUs have much higher bandwidth than the host memory (see Section 2.5.3 for more details). Another drawback of the zero-copy approach is the use of pinned pages to keep the data used by the GPU in the host memory until the end of the GPU kernel execution. This constrains the amount of physical memory available to the operating system.

For a multi-GPU system, zero-copy has another drawback. Zero-copy can suffer from contention between different GPUs when accessing the host memory. In order to remove this contention, and also utilize the memory of other GPU devices, NVIDIA introduced the GPUDirect
CHAPTER 2. BACKGROUND

feature [57]. GPUDirect allows the cores in one GPU to perform a remote access to the memory of another GPU, as shown in Figure 2.10(b). This type of access is similar in nature to the zero-copy approach, but the access is made to the memory of the other GPU instead of the host memory.

Accesses to the memory in both zero-copy and GPUDirect approaches suffer from the latencies imposed by the interconnection medium between the devices. The most conventional interconnect medium is Peripheral Component Interconnect Express (PCIe) that is a shared medium bus. So only one access can be active at a time over this medium, resulting in arbitration latencies for competing transfers from other devices connected to this medium. While PCIe has seen many improvements over its predecessors (PCI and PCI-X) in terms of bandwidth and efficiency (due to lower packet overhead), it still struggles to provide the high bandwidth needed for large data transfers. PCIe 3.0, with 16-lane links, is the most commercialized revision available presently. PCIe 3.0 has an effective bandwidth of 15 GB/s (peak bandwidth of 16 GB/s) [58]. On the other hand, GPU memories can provide $10 \times$ higher bandwidth [59]. PCIe’s lower bandwidth greatly impacts transactions between the memories, and the CPU/GPUs that are interconnected through this bus. Given these issues with PCIe, GPU vendors have started to deliver higher bandwidth solutions. NVIDIA’s NVLink supports peer-to-peer connections, with an effective bandwidth of 16 GB/s (peak bandwidth of 20 GB/s) per connection. NVLink supports up to 4 connections to other devices, providing an aggregate bandwidth of 80 GB/s [60].

2.5.3 Unified Memory

Any combination of the previous memory management approaches can be used by the user in their program. User has to bear the burden of the memory management between CPU and multiple GPU devices, using one or more of these approaches. The gained benefits of the fine-grained management of the memory can be overshadowed by its complexities. Therefore, to remove the burden of managing memory from the user, NVIDIA introduced Unified Memory (UM)[56]. Unified Memory enhances the Unified Virtual Address by using the CUDA runtime to transfer data between the host and devices in a user-transparent fashion.

The Unified Memory model, as presented by NVIDIA, performs management through the graphics driver and CUDA runtime libraries. With NVIDIA UM, the data is always allocated in the memory of an active GPU. Therefore, unlike memcpy, there is no initial memory copy to move the data from the host memory to GPU memory. The use of Unified Memory also enables the CPU to make changes to the data. The problem with the current software-based implementation is the large
CHAPTER 2. BACKGROUND

number of redundant memory copy operations between the host memory and GPU memory [38, 37].
We will further describe these shortcomings in detail in Chapter 5. section Memory Technology of the GPU

Unlike heterogeneous CPU/GPU devices that share a single unified physical memory, discrete GPUs have been using specialized Dynamic Random-Access Memory (DRAM) technologies that have evolved over time to meet the need of these devices.

2.5.4 Double-Data Rate Memory Technology

The two main performance metrics present in memory technology are the data per clock rate or data rate, and the memory bandwidth. The data rate is the amount of data that can be transferred per cycle by a single pin or input/output (I/O) bus, while the memory bandwidth is the data transfer rate for all of the available pins. There is a direct correlation between memory bandwidth and the data rate.

The most frequently used commercial technology for current GPU memories is Graphics Double Data Rate (GDDR), which provides substantially higher memory bandwidth than their CPU counterparts, the Double Data Rate (DDR) memory technologies. The fifth generation of the GDDR, GDDR5, provides data rates of up to 7Gbps, and a bandwidth of up to 28GB/s [59]. The main reason behind the high-bandwidth of the GDDR5 memory technology is the sensitivity of the GPU performance based on the available bandwidth versus memory latency [61].

GDDR5 has been shown to be effective in High-Performance Computing (HPC) systems, workstations, and the desktop systems. However the increase in the available bandwidth results in significant energy per access. This is while the fourth generation of DDR memory, DDR4, provides a data rate of 3.2Gbps at a fraction of the energy cost of the GDDR5 [61].

2.5.5 High-bandwidth Memory Technology

To provide high memory bandwidth with lower energy cost, the AMD has introduced the HBM technology. HBM is a die-stacked DRAM technology. Both DDR and HBM use the same command interface with a range of common commands such as ACTIVATE, PRECHARGE, READ, and WRITE. The DDR command interface is used since it is effective in reducing the pin count of the memory devices [62]. In essence, the DRAM memory is a 2-dimensional array, consisting of different rows and columns. The DRAM commands access these rows and columns. For instance, the ACTIVATE command opens the selected row or column to be read or written using READ and
CHAPTER 2. BACKGROUND

WRITE commands, while the PRECHARGE command is used to deactivate the open row(s) or column(s) [63].

Figure 2.11 shows the difference between the physical design layout of HBM technology versus the alternative GDDR5 technology. Unlike GDDR, in HBM multiple silicon die are placed on top of one another, and the connectivity between DRAM dies are provided by the Through-Silicon Vias (TSVs). HBM also includes a silicon interposer. The commands from the processor are routed through the command busses of the interposer instead of the substrate (as in DDR memories). This allows for an increase in the bus size. The use of interposer technology also results in an increase in the memory-area density, which means many stacks of DRAM silicon dies can be placed side-by-
CHAPTER 2. BACKGROUND

side on the interposer carrier.

This type of placement is shown in Figure 2.11(c), and is known as 2.5D chip stacking. An alternative to 2.5D stacking is the 3D chip stacking (or vertical stacking) where multiple silicon die (memory and processor) are placed on top of one another, and the communication between processor and the memory is directly provided by the TSVs (and not through the interposer). The vertical stacking can benefit from higher bandwidth between the processor and the memory due to common surface area between these components. But the amount of available DRAM will be less than 2.5D stacking, since the DRAM size is limited by the size of the processor. Additionally, as the available TSVs increase (to increase the bandwidth), so does the area overhead of the TSVs, since the TSV copper fill and silicon require “keep-out” regions, to account for thermal expansion [64].

The use of HBM 2.5D memory technology has been widely successful, so this technology is now incorporated in off-the-shelf commercial GPU devices such as the AMD Radeon Fury X [65]. Later in Chapter 5 we utilize this memory technology in our multi-GPU system evaluation.

2.5.6 Hybrid Memory Cube Technology

Hybrid Memory Cube (HMC) is an high-performance RAM interface for TSV-based 3D stacked DRAM memory. However, unlike the rival 2.5D High-Bandwidth Memory technology, it does not utilize the silicon interposer. Similar to HBM, each memory stack consists of a logic layer, however this logic layer is able to communicate with the logic layer of other stacks and the processor through a packetized memory interface. With this enhancement, a memory request can be routed within different cubes of the HMCs [66]. The communication with the DRAM stacks in the HMC are in the form of packetized requests. The base logic die of the stacked-DRAM memory in the packetized memory interface are more complex than the logic die of the stacked-DRAMs in HBM-like technologies. By including a memory controller, the HMC can remove the need of a memory controller on the processor. Since the layer includes a interconnect router, any topology between different cubes can be formed to enhance the performance of the GPU system [67].

2.6 Evaluation Methodology

2.6.1 Simulation Environment

Multi2sim [40] is a cycle-level simulation platform for modeling CPUs (x86), GPUs (AMD’s Southern Islands, and NVIDIA’s Kepler), and heterogeneous architectures. Multi2Sim
CHAPTER 2. BACKGROUND

Figure 2.12: The Multi2sim simulation philosophy for architectural modeling. The framework provides disassemblers, emulators, and detailed timing simulators, which are used to model different architectures (including x86 CPU, and AMD Southern Islands GPU). The visual tool is an additional feature used for detailed presentation of the simulation status/results.

uses a four-stage approach for performing and presenting the architectural simulation of a CPU-GPU based heterogeneous system, as shown in Figure 2.12.

Multi2sim follows a three stage paradigm for detailed modeling of different architectures. The **Disassembler** stage accepts the compiled executable binary (which contains machine instructions) and decodes these instructions into an alternative representation that allows a straightforward interpretation of the instruction fields, such as operation code, input/output operands, and immediate constants.

The **Emulator** accepts the output of the disassembler, and models the original behavior of the guest program. The emulator provides the illusion that the guest program is running natively on a given microarchitecture, and guarantees that the executing application on Multi2sim produces identical results to its execution on the native machine. To do so, Multi2sim keeps track of the guest program state, and dynamically updates it one instruction at a time until the program is over. The state of the program is expressed as the program’s virtual memory image and register file of the architecture.

The **Timing simulator** in Multi2sim provides a cycle-level simulation of an architecture by modeling each hardware stage of the processor pipeline, and its global memory components (i.e., cache units and interconnection networks). Multi2sim uses an event-driven simulation approach to schedule various events during the simulation, depending on the latencies that are imposed by hardware components for each event. The modeled hardware for the architecture includes pipeline stages, pipe registers, instruction queues, and functional units. While hardware structures are modeled in the timing simulator, the flow of instructions that utilize them are obtained from invocations
CHAPTER 2. BACKGROUND

Table 2.1: **AMD Radeon HD 7970 GPU specification.**

<table>
<thead>
<tr>
<th>Processor Cores</th>
<th>Memory System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fabrication process</td>
<td>28nm</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>925</td>
</tr>
<tr>
<td>Compute Units</td>
<td>32</td>
</tr>
<tr>
<td>MSHR per Compute Unit</td>
<td>16</td>
</tr>
<tr>
<td>SIMD Width</td>
<td>16</td>
</tr>
<tr>
<td>Threads per Core</td>
<td>256</td>
</tr>
<tr>
<td>Wavefront Size</td>
<td>64</td>
</tr>
</tbody>
</table>

to the emulator. The timing simulator requests emulation of a new instruction, after which the latter returns all information about the emulated instruction, as propagated by its internal call to the disassembler. During its travels across pipeline stages, an instruction can access different models of hardware resources such as functional units, effective address calculators, and data caches with potentially diverse latencies.

For the purposes of this thesis, we focus on the simulation of the x86 CPU as the host, and AMD Southern Islands GPUs as device, to execute an OpenCL application. We have modified Multi2Sim, adding in implementations of the electrical and silicon-photonic network components, required for the evaluations of the contributions of this thesis. As part of our modifications, we extended Multi2sim to provide support for packet processing and flitting support for NoCs. To validate the updated Multi2sim simulation framework, we leveraged the stand-alone network simulation mode of Multi2sim that can inject random traffic in various network topologies. We compared and verified the reported latencies and performance numbers in our cycle-based model against previous research [68, 69].

### 2.6.2 Target Architecture

#### 2.6.2.1 AMD Radeon HD 7970 GPU

To explore the NoC design in the memory hierarchy of a GPU, we build on an existing simulation model of a high performance AMD Radeon HD 7970 GPU from AMDs Southern Islands family. Compute Units (CUs) are clocked at 925 MHz, are manufactured in a 28nm CMOS technology node, and are tailored for general purpose workloads [1]. The die size for this GPU is
CHAPTER 2. BACKGROUND

(a) An image of the AMD Radeon HD 7970 GPU chip die.  
(b) The layout of the AMD Radeon HD 7970 GPU chip.

Figure 2.13: Extracting the detail of the GPU device based on the chip die.

reported to be 352 \( \text{mm}^2 \) and its maximum power draw is 250 watts [70]. Table 2.1 outlines the details of the different GPU components.

The global memory hierarchy of this GPU contains 40 L1 cache units, since the GCN architecture dedicates one vector L1 cache unit per CU, and one shared scalar L1 cache for every 4 Compute Units. The L2 cache is banked into six different partitions to reduce the load on the L2 and memory banks. Although not every detail of the interconnect used in the AMD Radeon HD 7970 chip has been publicly disclosed, online resources suggest that a crossbar topology is used between the L1 and L2 caches [71, 1]. Figure 2.13 show an image of the AMD Radeon HD 7970 GPU chip die [72], and the details of the physical layout of the components in this device.

2.6.2.2 AMD Radeon HD 7850 GPU

We choose a device from the AMD’s Southern Islands family as our baseline GPU device for analysing multi-GPU systems. The main reason for this choice is to be able to simulate a multi-GPU system with small number of compute units for each GPU. Reducing the number of compute units on each individual device allows us to focus primarily on the transactions between multiple GPUs and the host CPU device. However, we strongly believe that our design and evaluation are transferable to many other CPU and GPU architectures.

The targeted GPU for our evaluation is the AMD Radeon HD 7850, which has 16 compute units (CUs) clocked at 800 MHz [73]. Similar to Radeon HD 7970, the Radeon HD 7850’s CUs can execute 256 threads from one workgroup at a time. CUs are still equipped with the same types of L1
caches (dedicated vector L1, and shared scalar L1 caches) but these L1 caches are connected to four L2 caches, instead of 6 L2 caches. The application address space is interleaved across these four L2 units, addressable on a cache line granularity [1], and each L2 unit is connected to a separate main memory controller.

We use 3D-stacked DRAM memories for the main memory of this GPU architecture, which is used in the design of AMD’s next-generation high performance memory (see Section 2.5.3 for more detail). We have associated each memory controller of our target GPU with a single stacked DRAM. We leverage stacked-DRAM memories, since it is evident that GPU technology (and FPGAs) are moving toward this direction [59, 74, 75]. Our designs and analysis are inspired by the High-Bandwidth Memory technology (HBM), but are also applicable to packetized die-stacked memory interfaces (e.g., Hybrid Memory Cubes – see Section 5.4). Two different interconnection technologies, PCIe [58], and NVLink [60] are considered in this work. The details of the design of the multi-GPU system based on this GPU device are presented in Section 5.5.
Chapter 3

Asymmetric Network-On-Chip for Existing GPU Architectures

This chapter discusses our proposed Network-on-Chip designs to benefit a contemporary GPU architecture from AMD’s Southern Islands family. We study the impact of different NoC topologies across a wide range of GPU workloads, and identify the most attractive topologies for this architecture. By analyzing the impact of different on-chip interconnects and traffic pattern of OpenCL application, we propose alternative NoC designs that allow the applications to achieve similar performances to those best topologies, but at much lower power costs.

3.1 Related Work

Interconnection networks play an important role in systems that have large number of computing cores. As applications try to leverage more number of cores in a system, this will generate an increasing amount of communication across the memory network.

Generally in a manycore system, communication occurs between different cores, as well as between the different levels of the memory hierarchy. A large body of work has studied NoC designs for many-core architectures, targeting the power efficiency and latency of the communication medium. A broad range of network topologies for on-chip communication in Chip Multiprocessors (CMPs) has been explored. While there is a large body of prior work in this area, we limit our discussion to the most relevant research to the design tradeoffs considered in this dissertation.

Balfour and Dally [76] developed a detailed area and energy model for on-chip interconnection networks in order to design efficient networks for tiled chip multiprocessors. This design
space exploration describes the tradeoffs of different aspects of network design, such as topology, channel width, routing strategy, and the buffer size on performance, area and power efficiency of the system. The topologies considered in their work include Mesh, Concentrated Mesh, Torus, Fat-Tree, and Tapered Fat-Tree. Their detailed analytical area and power model for a 64-core CMP shows that architectures commonly assumed in on-chip networks studies (such as mesh) are unlikely to perform well in CMP systems as processor counts increase. Even with low-latency router architectures, a two dimensional mesh scales poorly in an on-chip environment. They demonstrate that replicating networks in an on-chip environment can improve performance while simultaneously improving area and energy efficiency. Topologies such as the Mesh and CMesh may accommodate a second network without increasing the die area, and with only a slight increase in network chip area. Therefore, a second parallel network is introduced that can increase performance while improving efficiency. Different strategies for distributing traffic over the subnetworks are evaluated.

This prior work has stood the test of time, as its technology model, layout-based network comparisons, use of multiple networks, and use of concentration are all still valid today. Both multiple networks and concentration are considered in our design-space exploration for our GPU NoC. However the multiple networks in their design is parallel, carrying the same traffic, while our model uses two parallel networks, one for each direction.

Kim et al. introduced the Flattened Butterfly [46], which has been shown to be a good alternative to the CMesh designs for CMPs [76]. They proposed the use of high-radix networks [77] in on-chip interconnection networks to design this flattened butterfly. In earlier work by Kim et al. [77], the high-radix router with many narrow ports is shown to be more effective in converting pin bandwidth to reduced latency and reduced cost than the alternative of building a router with a few wide ports. However, high-radix routers increase the complexity of the allocators and the wiring needed to connect them to the input and output ports (the complexity is increased by the square of the used radix). They address these challenges by proposing and evaluating alternative microarchitecture for high radix routers, including a hierarchical switch organization. Hierarchical crossbar switch is built by dividing the crossbar switch into subswitches where only the inputs and outputs of the subswitch are buffered. With per-virtual-channel buffers in each subswitch, the proposed network can deliver area savings and higher throughput as compared to a conventional crossbar implementation.

The design of the flattened butterfly leverages the high-radix routers to reduce the diameter of the network, and offers lower latency and energy consumption than conventional on-chip topologies such as a mesh. The flattened butterfly described in [46], exploits bypass channels such
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

that non-minimal routing can be used with minimal impact on latency. The bypass channel are channels that pass over other routers in the same row or column. That is, these channels bypass local routers to connect the source router of a message directly to its destination router. To map a 64-node CMP onto a flattened butterfly topology, a 3-stage radix-4 conventional butterfly network (4-ary 3-fly) is collapsed into a flattened butterfly that has 2 dimensions and uses radix-10 routers, and uses bypass channels. Unfortunately, the use of high-radix routers in on-chip networks introduces long wires. Kim et al.’s work [46] assumes optimally repeated wires to mitigate the impact of long wires and utilized pipelined buffers for multi-cycle wire delays. Two parallel proposals have consider how to improve interconnects via implementing variation of router bypassing [78, 2].

Joshi et. al. [79] also explore the design of energy-efficient low-diameter networks, such as flattened butterfly and Clos. Low-diameter networks are attractive as they can potentially provide uniformly high throughput and low latency across various traffic patterns, but require efficient global communication channels. Equalization is used in Joshi’s work to improve the bandwidth of a global communication channels in point-to-point interconnect. In an equalized interconnect, a feed-forward equalizer filter (FFE) is used instead of conventional repeaters to shape the transmitted pulse to minimize the intersymbol-interference (ISI) at the receiver side, supporting higher data rates. Equalization improves the latency close to the speed of light, and also the energy efficiency of the transmission. The low latency results from the fact that equalization uses the fast propagation velocity at high frequencies. Other work that has looked into reducing the latency and increasing the efficiency of the point-to-point global interconnects includes [80, 81, 82].

Mishra et. al. [83] propose an asymmetric resource allocation (buffers and link bandwidths) to exploit the non-uniform demand on a mesh topology, in the heterogeneous network configuration, for a CMP. They showed that better performance can be achieved by allocating larger resources (such as wider links and more virtual channels) to routers along the diagonals instead of homogeneous distribution of the resources. This work also suggests that the placement of the memory controller should be along the center of the chip, allowing memory to be more accessible to every core in the design.

The NoC design of CMPs has matured to the extent that several commercial designs are available. One such commercial design is the 80-tile, sub-100W TeraFLOPS processor in 65nm introduced by Vangal et al. [84]. Tiles are arranged as an 8x10 2-D array of floating-point cores and packet-switched routers, both designed to operate at 4 GHz. The on-chip interconnect is a 2-D mesh network that provides a bisection bandwidth of 2 terabits/s. Development of this device has provided some key insights on manycore architecture, including the amount of power consumed by
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

the network (one third of the entire chip), and that a tiled design methodology quadruples design productivity without compromising design quality [85].

The TILE64 [86] is another example of a commercial many core CMP which contains 64 cores and can execute 192 billion 32-bit operations per second in 1GHz. The tiled processor architecture leverages an iMesh interconnect that consists of 5 2D mesh networks, each specialized with a different use, such as transport medium for off-chip memory accesses, I/O, interrupts, and other communication activity. Having five mesh networks leverages the on-chip wiring resources to provide massive on-chip communication bandwidth.

Another related research area is the NoC design of the heterogeneous architectures, where CPUs and accelerator cores (such as GPUs) co-exist on the same die. Lee et al. [87] identify a novel trade-off in CPU-GPU heterogeneous systems concerning the NoC design. While CPUs run latency-sensitive threads, the GPUs demand high bandwidth. Lee et al. thoroughly survey the impact of the primary network design parameters on the CPU-GPU system performance, including routing algorithms, cache partitioning, arbitration policies, link heterogeneity, and node placement. Adaptive Virtual Channel Partitioning (VCP) was another technique suggested by Lee et. al. [88] for the NoC of their heterogeneous architecture. Through feedback-directed VCP for on-chip routers, an effective network bandwidth sharing between CPU and GPU cores in a heterogeneous architecture is achieved. VCP dedicates a few virtual channels to the CPU and the GPU applications with separate injection queues. Then the adaptive mechanism balances on-chip network bandwidth for applications running on the CPU and the GPU cores by dynamically choosing the best partitioning configuration.

Kilo-NOC proposed by Grot et. al. [89] is a NoC support for a system with thousands of interconnected components to result in high area and energy efficiency, good performance, and strong Quality of Service (QoS) guarantees. It includes a new lightweight topology-aware QOS architecture that provides service guarantees for applications such as consolidated servers on CMPs and real-time System on Chip (SoC). Network area- and energy-efficiency is improved through a novel flow control mechanism that enables a single-network, low-cost elastic buffer implementation. By isolating the shared resources into QoS-enabled regions, network complexity is minimized.

Kayiran [90] showed that in a heterogenous system with many CPU and GPU cores, the GPU applications tend to monopolize the shared hardware resources, such as memory and network, because of their high SIMT. Additionally, GPUs lack concurrency management, when employed in heterogeneous systems (similar to the topic we will explore in this thesis). To solve this problem, an integrated concurrency management strategy was suggested, which tunes thread-level parallelism.
in GPUs to control the performance of both CPU and GPU applications. This tuning was achieved by considering the state of the GPU cores, as well as the network congestion information, and system-wide memory congestion.

In regards to the NoC research proposed for GPU architectures, Yuan et al. [91] proposed a complexity-effective memory scheduler for GPU architectures, which leveraged from packet reordering mechanisms in the NoC routers to increase row-buffer locality in the memory controllers. As a result, a simple in-order memory scheduler (a modified router arbitration scheme coupled with a banked FIFO in-order scheduler) can perform similar to a highly complex, out-of-order scheduler. They evaluate their interconnection network arbitration scheme using crossbar, mesh, and ring networks for a baseline architecture of 8 memory channels, each controlled by its own DRAM controller and 28 shader cores with 64 in-flight memory request per shader. Similar to this work, Chen [92] designs a Network-on-Chip with coalescing network capability to allow efficient memory accesses for the GPGPUs applications. To do so an expanded NoC router that supports packet coalescing is designed and evaluated.

Prior to our work, the main NoC exploration performed for GPU architectures was done by Bakhoda [93, 17]. In [93], they explored on-chip networks for GPUs where the impact of different network parameters were evaluated. Much of the research in NoC has commonly focused on reducing network latency by improving different aspects of the NoC, including, lower-latency router microarchitectures [94, 95], or better flow control mechanisms [96, 78]. However, Bakhoda showed that accelerator applications are more sensitive to bandwidth than router latency, so providing higher bandwidth, versus reducing router latency, can better improve overall performance. Based on their earlier work, Bakhoda et al. proposed a new throughput-effective NoC for GPU architecture [17]. This design showed the effectiveness of using two kinds of routers (limited and full connectivity routers) in a mesh-based NoC. This design is based on the observation of many-to-few-to-many traffic patterns present in manycore accelerators. However, the limited connectivity routers allow traffic to be routed only in certain directions, and hence, is inefficient (as identified by the authors) for handling a large number of cache to cache transfers (i.e., coherence messages). Our baseline evaluations differ from their work in the following aspects. In [17], the design layout is not properly examined, nor is power estimation studied. Also, their hypothetical GPU assumed a smaller number of streaming multiprocessors and a higher number of memory controllers. This hides the effect of path diversity and congestion in the network.

In [97], Lotfi-Kamran analyzed the many-to-few-to-many traffic pattern (although for server applications) The many-to-few-to-many pattern was achieved by segregating server cores
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

and slicing last-level cache into different tiles. Similar to [17], they applied the concept of reducing connectivity to decrease the area of NoC. However, the applications under evaluation had high latency sensitivity, leading to much different results.

Kim et al. [98] show that by increasing the frequency of the NoC router results in a more cost-effective NoC. They proposed a direct all-to-all network overlay on a mesh (DA2mesh) in which a dedicated channel is provided between all the nodes in the reply network of the GPGPUs. By exploit the few-to-many traffic characteristics of the GPGPUs in the reply network, a DA2mesh network is formed that leverages channel-sliced 2D mesh network to overlay the direct all-to-all network. By providing direct channels between all the nodes, the DA2mesh removes all the contention in the network and removes the need for NoC router arbitration. This significantly reduces the router critical path and enables the router to operate at a higher frequency. This thesis work bears some similarity to this work, since we also analyze the different traffic patterns of GPU applications in different directions. However we propose to completely separate the two networks in two separate directions to increase the power efficiency of the network. This also allows us to tailor the topology for each direction.

Jang et. al. [99] also proposed an asymmetric NoC which utilizes VC monopolizing and memory partitioning schemes. They explore different memory controller placements and dimension order routing algorithms. The memory controller placement study is similar to the work of Abts et al. [18] which studied alternative memory controller placements for core-memory traffic in CMPs. The memory controller placement used in our mesh topology also leverages the same concepts discussed in [18].

Zhao at al. [100] take a different approach by designing a conflict-free network for the requests (instead of responses). This design results in area and power saving, with a slight increase in complexity. To support conflict-free networks in columns of the mesh, Zhao et al. deployed an exclusive subnet per column, and conflict resolution in the same column by tokens. The work Tu et al. [101] performs traffic-aware frequency scaling for balanced on-chip networks on GPUs. They examine the feasibility of scaling the network frequency dynamically to balance the throughput of the request and reply networks. The decision is guided by monitoring some shader cores to identify the memory traffic pattern. Jiang et al. [102] purpose a modified router microarchitecture to support specific stream paths on the packet-switched NoCs for stream processors, which results in a 16% latency reduction, introducing negligible area overhead.

There have a large of prior research on silicon-photonic link and interconnection network technology for designing the NoC for CMPs and GPUs. We present a thorough background on
previous related work to silicon-photonic based interconnects in Chapter 4.

3.2 Design Space Exploration of Interconnection Network Topologies

In this section we explore the trade-offs associated with various NoC topologies that can be used for interconnection between the cache units in the GPU memory hierarchy. We analyze a crossbar, a mesh, a concentrated mesh (Cmesh), a Butterfly, a concentrated crossbar, and a Clos network, to cover the entire spectrum from high-diameter, low-radix networks, to low-diameter, high-radix networks.

To have an objective discussion on this range of networks, first we have to match the bisection bandwidth across all topologies to provide a fair comparison of performance of the network topologies. The bisection bandwidth is the bandwidth required to achieve a balanced NoC design when considering the heavy off-chip demands of accelerator workloads. It is one of the physical constraints facing the bandwidth of the interconnection networks due to the available wiring area. Assuming that we have unlimited bandwidth for the network (although both wasteful and impractical), the GPU cores still become limited in performance due to lower bandwidth of the memory controllers. So it is rational to choose the overall bandwidth of the network based on the provided bandwidth of the main memory of the GPU.

Bakhoda et al. followed a similar procedure [17]. The bandwidth for each memory controller of the GPU memory is considered to be 16 bytes/clock, with 8 memory controllers in the GPU. The frequency of the memory device was considered to be 1107MHz. The bisection bandwidth of the network was chosen to be 70-80% of the peak memory bandwidth. However, this is much smaller bandwidth than today’s GPU memories can provide. A typical GDDR5 provides 28GB/sec per DRAM chip (the AMD Radeon 7970 has 6 DRAM chips embedded in the design) [59]. Looking further ahead, the new High-Bandwidth Memory Technology (HBM) memory provides more than 100 bytes/clock per stack (with many stacks on the chip die). So limiting the design of the network to the current memory bandwidth does not allow these designs to be relevant as memory bandwidth increases.

Hence we chose a much more aggressive memory bandwidth, and chose our bisection bandwidth to match this bandwidth accordingly. In our design, we consider the physical layout of the system, but chose not to consider the area of the network. By relaxing area constraints, we allow our analysis to be valid for future designs of the interconnect technology (which constantly aim to reduce the area of the network components [76]). The bisection bandwidth of the network
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

Figure 3.1: The use of repeaters along the wire to pipeline the signal transfer to avoid signal loss.

was chosen in such a way to allow one large message (72 bytes) to traverse each bisection links of a C-Mesh (which has the lowest number of bisection channels) in one cycle, without any serialization latency. In other words, the network provides the potential bandwidth for 8 messages from the memory to cross the chip’s bisection toward the requesting compute units in a single clock cycle (even if the memory itself is not capable of providing this bandwidth).

3.2.1 Topologies

3.2.1.1 Crossbar Topology

The first topology we consider is a global crossbar, which is generally considered to be the most well-known network topology. The layout of a crossbar is shown in Figure 2.7 in Chapter 2. Since we have 32 vector L1 caches, 8 scalar L1 caches, and 6 L2 cache units, we consider a $46 \times 46$ global crossbar that provide all-to-all connectivity between the cache units. This crossbar provides non-blocking connectivity between each pair of nodes, but the crossbar design is very challenging to layout. Since a crossbar requires a large number of global buses across the length of the chip, this can lead to significant power consumption in the wires, and high latency. Connecting a node from one corner of the chip to the opposite corner requires very long global wires, which means signals should be repeated throughout the transfer so they do not lose signal strength due to wire resistance.

Figure 3.1 presents a global wire that is designed using pipeline insertions. Wires usually exhibit signal loss as the length of the wire increases. As the length increases, the delay also increases quadratically. Repeaters are used to convert quadratic dependence into linear dependence on length. \footnote{For a typical \(0.6\mu m^2\) square wire, the delay is calculated to be \(1.9ns\) for \(10mm\). I.e., in a network with a 1GHz frequency, a repeater flipflop is required every \(5mm\).} Pipeline insertion allows the wire to transfer the data in a pipelined fashion from the source to the destination. In this figure, one flip-flop is placed along the wire, so while the first bit reaches its destination in the 2nd cycle from its transmission, the second bit reaches the destination.

\[\text{Figure 3.1: The use of repeaters along the wire to pipeline the signal transfer to avoid signal loss.}\]
Figure 3.2: Design of a crossbar NoC topology, using global MWSR buses for the GPU architecture.

in the 3rd cycle. In order to maintain uniform programmability over the crossbar interconnect, an average time of flight delay is considered for every bus in the design of the crossbar. The average is considered based on the physical layout of the NoC between the shortest distance (1 cycle for less than 5mm) and the longest distance (8 cycles for the maximum of 37mm – see Figure 2.13) that a bit traverses in this topology. Crossbars also require global arbitration, which can add significant latency and power dissipation. We place the arbiter in the center of the chip to minimize the arbitration overhead.

3.2.1.2 Concentrated Crossbar Topology

One shortcoming of a crossbar topology when used for large-scale NoCs is the amount of time each node has to spend waiting for their chance to transmit over a MWSR bus (arbitration latency). Using concentration, we can design cluster-based NoCs where the nodes within the same cluster compete with each other first, and then the winner of each cluster competes with the rest of the transmitters to transfer data over the bus. The concentration not only reduces the arbitration latency (by making it hierarchical), but it also reduces the number of buses required to provide all-to-all connectivity between the nodes. It comes with the cost of additional hop counts, due to the use of concentration at the beginning and the end of transmission over the bus.

Figure 3.2 shows the design of this concentrated crossbar. Considering the physical layout of the AMD Radeon 7970, we use concentration between the 4 adjacent compute units that share
one scalar L1 cache with with each other. Bear in mind that the figure is a logical layout, so 1) the crossbar does not map with the exact same arrangement on the chip, 2) the figure presents 46 transmitting unit and 46 receiving units, while in fact the transmitting and receiving units are mapped to each other, creating a single topology with only 46 units. The concentrated crossbar (C-crossbar) is an intermediate point between the conventional high-radix, low-diameter, networks and low-radix, high-diameter networks. The C-Crossbar utilizes the concentration of the neighboring cache units to reduce the required wiring. We allow the compute units that share the same scalar cache to leverage from the same concentration router. The L2 caches (and their associated memory controllers) are attached to the closest router.

### 3.2.1.3 Clos Topology

The concentrated Clos, shown in Figure 3.3, is another intermediate point between the conventional high-radix, low-diameter, networks and low-radix, high-diameter networks. It has uniform latency and throughput, which makes it attractive from the GPU programming perspective. However, this topologies also requires global on-chip communication channels, which as previously discussed, are expensive in terms of latency (and require pipeline insertion and/or repeaters).

In order to reduce the cost of global communication in Clos, we use the layout presented in Figure 3.4, which was inspired by the prior work of Joshi et al. [79]. Using this layout, the intermediate set of routers in the three-stage Clos design is concentrated in the middle of the chip, allowing shorter global communication between the concentration routers, and the routers in this

---

Figure 3.3: The Clos topology used as the interconnection network in the GPU architecture.
Having shorter global wires decreases both latency and loss of the wires, and reduces the number of pipeline flip-flops required to implement the wire. By placing the intermediate set of routers at the center of the chip, the data always travels through the intermediate set of routers in the middle of the chip, which results in uniform latency and throughput across various traffic patterns.

For the Clos network discussed in this thesis, we consider a 6-ary 3-stage Clos network. This means the first stage of the Clos has concentration routers with 5 to 6 nodes connected to them. The vector L1 cache of four neighboring compute units, the shared scalar L1 cache between these four compute units, and possibly a neighboring L2 cache units are the nodes that are connected to the first stage of the Clos. The Clos network provides a great degree of path diversity. Each node has the possibility to take multiple routes to its destination. The routing for the Clos is non-deterministic. This means each of these paths can be chosen. However, this does not necessarily means that the least congested route will be taken between routers of different stages. A random path is chosen for the messages from the routers of the first stage, to the routers of the intermediate stages during the transmission of the message. Additional hardware support is required to take advantage of adaptive route assignment in the Clos, in order to achieve better performance.

One drawback of the Clos network is the large number of bisection channels (64 channels). Since we use bisection matching to provide a fair comparison between different topologies, each channel has a much narrower bandwidth in comparison to alternative designs. The only other topology that has similar narrow channels is the crossbar, because it also has very large number of
Chapter 3. Asymmetric Network-on-Chip for Existing GPU Architectures

Figure 3.5: A 6-ary 4-fly conventional butterfly topology for the GPU architecture.

3.2.1.4 Conventional Butterfly

Figure 3.5 presents the logical layout of a 6-ary 4-fly conventional butterfly. The topology uses concentration in the first stage (the fly) of the butterfly between 4 L1 vector caches, 1 shared scalar L1 cache, and possibly a L2 cache unit. The conventional butterfly benefits from low-radix intermediate routers to provide multiple paths between the cache units in the memory hierarchy. The intermediate stages benefit from routers with $2 \times 2$ radices, which consume much less power than the alternative routers used in the intermediate routers of the Clos network ($8 \times 8$).

The conventional butterfly has 8 channels that cross the bisection in the chip, hence each channel is wider than the alternative Clos or crossbar topologies (32B per channel). We observe less path diversity in the conventional butterfly compared to a flattened butterfly. However, we favor the conventional butterfly over a flattened butterfly [46]. That is because one side of each communication in the network is typically a L2 cache unit. Since the L2 cache units are partitioned across different banks that are connected to different routers of the last stage of the butterfly topology, the network load becomes balanced by itself.

Additionally, the conventional butterfly uses shorter wires, so global point-to-point interconnects are not required. By using shorter wires, we avoid the additional cost and latency associated with the use of pipeline insertion. Each message in the network traverses four intermediate
routers before reaching its destination. We considered additional channels that are longer than 5mm between the nodes and the concentration routers in our evaluation. This introduces some latency during concentration, but allows for an easier and more flexible physical layout of the butterfly topology.

### 3.2.1.5 Mesh Topology

Figure 3.6 shows a 5 × 8 mesh network. The mesh is a commonly used topology in today’s commercial multicore designs [84, 86]. Ease of implementation is a major reason behind the popularity of the mesh. However, the mesh network has a high average hop count, which results in higher latency than low-diameter network. This high-diameter, low-radix, network uses shorter channels. But the mesh is harder to program, due to the challenge of mapping memory controllers (and their associate L2 caches) so they are uniformly accessible uniformly by all the compute units (and their associate L1 caches). For this thesis, we consider a diagonal placement for the L2 cache banks to minimize the average packet latency and request-response variance. By placing the L2 caches on the diagonal of the mesh, we can help keep traffic concentrated to high activity regions of the network [83].

The mesh network studied in this thesis utilizes Dimension-Order Routing (DOR) routing [104]. In DOR, the messages are first routed toward the destination router along one dimension of the topology (such as the horizontal dimension, X). The number of hops on the first dimension is based on the minimum distance. Then the message continues along the next dimension (e.g., the
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

(a) A deadlock scenario in a $2 \times 2$ 2D submesh. Four messages are blocking each other from making forward progress.

(b) Using XY routing to avoid deadlocks. Each message has to follow the same order to move in the network, avoiding circular dependencies.

Figure 3.7: Deadlock, and deadlock avoidance using XY routing in 2D mesh [3].

vertical dimension, $Y$) until it arrives at the destination. We have implemented a XY DOR in the 2D mesh evaluated in this thesis. DOR routing is also a mechanism for deadlock avoidance. Deadlock in routing is a byproduct of circular dependencies in the topologies. A circular dependency is formed when two (or more) messages are holding two resources (e.g., buffer space) while each message is waiting for the other message to release its resource. A circular dependency between four messages, $P_0$ to $P_3$, are shown in Figure [3.7]a. This is avoided in XY routing (and its alternative $YX$ DOR routing) since each message has to traverse the same dimension first (i.e., $X$) and then traverse the second dimension, leading to only one dimension change per message, as shown in Figure [3.7]b).

3.2.1.6 Concentrated Mesh Topology

Figure 3.8 shows an example of a mesh with a concentration of 6. Here, we balance the lower router and channel count with the higher router radices and channel lengths. The Cmesh has $5 \times$ fewer routers than mesh, and the number of channels in the Cmesh is also consequently much lower than the mesh. However, each router now has a much larger router radix.

Similar to mesh, Cmesh leads to non-uniformity during programming. We also use the same routing scheme as we use in the mesh for our Cmesh topology. The XY DOR routing is used to avoid deadlock in the routing of the messages. This routing algorithm is deadlock free, and therefore, introduces no additional constraints on the flow control policy. Similar to the butterfly
topology, concentration is used between vector L1 caches of four neighboring compute units, and the scalar L1 cache (which they all share). We also considered using a longer channel between the nodes and the concentration routers to allow more flexibility in the layout of the Cmesh. While the concentration introduces extra latency, it makes the physical implementation of the Cmesh topology on the die of the target chip possible.

### 3.2.2 Summarizing the Characteristics of the Topologies

Table 3.1 includes the key architectural parameters of these NoC topologies. The bisection bandwidth is matched across all topologies for a fair comparison of performance of network topologies. Networks are sized to support two types of messages that create the GPU traffic. In this table we list the number of channels for each topology. The networks are designed to have the same channel width for every channel in the topology. This channel width is decided based on the bisection matching.

Different topologies can have different hop counts. For example, the butterfly topology has fixed 4-hop routes between every source and destination node in the network. Mesh and Cmesh provide a non-uniform number of hop counts that can range from 2 to 12 hops (8 horizontal hops, followed by 4 vertical hops), and 1 to 5 hops, respectively.

As mentioned earlier, for the design of Cmesh, butterfly and mesh, the addition of a link between the nodes and the concentration routers (denoted by $T_C$) is considered. This allows for a more flexible physical layout of these networks on the chip die. However, we do not consider this for the crossbar, nor the mesh, since they are more straightforward to lay out. The crossbar, C-crossbar
Table 3.1: Network Configuration - Networks are sized to support two types of messages that create the GPU traffic. \( N_C \) = number of channels, \( b_C \) = bits/channel, \( N_{BC} \) = number of bisection channels, \( N_R \) = number of routers, \( H \) = number of hops along data path, \( T_R \) = router latency, \( T_{GR} \) = average latency of global connections (if used), \( T_C \) = channel latency, \( T_{TC} \) = latency from cache units to the concentration switch, \( T_S \) = serialization latency, \( T_0 \) = zero-load latency. Both types of messages are considered in the latency calculations. The latency values are separated by “,”.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Channels</th>
<th>Routers</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( N_C ) ( b_C ) ( N_{BC} ) ( N_{BC} \cdot b_C )</td>
<td>( N_R ) Common Radix</td>
<td>( H ) ( T_R ) ( T_{GR}(avg) ) ( T_C ) ( T_{TC} ) ( T_S ) ( T_0 )</td>
</tr>
<tr>
<td>Butterfly</td>
<td>48 ( 32 \times 8 ) 8 2048</td>
<td>32 ( 2 \times 2 )</td>
<td>4 1 n/a 1 1,3 0,2 9,19</td>
</tr>
<tr>
<td>Clos</td>
<td>128 ( 8 \times 8 ) 64 4096</td>
<td>24 ( 8 \times 8 )</td>
<td>3 1 2 n/a 1,9 0,8 8,36</td>
</tr>
<tr>
<td>C-crossbar</td>
<td>8 ( 32 \times 8 ) 8 2048</td>
<td>16 ( 8 \times 8 )</td>
<td>2 1 4 n/a 1,3 0,2 8,15</td>
</tr>
<tr>
<td>C-mesh</td>
<td>20 ( 72 \times 8 ) 4 2304</td>
<td>8 ( 9 \times 9 )</td>
<td>1 – 5 1 n/a 1 1 0 3 – 11</td>
</tr>
<tr>
<td>Crossbar</td>
<td>46 ( 8 \times 8 ) 46 2944</td>
<td>1 ( 46 \times 46 )</td>
<td>1 n/a 4 n/a 0 0,8 5,13</td>
</tr>
<tr>
<td>Mesh</td>
<td>134 ( 32 \times 8 ) 10 2560</td>
<td>40 ( 5 \times 5 )</td>
<td>2 – 12 1 n/a 1 0 0,2 5 – 51</td>
</tr>
</tbody>
</table>

and Clos use global point-to-point connections, which introduce additional time of flight latency due to the use of pipeline stages inserted the wires \( (T_{GC}) \). A channel may also encounter serialization latency, depending on the width of the channel and the size of the message that traverses that link. The serialization latency is \( n \) cycles for the link, where \( n = \lfloor \frac{\text{message size}}{\text{link bandwidth}} \rfloor \) (e.g. 4 cycles for a 72B message over a 16B link).

The zero-load latency \( (T_0) \) is the time it takes for a message to traverse the network from its source to its destination, assuming no competition for network resources from other messages. Messages in the network come in two formats, either 8 bytes (control only) or 72 bytes (control and data together). In the case of uniform networks (such as the butterfly, which has a fixed latency between source and destination), the zero-load latency reported in the table includes both types of messages, separated by “,“. For non-uniform networks, two numbers (separated by “-“) are provided, indicating the range of possible zero-load latencies for different message types over different paths of that topology.

### 3.2.3 Evaluation

To begin our study of GPU system performance for different NoCs topologies, we utilize applications from the AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) [19]. AMD has provided this SDK to highlight efficient use of the AMD Southern Islands family of GPUs. For each application, we can change program inputs to specify the workload in-
Table 3.2: Workloads from the AMD APP SDK.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV</td>
<td>Simple Convolution</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transforms</td>
</tr>
<tr>
<td>DWTHAAR</td>
<td>One-dimensional Haar Wavelet Transform</td>
</tr>
<tr>
<td>FWSHALL</td>
<td>Floyd-Warshall Shortest Path Calculation</td>
</tr>
<tr>
<td>FWT</td>
<td>Fast Walsh Transform</td>
</tr>
<tr>
<td>HIST</td>
<td>Histogram</td>
</tr>
<tr>
<td>MATMUL</td>
<td>Matrix Multiplication</td>
</tr>
<tr>
<td>MT</td>
<td>Matrix Transpose</td>
</tr>
<tr>
<td>RED</td>
<td>Reduction</td>
</tr>
<tr>
<td>RG</td>
<td>Recursive Gaussian Filter</td>
</tr>
<tr>
<td>SOBEL</td>
<td>Sobel Edge Detection Algorithm</td>
</tr>
<tr>
<td>URNG</td>
<td>Uniform Random Noise Generator</td>
</tr>
</tbody>
</table>

The power usage of each NoC is estimated using a detailed transistor level circuit model, the physical layout, the flow control mechanism and network traffic workloads. The wires in the crossbar, butterfly and Clos topologies are designed to be implemented in the global metal layers using pipelining. All of inter-router channels in all of the mesh-based topologies are implemented in the semi-global metal layers using standard repeater wires. The power dissipated in the SRAM array and crossbar of the routers is calculated by adapting the methodology described in previous work [105, 106, 107].

The dynamic power dissipation of the SRAM arrays include wordline capacitance dissipation, bitline capacitance dissipation and short circuit power consumption. The output driver of the row decoder charges the wordline, which results in wordline power dissipation each time the row is...
read/written. The bitline capacitance dissipation is based on the power dissipation of the read cycle and the write cycle [105].

Analytical power modeling in [105] is superior to alternative power models [108, 109] since it provides a more accurate leakage power model of the SRAM arrays. The leakage power usually depends on supply voltage, number of transistors, circuit states, temperature, and other process conditions. Hence, the five states of the circuit are defined as (pre-charge, read selected, read not selected, write selected and write not selected). Based on these states, the circuit is broken down into three major components, wordline drivers, memory cells, and pre-charge circuits, and each circuit is evaluated in different states for leakage measurements to extract the leakage power using HSPICE [110].

Figure 3.9 presents a comparison of the various NoC designs for the targeted GPU architecture in terms of performance. We normalized the performance of the GPU with different NoCs for each application using an ideal NoC with a fixed 3-cycle latency. The 3-cycle latency is the lowest latency we can have in our design, which is the minimum zero-load latency of the Cmesh. As shown in this Figure, the mesh, Cmesh and butterfly NoCs exhibit comparable performance across all the applications. The highest performance is achieved by a Cmesh network. On average, the performance of the Cmesh is 71% the ideal network, while the mesh and butterfly both achieve a performance (on average) equal to 69% of an ideal network.

The wavefronts of a workgroup that reside on a single CU can start their execution as soon as their data becomes available through the global memory hierarchy. The latency associated with retrieving the required data will determine the start of execution of the wavefront. Since L2 caches in the GPU architecture are partitioned, the required data for the wavefronts may reside in
 CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

a number of L2 units (and not just in a single unit). Any network that combines L2 traffic from multiple compute units (that are executing different wavefronts), can impact the performance of multiple CUs. Because the mesh, Cmesh, and butterfly NoCs provide multiple paths between L2s and individual L1s, they lead to higher utilization of the CU. On the other hand, the crossbar, Clos and C-crossbar, which at some point combine the traffic from the L2s, achieve lower performance.

Clos has the lowest performance in comparison with other topologies (16% of an ideal network). While Clos uses routers with the same radix as a C-crossbar, it is designed with the narrowest channel width (8 bytes), to match the bisection bandwidth of other topologies. While Clos provides the most path diversity between links, the routing algorithm currently does not leverage this diversity. A better adaptive routing algorithm should solve the issue we observe in the Clos topology. On the contrary, any alternative routing algorithm would still not benefit the Clos since each message is still serialized over the global links (8 cycle serialization latency + 2 cycle global point-to-point time of flight for each link). For Clos, the best solution might be to transfer bits of the same message on different available paths at the same time. However, this would increase design costs to support such a sophisticated routing mechanism, making this design costly.

We observe that the URNG and HIST applications have the best performance, regardless of the topology used in the GPU system. This is mainly due to their compute-intensity and the memory access patterns present in these applications. Both applications have a small memory footprint, so the impact of the global memory is not significant on their performances. For the FWT application, some of the topologies diverge from the performance of the ideal topology. The main reason for this is the irregular sparse access pattern of this application. Applications that exhibit irregular access patterns place non-deterministic non-uniform pressure on different partitions of the L2 caches, making performance more dependant on available bandwidth of each memory bank, rather than the bandwidth provided by the network.

Figure 3.10 illustrates the power breakdown of the different NoC topologies. Among these topologies, Cmesh has the highest static and dynamic power consumption. The highest static power can be attributed to the fact that Cmesh has the widest channel width (72 bytes) and it uses high-radix routers. The power dissipated by the wires in the 72-byte links is the source of this higher power dissipation. At the same time, the power consumed by the crossbar within the router is large since power is directly correlated with the flit size, which in this case is 72 bytes. Clos has the lowest performance in comparison with other topologies, but it also uses the least dynamic power compared to all other topologies. While Clos uses routers with the same radix as a C-crossbar, it is designed with the narrowest channel width (8 Bytes). Moreover, in a Clos layout, we placed
all the middle-stage routers together in the center of the chip to provide shorter global channels, which in turn results in lower power consumption, as suggested by Joshi et al. [79], and confirmed in Figure 3.4.

The butterfly is the next topology (after Clos) that exhibits the lowest static power dissipation. In a butterfly topology, the router radix is very small in comparison to all other alternative designs ($2 \times 2$). Even if the butterfly has $4 \times$ more number of routers in its design compared to Cmesh, it still dissipates substantially less static power due to using routers with lower radices. The power dissipation of the C-crossbar and crossbar are comparable since the power dissipation resulted from additional global wires in the crossbar is compensated by the concentration routers used in the design of the C-crossbar.

While Cmesh is the most power consuming topology in our exploration, it is also the most effective in reducing the execution time of the applications. In order to provide a method of fair comparison between these topologies that can uniquely describe the best topology, we leverage from the Energy-Delay-Product (EDP) metric. EDP allows evaluation of the trade-offs between circuit level power saving techniques for digital designs. Smaller energy-delay values imply a lower...
energy solution at the same level of performance, which means a more energy-efficient design [111].

Figure 3.11 shows an overall comparison of all the topologies using the EDP metric, which jointly accounts for changes in performance and energy. We have normalized the EDP of each design based on the EDP of the Clos, which has the highest EDP among all the network topologies which we evaluated. As shown in the figure, a butterfly network achieves the lowest EDP across all the applications (0.10% of the EDP of the Clos, averaged across all the benchmarks). The mesh and Cmesh are the next best topologies that exhibit low EDP, 0.12% and 0.31% of the EDP of the Clos, respectively.

The only exception is the EDP achieved by different topologies for the URNG application. Obviously, due to the lower global memory activity of this application, performance remains the same under different topologies (except Clos), so the static power dissipation of the topologies becomes the major factor impacting the EDP metric. That is why the Cmesh has the highest/worst EDP for URNG, while the butterfly network has the lowest/best EDP.

In the next section, we propose the design of an energy-efficient asymmetric NoC architecture, based on our analysis assuming different topologies (butterfly, mesh and Cmesh) across our range of application.

### 3.3 Asymmetric NoC Design for the GPU architecture

In the previous section we compared the use of different NoC topologies for our target system while running a variety of applications. In this section, we analyze the trends found in our
As mentioned in Chapter 2, the OpenCL data parallel programming model achieves scalable performance through independent execution of workgroups. Applications written in OpenCL exhibit unique memory access patterns, and hence the NoC can be designed based on these expected memory access patterns to maximize NoC energy efficiency.

The amount of communication between cores (L1-to-L1) in a GPU is limited. This limited communication includes interaction due to false-sharing (where two cores share different sub-blocks of the cache-lines), and limited cache coherency signals (if any), which are enforced by the AMDs relaxed consistency model \[1\]. At the same time, the latency of L1-to-L2 traffic is typically not critical, as the GPU programming model enforces the fact that no other CU should be stalled awaiting completion of a write-back transfer. Moreover, GPU workloads exhibit lighter traffic for L1-to-L2 accesses due to architectural enhancements such as coalescing.

Figure 3.12 breaks down the percentage of different types of communication between the cache units (L1-to-L1, L1-to-L2, and L2-to-L1) for our selected applications. We consider the knowledge of the expected network traffic, and the results presented in this figure, to enhance our NoC design.

As a first step for shaping our proposed topologies, we consider two separate networks, one for each direction of the communication, i.e., L1 to L2 and L2 to L1. We consider this strategy for two reasons: 1) we can improve the performance of both networks in terms of latency by restricting the traffic to one direction, and 2) we can design asymmetric logical networks optimized for these communication patterns.
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

Figure 3.13: (a) The L1-to-L2 network for MeshX2, (b) The L2-to-L1 network for MeshX2. The use of XY routing leads to several unused links. These links were removed to reduce the static power consumption and switch radices.

As a second step, we eliminate the paths between the L1 cache units and replace any L1-to-L1 communication with a L1-to-L2 transfer followed by a L2-to-L1 transfer. While this slightly increases the traffic between the L1-to-L2 and the L2-to-L1, it allows us to reduce the power consumption in the network. This reduction in power is due to a lowering of the number of physical links and router radix. For example, in the L1-to-L2 network in Cmesh, 5 L1s and one L2 (in 6 out of 8 routers) are connected to each concentration switch unidirectionally instead of bidirectionally, which leads to a reduction in the number of wires (32 \times 8 wires are eliminated for each. Links also become unidirectional – 1280 wires per concentration switch). We also see a reduction in the router radix, from 10 \times 10 (or 9 \times 9) to 8 \times 4 (or 7 \times 4).

3.3.1 Asymmetric Topologies

Based on our study of symmetric topologies, we propose the use of asymmetric parallel networks, one for L1-to-L2 communication and one for L2-to-L1 communication. We eliminate the links that create paths between the L1 units. The new parallel design for the mesh (MeshX2) network is shown in Figure 3.13. Each logical network in the MeshX2 design use XY routing to avoid deadlock. This have an additional advantage for the MeshX2 design as well. The mesh presented in Figure 3.13(b), is designed for the L2-to-L1 traffic. The L2 caches are placed diagonally at the center of the mesh. So if XY routing is used, the first and last row of the mesh are never used by any messages. Therefore, this allows us to remove these additional links, which reduces both static...
power consumption of the NoC and the number of bisection channels. This design choice can also be applied to the mesh network for L1-to-L2 traffic, leading to the reduction of many unnecessary channels along the Y direction.

The CmeshX2 design uses two parallel Cmesh networks, similar to the design presented in Section 3.2.1.6. However, this design benefits from lower link counts (as mentioned earlier, the concentration channels are now unidirectional), and lower router radices.

Figure 3.14 presents a butterfly logical topology that is used for both directions of traffic. Two identical butterflies are placed on top of each other to direct the network traffic for each direction of the system. Since the number of L2 caches is limited to six units, the design uses two fewer routers than are used in the last fly (stage) of the butterfly. This also reduces the number of bisection channels in each individual butterfly network. More precisely, there are 8 bisection channels in the conventional butterfly, as shown in Figure 3.5. However, here each butterfly network has only 6 bisection channels.

The next step in our design is bisection matching. It is clear that it is beneficial to simply multiply the number of available networks to two, and allow each type of traffic (L1-to-L2, and L2-to-L1) to benefit from its own dedicated network. However, this will become very costly in terms of power since the size of the network is doubled. By using bisection matching, we guarantee to keep the cost of our parallel networks (and their physical constraints in terms of area) relatively similar to the original topologies.

The design parameters, described in Table 3.3, are calculated using the bisection bandwidth criteria for the same network throughput as used in previous designs. For each baseline topology, two variations of the NoCs have been proposed; symmetric and asymmetric.

The symmetric designs, MeshX2-sym, CmeshX2-sym, and ButterflyX2-sym, adopt the
Table 3.3: Bisection bandwidth for symmetric and asymmetric designs for the targeted topologies, butterfly, mesh and Cmesh.

<table>
<thead>
<tr>
<th>Topology</th>
<th>L1-to-L2</th>
<th>L2-to-L1</th>
<th>Bisection Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$bc$</td>
<td>$N_{bc}$</td>
<td>$bc$</td>
</tr>
<tr>
<td>ButterflyX2-sym</td>
<td>22 x 8</td>
<td>6</td>
<td>22 x 8</td>
</tr>
<tr>
<td>ButterflyX2-asym</td>
<td>16 x 8</td>
<td>6</td>
<td>22 x 8</td>
</tr>
<tr>
<td>MeshX2-sym</td>
<td>22 x 8</td>
<td>10</td>
<td>22 x 8</td>
</tr>
<tr>
<td>MeshX2-asym</td>
<td>16 x 8</td>
<td>10</td>
<td>22 x 8</td>
</tr>
<tr>
<td>CmeshX2-sym</td>
<td>32 x 8</td>
<td>4</td>
<td>32 x 8</td>
</tr>
<tr>
<td>CmeshX2-asym</td>
<td>22 x 8</td>
<td>4</td>
<td>32 x 8</td>
</tr>
</tbody>
</table>

same channel width for every channel in both parallel networks, i.e., the L1-to-L2 and L2-to-L1 networks. In the symmetric designs, the total bisection wires are equally divided between the two separate networks. For example, the ButterflyX2-sym has 2112 wires to transfer data across the chip bisection. Therefore, the L1-to-L2 network uses 1056 wires for crossing the bisection, and the remaining 1056 wires are only used by the L2-to-L1 network.

On the other hand, the asymmetric topologies (denoted by “-asym” in their name) employ another modification in their network design. As mentioned earlier, the L1-to-L2 communication latency has minimal effect on the overall system performance. The traffic in the L1-to-L2 direction is also lighter, as shown in Figure 3.12. So we chose to use a lower count of bisection channels (and wires) for the L1-to-L2 traffic to reduce the power consumption of the network. The design is asymmetric since the L2-to-L1 network uses channels with wider widths in comparison to the channels in the L1-to-L2 network. Looking at the previous example, the L1-to-L2 network in the ButterflyX2-asym network has narrower links (16B per channel) in comparison to the L2-to-L1 network (22B per channel). So the total bisection wires are 1824, where 1056 wires result in a more latency sensitive L2-to-L1 network.

3.3.2 Evaluation

Overall we propose and analyze 6 different networks for the GPUs; MeshX2-sym, MeshX2-asym, ButterflyX2-sym, ButterflyX2-asym, CmeshX2-sym, and CmeshX2-asym. All of our proposed designs for mesh and Cmesh (MeshX2-sym, MeshX2-asym, CmeshX2-sym and CmeshX2-asym) use X-Y Routing.

Figure 3.15 presents the performance of the proposed topologies, normalized to the re-
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

(a) Speedup of asymmetric and symmetric ButterflyX2 against the baseline butterfly. (b) Speedup of asymmetric and symmetric MeshX2 against Mesh.

(c) Speedup of asymmetric and symmetric CmeshX2 against Cmesh.

Figure 3.15: Performance comparison of various parallel designs against baseline NoCs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ButterflyX2-sym</th>
<th>ButterflyX2-asym</th>
<th>Butterfly</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIV</td>
<td>1.10</td>
<td>1.09</td>
<td>1.00</td>
</tr>
<tr>
<td>DCT</td>
<td>1.15</td>
<td>1.14</td>
<td>1.05</td>
</tr>
<tr>
<td>FWHTH</td>
<td>1.20</td>
<td>1.19</td>
<td>1.08</td>
</tr>
<tr>
<td>PWT</td>
<td>1.25</td>
<td>1.24</td>
<td>1.17</td>
</tr>
<tr>
<td>HIST</td>
<td>1.30</td>
<td>1.29</td>
<td>1.20</td>
</tr>
<tr>
<td>MT</td>
<td>1.35</td>
<td>1.34</td>
<td>1.25</td>
</tr>
<tr>
<td>RED</td>
<td>1.40</td>
<td>1.39</td>
<td>1.30</td>
</tr>
<tr>
<td>SOBEL</td>
<td>1.45</td>
<td>1.44</td>
<td>1.35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MeshX2-sym</th>
<th>MeshX2-asym</th>
<th>Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIV</td>
<td>1.10</td>
<td>1.09</td>
<td>1.00</td>
</tr>
<tr>
<td>DCT</td>
<td>1.15</td>
<td>1.14</td>
<td>1.05</td>
</tr>
<tr>
<td>FWHTH</td>
<td>1.20</td>
<td>1.19</td>
<td>1.08</td>
</tr>
<tr>
<td>PWT</td>
<td>1.25</td>
<td>1.24</td>
<td>1.17</td>
</tr>
<tr>
<td>HIST</td>
<td>1.30</td>
<td>1.29</td>
<td>1.20</td>
</tr>
<tr>
<td>MT</td>
<td>1.35</td>
<td>1.34</td>
<td>1.25</td>
</tr>
<tr>
<td>RED</td>
<td>1.40</td>
<td>1.39</td>
<td>1.30</td>
</tr>
<tr>
<td>SOBEL</td>
<td>1.45</td>
<td>1.44</td>
<td>1.35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CmeshX2-sym</th>
<th>CmeshX2-asym</th>
<th>Cmesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIV</td>
<td>1.10</td>
<td>1.09</td>
<td>1.00</td>
</tr>
<tr>
<td>DCT</td>
<td>1.15</td>
<td>1.14</td>
<td>1.05</td>
</tr>
<tr>
<td>FWHTH</td>
<td>1.20</td>
<td>1.19</td>
<td>1.08</td>
</tr>
<tr>
<td>PWT</td>
<td>1.25</td>
<td>1.24</td>
<td>1.17</td>
</tr>
<tr>
<td>HIST</td>
<td>1.30</td>
<td>1.29</td>
<td>1.20</td>
</tr>
<tr>
<td>MT</td>
<td>1.35</td>
<td>1.34</td>
<td>1.25</td>
</tr>
<tr>
<td>RED</td>
<td>1.40</td>
<td>1.39</td>
<td>1.30</td>
</tr>
<tr>
<td>SOBEL</td>
<td>1.45</td>
<td>1.44</td>
<td>1.35</td>
</tr>
</tbody>
</table>

We present the speedup achieved by CmeshX2-sym and CmeshX2-asym as compared to the baseline Cmesh network in Figure 3.15(c). In both of our proposed designs, the performance is almost equal to the baseline Cmesh network (99% and 97% of baseline performance), while
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

the channel width is half of the channel width of the baseline Cmesh. The main reasons why we can obtain similar performance is due to a reduction in arbitration latency (CmeshX2 networks have switches with lower radices, since the connection between concentration switch and injecting nodes are unidirectional), and routing the L1-to-L2 and L2-to-L1 traffic on two different physical networks.

![Figure 3.16: Dynamic and static power breakdown for various parallel symmetric and asymmetric designs.](image)

Figure 3.16 shows a comparison of conventional, symmetric and asymmetric NoCs, in terms of power consumption. The ButterflyX2-sym and ButterflyX2-asym designs do not exhibit a significant power reduction in comparison to the baseline butterfly network (18% and 25% power reduction, on average, for the symmetric and the asymmetric design, respectively) because the reduction in power, due to reduction in channel width, has been mostly compensated by the increase in the number of low-radix routers.
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

(a) EDP comparison of ButterflyX2-sym and ButterflyX2-asym against the baseline butterfly. (b) EDP comparison of proposed MeshX2-sym and MeshX2-asym against baseline Mesh.

(c) EDP comparison of proposed CmeshX2-sym and CmeshX2-asym against baseline Cmesh.

Figure 3.17: Design comparison based on the Energy-Delay Product metric.

The MeshX2-sym and MeshX2-asym designs dissipate 35% and 44% less static power than the baseline mesh, respectively. This is due to the narrower bandwidth and lower channel counts of our MeshX2 designs, versus the higher component count of the mesh. While the router count is doubled in the design, most of these routers have low radices. The total power savings achieved by the MeshX2-sym and MeshX2-asym design are 30% and 37% on average in comparison to baseline mesh, respectively.

The largest power savings are observed when using parallel asymmetric CmeshX2 (CmeshX2-asym) and symmetric CmeshX2 (CmeshX2-sym). The CmeshX2-asym dissipates 69% less power, in comparison to the baseline concentrated mesh, while the CmeshX2-asym dissipates 65% lower power. This significant reduction in power is directly related to the narrowing of the channel width.

The EDP for each application is calculated based on the execution time of the application and energy consumed by this NoC during execution. Even though the proposed NoC designs have slightly lower performance on average in comparison to the baseline designs, they exhibit larger power savings that results in lower EDP than the baseline topologies. As shown in Figure 3.17(a), the ButterflyX2-asym has 7% lower EDP on average (lowest EDP among the butterfly designs) as compared to baseline butterfly network. In Figure 3.17(b), the EDP of the proposed MeshX2 designs
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

Figure 3.18: Overall comparison of baseline and proposed designs using the EDP metric.

(MeshX2-sym and MeshX2-asym) is compared with the EDP for the baseline mesh network. We see significant power savings with the MeshX2-asym, while achieving comparable performance to the baseline mesh. This leads to a 72% reduction in EDP, as compared against the baseline. The significant power savings and comparable performance provided by the CmeshX2-asym versus the baseline Cmesh results in an 88% reduction in EDP (see Figure 3.17(c)).

Figure 3.18 compares the range of topologies proposed in this Chapter in terms of EDP. The EDP values are normalized to the topology with the highest EDP, i.e., Cmesh. Cmesh also had the highest performance among all topologies that we evaluated in Section 3.2.3. As can be seen, our proposed CmeshX2-asym design has the lowest EDP among all the topologies (88% reduction against Cmesh), while it provides comparable performance as Cmesh (97%). MeshX2-asym designs are a close second, with a 86% reduction in EDP. High performance and low power consumption in CmeshX2-asym and MeshX2-asym designs make them suitable options for GPUs.

3.4 Summary

In this chapter, we evaluated a number of network-on-chip designs, comparing power and performance metrics, targeting the memory subsystem of a contemporary state-of-the-art GPU architecture. We first analyzed the memory access patterns of GPU applications, and used this to motivate the design of a range of NoCs that were specifically tailored for GPUs. We arrived at an asymmetric NoC design that uses a logical topology with two different set of channel widths for different traffic directions, i.e., L1-to-L2 and L2-to-L1. This strategy reduces contention in the NoC, and in turn, and allows finer-grained optimization of the topology, including reduction in the router radices, and number of required channels. We compared various asymmetric NoC designs based on performance, power and EDP metrics. Our analysis shows that CmeshX2-asym provides
CHAPTER 3. ASYMMETRIC NETWORK-ON-CHIP FOR EXISTING GPU ARCHITECTURES

comparable performance to the best baseline design, Cmesh, but consumes 65% less power. The MeshX2-asym topology also consumes 37% less power than the baseline mesh while providing comparable performance to this topology. Based on this evaluation, we conclude that the CmeshX2-asym and MeshX2-asym are the most suitable electrical NoC designs for GPU systems.

This chapter has focused on the design of an energy-efficient electrical network. In the next chapter, we will use our knowledge gained in this chapter, and the use of new technologies to design NoCs that can scale to GPU chips with many more compute units.

\[\text{\textsuperscript{2}}\]

\[\text{An earlier version of this work was presented in a paper by the author [112].}\]
Chapter 4

Silicon-Photonic NoCs for Existing and Scalable GPUs

Up to this point, we have extensively studied the design space of different electrical Network-on-Chip topologies for a contemporary GPU architecture. In this chapter, we examine the implementation of photonic interconnects which leverage from silicon-photonic link technology for allowing high-bandwidth low-latency communication between memory units of the contemporary GPU architecture. We tailor a silicon-photonic interconnect best suited for the GPU architecture, and extend this design to scalable GPU systems that have more number of compute units.

4.1 Related Work

A large amount of prior work has been done in the area of NoC design for manycore architectures, with the goal of providing energy-efficient on-chip communication. We discussed both commercial-available and proposed electrical NoC designs for CMPs in Chapter 3.

On the photonic NoC front, there are no commercial implementations available, but researchers have studied a wide spectrum of silicon-photonic network topologies. The low-radix high-diameter topologies such as mesh and torus are implemented and examined in prior work [52, 113, 114, 52, 49, 115, 116]. The optical network presented by Schacham et. al. [114] is a 2D mesh of routers, forming a folded torus topology that uses on-chip optical switches. The main responsibility of the switches is to redirect data from an input port to the output port based on the routing information. This is done using a network consisting of a 2D grid of optical waveguides, with optical resonators at intersecting points to perform turns in the physical network. However,
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

an electrical sub-network sets up the switches in advance before the data transmission. Once the path is set, communication proceeds between source and destination. The same sub-network resets the network switches after the transmission is done. Multiple issues with this design limit its effectiveness. First, the setup needs to be done in advance, leading to a high latency of the electrical setup. Second, the small number of waveguides in each channel limits the number of cross-points. This is while the network must transmit a large amount of data to amortize the electrical setup cost, which is not typical in a cache hierarchy. In the common case, the largest data transmitted between a source and a destination is usually a cache line (typically 64 Bytes).

Phastlane [52] is another example of a low-radix high-diameter topology that uses the optical crossbar switches. It is a 2D on-chip grid of optical crossbar switches. Each Phastlane router in this 2D mesh is located on a separate chip, which is integrated into a 3D structure with the processor die. Routers are using simple optical-level control signals which allow packets to traverse several hops in the network in the absence of contention. This is similar to the way that virtual express channels enable bypassing of packets in an electrical network [78]. However, if we encounter contention in the network, packets need to be stored in router buffers, which means they the data needs to change to the electrical domain and then buffered in the router. The packets are dropped and retransmitted if the buffer is full. Unfortunately, this can significantly increase the worst case power consumption in architectures that execute memory-intensive and irregular applications [117].

Kirman and Martinez [49] proposed an all-optical torus-like network for manycore architectures. The design of an all-optical network was achieved by employing bold decisions, such as wavelength-based routing, using an invariant wavelength between source and destination pairs, and using passive optical routers. The routing pattern of the optical routers are set during the design time, and each route between the source and the destination employs a wavelength that is invariant, and does not depend on ongoing transmissions of other nodes (not affected by the routing/arbitration), which simplifies design and operation. The use of an invariant wavelength (wavelength-based routing) also means that only the transmitting data needs to be transferred in the packets. This reduces design complexity and power efficiency, but the design is not scalable nor extensible (the wavelength allocation and routing patterns are all set at design time). For larger scalable architectures, the entire design has to be redesigned to consider the needs of the new architecture. The crossbar switch used in this design has a fixed $5 \times 5$ radices. The work by Sherwood et. al. [118] presents an early design for optical silicon non-blocking routers with $4 \times 4$ radices, allowing the routing of four optical inputs to four outputs with individual bandwidths.
THOE \cite{116} is a torus-based hierarchical optical-electronic Network-on-Chip for CMPs. THOE takes advantage of both electrical and optical routers and interconnects in a hierarchical manner. Every group of four processors form a cluster through an electronic switching fabric, and all the clusters are interconnected by an unfolded or folded torus network through optical switching fabrics and optical waveguides. Processors in the same cluster share a hierarchical router that includes the local electronic switching fabric, optical switching fabric, and a control unit. This work is similar to the work by Kirman \cite{49}, but is scalable due to use of a hierarchical network design.

Iris \cite{113} is a multilayer design, consisting of a low-latency broadcast/multicast nanophotonic subnetwork and a throughput-optimized circuit-switched subnetwork, in form of a mesh topology, that employs photonic channel waveguides. In Iris, the broadcast subnetwork handles short, multicast-like coherence protocol messages, while the circuit-switched subnetwork transfers large packets. FONoC \cite{115} is a Fat-tree based Optical NoC for application-specific manycore SoCs. FONoC develops a novel combination of topology (Fat-tree), floorplan, and the coherency protocol in its design. It carries both payload data (using circuit switching) and network control data (using packet switching) on the same optical network. It leverages a unique protocol to minimize network control data, while reducing power consumption. It also utilizes novel optical turnaround routers that route a packet upstream until it reaches the common ancestor node of the source and destination in the fat-tree, and then, routes it downstream to the destination.

A number of topical topology features, such as optical logic gates, storage (optical buffers), and the optical router design are far from mature. This is the main reason that routing and arbitration control is usually implemented in the electrical domain. As a result, many proposed optical interconnect architectures are bus-based \cite{119, 50, 120}, use hybrid NoCs that are medium-radix medium-diameter topologies \cite{48, 51}, or are multilayer topologies \cite{121, 122, 123}. The proposed designs in these also consider employing a hybrid bus-based topology, implementing hybrid bus-based crossbars, and hybrid concentrated hybrid crossbars.

The multibus architecture presented by Kirman et al. \cite{119} comprises an optical ring that assigns unique wavelengths per node in order to implement a multibus. Every bus cycle, the contents of the buses are optically received, converted to electrical signals, and then handled by logic in the electrical domain. The busses interconnect multiple L2 caches and a single memory controller. Corona \cite{50} is a crossbar design with several MWSR buses. This architecture associates one wavelength to each optical router, and the routers are not required to route the optical data streams based on the modulated wavelength. Using this bus-based architecture, various nodes are simply connected through multiple optical buses and the modulated wavelength of the optical
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUs

stream is used to identify the destination node, providing simplicity in the design. ATAC [120] is a bus-based optical broadcast network that presents a new coherence protocol, ACKwise, in order to improve multicore programming. ATAC interconnects 64 optically-connected clusters, using a ring topology. Groups of 16 nodes within the cluster have access to the optical hub through two separate meshes (one for sending messages and one for receiving messages). This hub performs a different role than the concentration routers in an electrical network.

For hybrid networks, Firefly [51] is a network with multiple, small, crossbars. To avoid the overhead of global arbitration, localized, electrical, arbitration is performed between a small number of ports in each crossbar. The Firefly topology uses Single-Write, Multiple-Read (SWMR) optical buses, assisted with broadcast communication, for path reservations and channel arbitration. Joshi et. al. [48] explore the use of silicon-photonic links to implement non-blocking crossbar and Clos networks. Two different implementations of an 8-ary 3-stage Clos are proposed in this thesis. In one implementation, we consider using two photonic point-to-point channels to interconnect the first and last stages to the intermediate set. Alternatively, we consider a modified Clos that allows the photonic channel to act as an intermediate set of routers.

The multilayer optical network [121] provides direct optical connectivity between nodes of different clusters in a tiled manycore architecture. Another multilayer design is presented by Xiang et. al. [123] that divides the tiles similarly into four clusters. It consists of 16 decomposed optical crossbar slices mapped on four optical layers. Each slice is a 16 × 16 optical crossbar, interconnecting all tiles from one cluster to another (for inter-cluster communication), or all tiles from the same cluster (for intra-cluster communication). Koohi et. al. [117] present a design with numerous MWSR buses routed in a snake pattern among the nodes of a similar 64-core tiled CMP. Udi pi et. al. [122] show how to combine memory and memory controllers with photonic networks through 3D integration. In this design the 3D-stacked memories contain a photonic interface on the die. Processors can write to the stacks using SWMR buses, or read the data through a daisy-chained ring.

The area of silicon-photonic NoCs for GPU and heterogeneous systems is not widely explored. Goswami et al. [124, 125] explore a 3D-stacked GPU micro-architecture that uses an optical on-chip crossbar to connect shader cores and memory controllers in the GPU memory hierarchy. The main difference between the work in this thesis and this quality prior work is that we present our own tailored monolithically-integrated photonic NoCs for communication between L1 caches and L2 caches and evaluate them against different electrical designs for current and future scaled-up GPUs. Van Winkle et. al. [126] builds upon our work [127] to propose an optical crossbar-based
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

4.1 Dual-Ring Network-on-Chip Architecture

A dual-ring Network-on-Chip architecture that interconnects the last level caches with both CPU and GPU cores to reduce network latency. This provides an energy-efficient optical Network-on-Chip architecture for Heterogeneous multicores.

Figure 4.1: SWMR and MWSR busses as building blocks for photonic crossbar topology.

4.2 Silicon-Photonic Based Crossbar for Current GPUs

The link-length-independent, data-dependent, energy of silicon-photonic links makes this technology more appropriate for designing low-diameter high-radix topologies, such as a crossbar, Clos or a flattened butterfly network. Here, we consider a crossbar given it’s strictly non-blocking connectivity, and it’s programmability. A photonic crossbar provides lower latency as compared to an electrical crossbar.

4.2.1 Design Considerations for Silicon-Photonic Crossbar

The low-diameter high-radix crossbar topology supports strict non-blocking connectivity. The crossbar network topology can be implemented as a multi-bus topology, where each node in
the network has a dedicated bus. One example of such a design was presented in Section 2. An alternative crossbar design to a MWSR bus-based design (used for both crossbar, and C-crossbar in Chapter 3), is using the SWMR bus-based crossbar. A simplified design of a photonic MWSR and a photonic SWMR are presented in Figure 4.1. As shown in the figure, each L1 and L2 cache unit in the GPU system acts as a node in the network. For example, in the design of the SWMR crossbar NoC in the 32-CU GPU, there are 40 dedicated SWMR buses (32 for vector caches and 8 for scalar caches) required for the L1 units, and 6 dedicated SWMR buses for the L2 units to send information to all the other units, totalling 46 SWMR bus.

In the SWMR crossbar, each transmitting node has a designated channel to transfer messages. All receiving nodes are able to listen to all sending channels. All nodes decode the header of the control messages on the channel and decide whether to receive the rest of the message or not. The destination node is the only node that receives the cache line that directly follows the control message. A node receives a message if it is the destination of that message, and has enough credit. The credit information of the buffer in the receiving nodes piggybacks in the reverse direction with acknowledgments control messages. The message is received and buffered in the destination node’s input buffer. Our photonic SWMR implementation avoids the need for arbitration. At every transmission, the source node sends the control message, and its head flit is decoded by all the receiving nodes. All nodes except the destination node turn off their receiver ring detector after the decode process [51].

In the MWSR crossbar, each transmitting node needs to acquire access to the receiver’s dedicated bus to transmit a message. An access is granted in a Round Robin (RR) fashion between nodes that have a message to transmit. The channels in both designs utilize credit-based flow control for buffer management. Our photonic MWSR implementation uses an optical token channel with fast-forwarding, which provides the same RR arbitration as the electrical crossbars [50][128]. Fast-forward token channels are coupled with credit-based flow control to provide fairness in arbitration. When a node requires to send a message, it turns on its detector for a waveguide, dedicated to arbitration (Arbitration WG). The only reader of the MWSR bus transmits its credit token (if buffer space is available) on the same waveguide. The reader and the current winner of the arbitration communicate through Fast-Forward (FF) tokens in case the flow of the arbitration stops. The FF token is able to bypass previously served senders. At some point along the waveguide the FF token changes into a arbitration token and the arbitration function continues. Figure 4.2 shows a detailed presentation of the fast-forward arbitration between nodes in a MWSR crossbar [128]. A drawback of this approach is the lack of an accurate credit count during arbitration. If buffer space opens up
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

(a) "Sender Node 0" injects a message to the Data WG (not shown) and injects the arbitration token (which contains the information on the number of credits on the buffer of the receiver node) to the arbitration channel. The "Sender Node 1" does not have any message to send, so it has its sender and receiver off. "Sender Node 2" receives the arbitration token, and if there are credits left, sends its data on the Data WG.

(b) If there are no credits left, the arbitration has to stop, since if it continues, the "Sender Node 2" will lose its chance (no fairness). In case there are no credits left in the arbitration token, "Sender Node 2" injects a fast-forward token in the FF WG, and turns its receiver on.

(c) When credit becomes available again, the receiver node injects its token onto the Fast-Forward WG. The "Sender Node 2" is expecting the token, so it removes the token, sends its data on the data WG (not shown), reduces the credit, and continues the normal function.

Figure 4.2: Use of Fast-Forwarding for arbitration in photonic links.
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUs

Table 4.1: Energy Projections for Photonic links based on prior work [4, 5, 6]. – Tx = Modulator driver circuits, Rx = Receiver circuits, Dynamic energy = Data-traffic dependent energy, Fixed energy = clock and leakage We consider 20% (conservative projection) and 30% (aggressive projection) laser efficiency.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Data-dependent energy (fJ/bt)</th>
<th>Fixed energy (fJ/bt)</th>
<th>Thermal tuning (°K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tx</td>
<td>20</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Rx</td>
<td>20</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

in the receiver node, the credit in the arbitration token is not updated, until it reaches zero.

4.2.1.1 Integration

For our silicon-photonic NoCs, we consider monolithically-integrated links [4, 5, 6]. Monolithic integration of silicon-photonic links use the existing layers in the process technology to design photonic devices. This tends to be a better choice than 3D integration, because the link transmitter circuits experience lower parasitics, which leads to lower energy consumption in the link. Working prototypes of links designed using monolithic integration have been presented in previous work [4, 5, 6].

One major obstacle is the amount of crosstalk between adjacent optical waveguides, which is the sum of light in one waveguide coupled from neighboring waveguides. Crosstalk can become profound when the distance between adjacent waveguides is smaller than the wavelength λ of the light propagating through them. Joshi et al. [48] performed analysis of various photonic technologies and integration approaches, and suggested the use of 4THz Free-Spectral Range (FSR). The FSR is the spacing in the optical frequency spectrum between two successive reflected or transmitted optical intensities. By using 4THz FSR and double-ring filters, up to 128 wavelengths, modulated at 10 Gb/s, can be placed on each waveguide. 64 wavelengths are available in each direction. The wavelengths are interleaved to alleviate crosstalk and filter roll-off requirements. A non-linearity limit of 30 mW at 1 dB loss is assumed for the waveguides, based on the analysis in prior work [48].

We next evaluate our link design [48]. For our analysis, we project the E-O-E conversion cost, thermal tuning, and photonic device losses for this next generation link using the measurements results reported in previous work [4, 5, 6]. The waveguides are single mode, and have a pitch of 4 µm to minimize the crosstalk between neighboring waveguides. The modulator ring and filter ring...
Table 4.2: Projected/Measured Optical Loss per Component [4][5][6]. We consider -17 dBm (a conservative projection) and -20 dBm (an aggressive projection) for the photodetector sensitivity.

<table>
<thead>
<tr>
<th>Device</th>
<th>Loss (dB)</th>
<th>Device</th>
<th>Loss (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optical Fiber (per cm)</td>
<td>5e-6</td>
<td>Coupler</td>
<td>1</td>
</tr>
<tr>
<td>Non-linearity (at 30 mW)</td>
<td>1</td>
<td>Splitter</td>
<td>0.2</td>
</tr>
<tr>
<td>Modulator Insertion</td>
<td>1</td>
<td>Filter through</td>
<td>1e-3</td>
</tr>
<tr>
<td>Waveguide crossing</td>
<td>0.05</td>
<td>Filter drop</td>
<td>1.5</td>
</tr>
<tr>
<td>Waveguide (per cm)</td>
<td>2</td>
<td>Photodetector</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Figure 4.3: Physical layout of the silicon-photonic crossbar NoCs for the target GPU chips.

4.2.1.2 Physical Layout

Based on our discussion at the beginning of this Chapter, we can design a photonic crossbar topology with either SWMR buses or MWSR buses. Our crossbar topology is mapped onto the physical layout of our target GPU (an AMD Radeon 7970) using a “U-shaped” layout, as illustrated in Figure 4.3. As shown in the figure, the photonic crossbar is powered by an off-chip laser source.
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

Table 4.3: The workloads selected from the AMD APP SDK.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>It finds the position of a given element in a sorted array of size 1048576.</td>
</tr>
<tr>
<td>CONV</td>
<td>Convolution filtering on each element of an input matrix of 4096 × 4096 with blur mask of 5 × 5</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transforms on an input matrix of size 8192 × 8192</td>
</tr>
<tr>
<td>DWTHAAR</td>
<td>One-dimensional Haar wavelet transform on a one-dimensional matrix of size 8388608</td>
</tr>
<tr>
<td>LARGSCAN-LS</td>
<td>Performs scan on a large default array of 134217728 elements</td>
</tr>
<tr>
<td>MTWIST</td>
<td>SIMD-oriented Fast Mersenne Twister (SFMT) generates 4194304 random numbers and uses Box-Muller to convert these numbers to Gaussian random numbers.</td>
</tr>
<tr>
<td>RED</td>
<td>Performs reduction by dividing an array of 33554432 into blocks, calculating the sum of blocks and calculating the sum of block sums.</td>
</tr>
<tr>
<td>RG</td>
<td>Performs recursive Gaussian filter on an image of size 1536 × 1536.</td>
</tr>
<tr>
<td>SOBEL</td>
<td>Performs Sobel Edge detection algorithm on an image of size 1536 × 1536.</td>
</tr>
<tr>
<td>URNG</td>
<td>Generates uniform noise on an input image of size 1536 × 1536.</td>
</tr>
</tbody>
</table>

The target GPU is designed in 28 nm technology. Monolithic integration of a photonic crossbar would slightly increase the area of the GPU chip. We use 4 μm-pitch waveguides and 10 μm-diameter rings. The total floorplan area of the GPU chips with the photonic NoCs, is 390.8 mm² (vs. the target chip’s floorplan areas of 350.0 mm²). Alternatively, this area can be used to increase the number of the compute units by two in the system. Each compute unit is considered to be 16 mm² in 28 nm technology. In the best case scenario, assuming ideal scalability, this increase in the number of compute units can increase the performance of the system by 6%. Later in this section, we will present how this increase in the performance is not comparable with the benefits that can be achieved by using a silicon-photonic NoC.

4.2.2 Comparison of Electrical vs. Photonic Crossbars

The applications evaluated in this work are taken from the AMD APP SDK [19]. For each application, we can change program inputs to control the workload intensity. We chose large input sizes for our benchmarks to show how current GPUs are unable to support the growth in data set sizes we expect to see in future workloads. We made sure that the selected subset of applications include a diversity of workload features and varying intensities for creating a range of traffic behaviors. Table 4.3 lists the set of applications selected from this benchmark suite, and includes a brief description for each application.

The power for the electrical network is estimated using a detailed transistor-level circuit model, as described in Chapter 3.2.3. For the 128-CU GPU, the wires in the crossbar are designed in the global metal layers using pipelining and repeater insertion in 14 nm technology [129]. We

81
use the photonic technology power and loss estimation described in Sections 4.2.1.1. Our metric of choice for performance and energy comparison is energy-delay\(^2\) product or \(ED^2P\). The choice of only energy \((E)\) gives an advantage to systems that stress energy efficiency over performance. The choice of \(EDP\) favors systems that value both performance and energy efficiency, equally. \(ED^2P\) favors high performance systems whose design allocates a large expenditure of energy. In our design we favor performance over energy, so we use \(ED^2P\) merely as a metric that stresses performance over energy savings. Nonetheless, in all our evaluations, we also present the results for both performance and power consumption.

Table 4.4 describes the number of global buses for each crossbar topology, either SWMR crossbar, or MWSR. It also outlines the main photonic elements required for this implementation. The number of waveguides, filters, and modulators are calculated, considering channels are 72 bytes wide. The number of required wavelengths per channel to transfer 72 bytes of data can be calculated based on the ratio of the frequency of the electrical components of the network vs. the frequency of the photonic components. In SWMR, one modulator and 40 ring-filters are required for each wavelength. The MWSR requires 40 modulators and 1 ring filter for each wavelength.

Figure 4.4 compares the performance of the applications in a system with silicon-photonic crossbars and electrical crossbars. In our evaluation, we considered MWSR and SWMR electrical buses with channel widths of 16, 32 (divisors of 64 bytes of data), and 72 bytes (packing control signals and the cache line in a single message) for the 32-CU GPU system. For the photonic NoC, due to the bandwidth density advantage of silicon-photonic link technology versus electrical link technology, we considered SWMR and MWSR buses with a channel width of 72 bytes. In Figure 4.4, we compare the electrical and photonic MWSR and SWMR crossbars in terms of performance (top plot) and network traffic (bottom plot). To calculate the performance improvement (i.e., speedup) for each application, we divided the application execution time when using the MWSR electrical crossbar NoC with a 32 byte channel width (E-MWSR-32) by the execution time for the other NoC designs. This means the performance results are normalized to performance of a system

Table 4.4: The number of global buses required for topologies with 32-CU GPUs – DOWN = home node is a L1 unit, and UP = home node is a L2 unit. The channel width is 72 bytes.

<table>
<thead>
<tr>
<th></th>
<th>DOWN</th>
<th>UP</th>
<th>Waveguide</th>
<th>Filters</th>
<th>Modulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWSR</td>
<td>6</td>
<td>40</td>
<td>63</td>
<td>2668</td>
<td>27840</td>
</tr>
<tr>
<td>SWMR</td>
<td>40</td>
<td>6</td>
<td>63</td>
<td>27840</td>
<td>2668</td>
</tr>
</tbody>
</table>

82
Figure 4.4: Speedup and bandwidth for a 32-CU GPU with different crossbars. Ticks on the x-axis follow the pattern $T$-$X$-$N$, where $T$ refers to the type of technology (electrical = E; photonic = P), $X$ is the type of crossbar (MWSR or SWMR), and $N$ refers to the channel width in bytes. For electrical NoCs three different link bandwidths (from left to right 16, 32 and 72) are considered.

With E-MWSR-32 network. As shown in the corresponding plot, the photonic SWMR crossbar (P-SWMR-72) achieves an average $2.60 \times$ speedup, and the photonic MWSR crossbar (P-MWSR-72) achieves an average $2.64 \times$ speedup compared to E-MWSR-32.

A similar performance improvement is observed if we compare P-MWSR-72 and P-SWMR-72 with E-SWMR-72. The magnitude of speedups directly correlate with the network bandwidth required by the applications (see the bottom plot in the Figure 4.4). Applications with a higher number of memory transactions (i.e., larger offered bandwidth) exhibit larger benefits (i.e. $6.3 \times$ for mtwist and $5.2 \times$ for conv). The benefits we observe are a result of the low latency nature of photonic NoCs. As can be seen in the figure, increasing the channel bandwidth of the electrical crossbars results in minor performance improvements in memory-intensive applications (such as conv), since transmission latency of the electrical channel (4 cycles on average) masks any benefit of any reduction in serialization latency through increases in bandwidth. Compute intensive applications (such as urng), and applications with a small number of thread blocks (such as bs), can
store their entire dataset in L1 data caches (generating very few memory accesses to the L2 banks). Therefore, these applications do not benefit from the high bandwidth offered by a photonic network (i.e., the NoC is not utilized).

Figure 4.5 compares the E-MWSR, E-SWMR, P-MWSR and P-SWMR crossbars using NoC power (see top plot) and the \(ED^2P\) metrics (see bottom plot) across the different benchmarks. For all benchmarks, the static power of electrical crossbars is dominant and increases from 2.1 watts to 19 watts when the channel width is increased from 8 to 72 bytes. A 72 byte link requires \(72 \times 8\) wires, and each wire is 46.71 mm long (if the same U-shaped layout as photonic crossbar is used). We have 46 buses in the network. This means 26,496 wires are required. If we assume 4 segments comprise a wire, running at a frequency of 1 GHz, the wire consumes roughly 180 fJ/bit in 28 nm technology. This leads to a 19 watts power dissipation for the NoC, which is too high, so commercial GPUs do not use the electrical networks that have links with 72 byte bandwidth.

The P-MWSR-72 and P-SWMR-72 crossbars consume 29% lower power on average as compared to E-MWSR-72 and E-SWMR-72, but they consume 32% more power on average as
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUs

compared to crossbars with 32 byte channels, E-MWSR-32 and E-SWMR-32. Note that the static power values for both the P-MWSR and P-SWMR crossbars (fixed, thermal and laser components) are very similar. The reason for their similar power dissipation is that both P-MWSR and P-SWMR crossbars utilize the same number of photonic components and waveguides (see Table 4.4). To reduce this large fraction of static power, a NoC layout using concentration can be adopted, which we will consider later in this thesis when designing photonic NoCs for large-scale GPUs.

The bottom plot of Figure 4.5 shows that the photonic NoCs achieve the lowest $ED^2P$ for all the benchmarks, except urng (lower is better). The urng application is insensitive to NoC channel bandwidths, so the electrical crossbar NoC with the narrowest channel width reports the best $ED^2P$ as its power is lower than its electrical counterpart, and its performance equals the performance of the crossbar with the largest bus bandwidth. Note that the $ED^2P$ is significantly improved when considering our photonic NoCs. The average reduction across all benchmarks in terms of $ED^2P$ is 66.5% for P-SWMR-72 and 67.7% for P-MWSR-72, when compared to E-MWSR-32).

4.3 Tailoring a Hybrid Photonic NoC

As shown in the previous section, there are considerable drawbacks in terms of power dissipation with both the SWMR and MWSR crossbars, due to the large number of global channels. To address this problem, we design a NoC with a significantly lower number of channels, by taking advantage of the asymmetry in the L1-to-L2 versus L2-to-L1 network traffic, and by analyzing the communication between L1 units, as discussed previously in Chapter 3.3.

We propose to use SWMR buses for L2-to-L1 communication and MWSR buses for L1-to-L2 communication for our hybrid design. The choice of a SWMR bus in the L2-to-L1 network is easy to justify. The data requested from the L2 cache units and sent to the L1 cache units is latency sensitive since the data is needed for the work-items to start their execution. In SWMR, there is no arbitration latency in the transmitting side (L1 cache). Additionally, the number of buses is low, equal to the number of L2 caches.

Using the MWSR buses in the L1-to-L2 direction requires fewer buses (equal to the number of L2 caches), at the expense of introducing additional arbitration latency – hardware must deal with the worst case where all L1 caches attempt to access the same bus simultaneously. However, this has minimal impact on the overall performance because the latency of L1-to-L2 traffic is typically not critical. Further, the GPU programming model enforces that no other CU should be stalled
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

Figure 4.6: The breakdown of the ratio of different types of data that traverse the network in the L1-to-L2 direction. More than 80% of the bytes in the L1-to-L2 traffic are due to small 8-byte control messages.

Figure 4.7: Partial design of a hybrid NoC for the GPU architecture.

awaiting completion of a write-back transfer. At the same time, GPU workloads exhibit light traffic for L1-to-L2 accesses (as shown in Figure 3.12 in previous Chapter).

As shown in Figure 4.6, our analysis of the GPU applications in the AMD APP SDK benchmark reveals that, on average, 80.5% of the traffic generated from L1-to-L2 is due to control messages (8-byte messages), such as read requests and invalidations. This means any bus with a bandwidth wider than 8 bytes would have a 0-cycle serialization latency for 80% of the messages from L1 to L2. Thus, scaling down the L1-to-L2 network causes no noticeable increase in contention.

A crossbar topology provides point-to-point connections between all the L1 and L2 units. Therefore, every L1-to-L1 communication has a dedicated path. As mentioned in Chapter 3.3, the GPU applications studied contain a very small amount of L1-to-L1 communication. So in our hybrid design, in order to reduce the power consumption of the NoC, we also removed the dedicated connections between L1 units. So any L1-to-L1 transaction will be replaced with two transactions: one from L1-to-L2 and a second one from L2-to-L1. By making this change, we would also reduce
any latency that might be introduced by arbitration due to the small amount of communication between L1 units. Overall, the hybrid design reduces the number of buses in the crossbar for the 32-CU GPU from 46 to 12. Figure 4.7 shows a portion of the design of the hybrid NoC. This NoC design can benefit from both photonic, and electrical technologies. Table 4.5 presents the required buses and photonic components in order to design a hybrid photonic NoC.

We performed a comparison between a silicon-photonic hybrid NoC, the electrical hybrid NoC, and the electrical MWSR crossbar. Figure 4.8 presents the results of the electrical and photonic implementations of our proposed hybrid design, in terms of performance and throughput. Figure 4.9 illustrates the power and $E D^2 P$ results for different implementations of the hybrid NoC, i.e., photonic and electrical. We next discuss the results presented.

Table 4.5: Number of global buses required for the hybrid topology support 32-CU GPUs – DOWN = home node is a L1 unit, and UP = home node is a L2 unit. The channel width is 72 bytes.

<table>
<thead>
<tr>
<th></th>
<th>DOWN</th>
<th>UP</th>
<th>Waveguide</th>
<th>Filters</th>
<th>Modulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid</td>
<td>6</td>
<td>6</td>
<td>26</td>
<td>14268</td>
<td>14268</td>
</tr>
</tbody>
</table>
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUs

Figure 4.9: Breakdown of the total power and energy-delay² product of the hybrid NoCs in a 32-CU GPU. Captions in x-axis follows the pattern $T-N$, where $T$ refers to type of technology (electrical = E; photonic = P) and $N$ refers to channel width in bytes (8, 16, 32 and 72). ED²P is normalized to E-MWSR-32 (not shown for clarity).

4.3.1 Hybrid NoC vs. MWSR/SWMR Crossbars

4.3.1.1 Electrical Hybrid vs. Electrical Crossbars

By comparing Figures 4.8 and 4.4 (results in both Figures are normalized to E-MWSR-32), we can observe that our electrical implementation of the hybrid design, E-HYB-72, does not show significant performance improvements over its crossbar counterpart E-MWSR-72. A slight performance improvement is observed in comparison to the E-MWSR-32 (8% on average). But this implementation reduces the power consumption of the electrical crossbar on average by 68% (due to less hardware), while providing the same performance (can be deduced by comparing Figures 4.9 and fig:photonic:crossbar-power). Power savings is the main benefit of the electrical implementation of the hybrid design.

4.3.1.2 Photonic Hybrid vs. Electrical Crossbars

We can see, by comparing Figures 4.8 and 4.4 that our photonic hybrid design (P-HYB-72) provides higher speedup than E-MWSR-72 ($2.7 \times$). As we have shown before in Chapter 4.2.2, increasing the bandwidth of the electrical crossbar did not affect the GPU’s performance due to high electrical link latency.
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUS

The P-HYB-72 consumes 51% less power than both E-MWSR-72 and E-SWMR-72. It has just a marginally higher power dissipation (2%) than E-MWSR-32 and E-SWMR-32. This makes our hybrid design a very good contender compared to any electrical crossbar design.

4.3.1.3 Photonic Hybrid vs. Photonic Crossbars

The P-HYB-72 achieves higher performance than P-MWSR-72 (1.05×) and P-SWMR-72 (1.03×) on average, by slightly reducing the latency (separating the traffic) and achieves 32% and 33% lower power dissipation on average, by using less hardware resources. Therefore P-HYB-72 achieves a 34% and a 30% reduction in the ED²P in comparison to P-MWSR-72 and P-SWMR-72, respectively.

4.3.1.4 Photonic Hybrid vs. Electrical Hybrid

The comparison between our photonic hybrid design and our electrical hybrid design (E-HYB) (Figure 4.8) in terms of performance (top plot) and network bandwidth (bottom plot) reveals the clear benefits of using low-latency photonic technology. P-HYB-72 exhibits, on average, a 2.5× speedup and offers 0.6 Tbit/s more bandwidth versus E-HYB for the topology with the highest bandwidth (E-HYB-72).

Figure 4.9 compares the electrical hybrid (E-HYB) and photonic hybrid (P-HYB) designs in terms of power (top plot) and the ED²P metric (bottom plot). P-HYB-72 reports 51% higher power than the E-HYB-72 NoC due to higher static power consumption (laser, thermal tuning and fixed power). One solution is to adopt a run-time management mechanism based on the workload running on the compute units. Using run-time management, we can deactivate photonic links when they are not being used [130]. Nonetheless, when analyzing the ED²P metric in the bottom plot of the Figure 4.9, we can see that our P-HYB-72 produces on average 82% lower ED²P as compared to the E-HYB-72 NoC.

4.4 Electrical and Photonic NoCs for scalable GPUs

Now that we have seen the extent our NoC designs can benefit contemporary GPUs, we would like to consider this fabric to support scalable, forward-looking, GPU architectures. But before doing that, we need to define the future for GPUs. We would like to perform a fair comparison
between a good alternative electrical NoC that is suitable for this forward-looking GPU and our photonic hybrid design.

4.4.1 Designing the Scalable GPU Architecture

We consider our forward-looking GPU design will have an increased number of CUs, using the 32-CU Radeon HD 7970 GPU or some similar architecture as a building block. In this new architecture, we quadruple the number of CUs to 128, while keeping the same CU design as in the 32-CU GPU for individual CUs. The 128-CU GPU also quadruples the number of memory components in the HD 7970 GPU chip: 128 L1 vector caches, 32 L1 scalar caches, and 24 shared L2 Caches. We assume that the 128-CU GPU is designed using 14 nm CMOS technology, which results in a reasonable floorplan area of 402.2 mm$^2$.

4.4.1.1 Electrical NoCs design

As the number of CUs in a GPU grows, the number of L1 and L2 units that need to communicate also grows. This means that additional SWMR/MWSR buses are required in the crossbar to support this communication. A larger number of buses, implemented using electrical link technology, translates to large die area and higher power dissipation. Moreover, in a MWSR, the transmitting units need to access to one of the buses in the crossbar via arbitration. Also, if each node holds on to the bus for a large number of cycles (due to serialization and transmission delays), this will result in very long wait times for other nodes. Using a large number of SWMR buses imposes pressure on the receiving cache unit, which has a limited number of ports. Therefore, for GPUs with large CU counts, we propose to use an electrical 2D-mesh network.

We discussed in the previous chapter that a mesh is a low-radix, high-diameter, topology that uses decentralized flow control. We found that the mesh was quite adequate for providing high performance in GPU workloads. An electrical mesh network is easy to design from a hardware perspective due to its short wires and low-radix routers. While a mesh avoids the long arbitration delays present within a crossbar, it has higher zero-load latency, as each packet has to traverse multiple routers (hops) to reach its destination. Instead of using a typical 2D-mesh, which is constructed using radix-5 routers, we leverage from Cmesh (i.e., the concentrated mesh, which was shown as the best electrical NoC in Chapter 3). By using Cmesh, we avoid the main drawback of using a 2D-mesh, which is the high hop count, which can result in longer latencies. In our proposed con-
centrated 2D-mesh, each router has a larger radix (7 × 7 or 8 × 8). Figure 4.10 presents the layout of our proposed GPU chip, using an electrical Cmesh NoC.

In our Cmesh designs for the 128-CU GPU, a single router is connected to the L1 vector and scalar cache units, depending on the location of the router in the layout. Various configurations are possible for placement of L2 units (as discussed in Chapter 3). We chose to connect L2 units to the mesh through the routers placed along the periphery of the mesh to avoid long concentration wires for interconnecting them to the router. This is a different decision from our design presented in the previous chapter, where we chose to have longer concentration wires but place the L2 units diagonally. With this design decision, we also follow a similar layout to the AMD Radeon 7970 for our 128-CU GPUs. In the AMD Radeon 7970 the L2 banks are placed on the periphery of the die [72]. The mesh design is modeled using single-cycle routers (with SA and ST performed in a single cycle [2]), each having room for 8 cache lines in the input buffers and no virtual channels (avoiding VA). The routers use a standard matrix-based crossbar that implements round-robin arbitration. We use an XY routing scheme with credit-based flow control.
4.4.1.2 Photonic Hybrid NoC Design

Similar to our photonic hybrid NoC for 32-CU GPUs, our photonic hybrid design for 128-CU GPU also utilizes SWMR buses for L2-to-L1 and MWSR for L1-to-L2 communications. We will only consider this hybrid design for the photonic NoC for the larger 128-CU GPU. The main reason is the high power dissipation of the MWSR or SWMR crossbars for 128-CU GPU, since they require 184 buses (hybrid only requires 48). The hybrid NoC can be mapped as a serpentine onto the physical layout of the 128-CU GPU device. To further minimize static NoC power, the hybrid NoC utilizes concentration through Access Points (AP). In particular, 4 vector L1 caches and 1 shared L1 cache use the same AP to access their corresponding crossbar’s bus on a time-division multiplexing basis (see Figure 4.11). Access points can be used to attach 5 L1 through a single link to the bus. By adding access points, we can reduce the total number of rings by a factor of 4.78, and reduce the number of waveguides. This technique reduces the NoC power for scaled-up GPU system. Table 4.6 presents the required components for leveraging photonic hybrid NoC in design of the scalable GPU architecture.

The area increase we observe by incorporating a photonic NoC is from 402.2 mm$^2$ to 410.0 mm$^2$ for the 128-CU GPU. Photonic components are not scalable (as compared to CMOS.
transistors) and that is the main reason for this increase in area. The area overhead in the 128-CU GPU is 7.8mm$^2$. This area is equal to the area of approximately 2 CUs (the area for a CU is 4 mm$^2$ in 14 nm technology). Increasing the number of CUs from 128 to 130 results in a very small performance improvement (ideally 1%).

4.4.2 Evaluation

In this section, we consider future GPUs with 128 CUs and evaluate the scalability of electrical and photonic NoC designs. For our power evaluations, we implemented all of inter-router channels in the 2D mesh in semi-global metal layers with standard repeater wires. We use the 14 nm technology node [129] for the 128-CU GPU systems. The power dissipated in the SRAM array and crossbar of the routers is calculated using the methodology described before in this dissertation ([105] and [106]).

For comparison, we consider an electrical mesh design and photonic hybrid design with channel widths of 16, 32 and 72 bytes. Here, all comparison metrics are normalized to an electrical 2D-mesh with 16 byte channel widths (E-MESH-16).

Figure 4.12 compares performance (top plot) and bandwidth (bottom plot) of the E-MESH design with our P-HYB, for 128-CU GPU, assuming the same set of channel widths as the previous evaluation. The performance speedup reported for P-HYB-72 is 82% better than the speedup for E-MESH-72 (with maximum 3.43 $\times$ speedup for mtwist). The E-MESH-72 offers increased bandwidth up to 2.81 Tbit/s ($dct$), whereas P-HYB-72 achieves up to 7.28 Tbit/s bandwidth ($mtwist$).

Figure 4.13 compares E-MESH and P-HYB in terms of power (top plot) and ED$^2$P (bottom plot). P-HYB NoCs generally consume more power than an E-MESH counterpart. This leads to 19% higher ED$^2$P for P-HYB-72, in comparison to E-MESH-72.

We can reduce the bandwidth of the channel to reduce power dissipation of the P-HYB. Reduction in the P-HYB channel bandwidth from 72 bytes to 32 and 16 bytes, reduces this power dissipation by 39% and 60%, respectively. The average speedup observed for all the applications in

<table>
<thead>
<tr>
<th>DOWN</th>
<th>UP</th>
<th>Waveguide</th>
<th>Filters</th>
<th>Modulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid</td>
<td>24</td>
<td>24</td>
<td>186</td>
<td>45936</td>
</tr>
</tbody>
</table>
CHAPTER 4. SILICON-PHOTONIC NOCS FOR EXISTING AND SCALABLE GPUs

Figure 4.12: Speedup and bandwidth of benchmarks running on a 128-CU GPU system, varying the photonic hybrid and electrical mesh NoCs. Captions along the x-axis follow the pattern \( T\cdot N \), where \( T \) refers to type of technology and topology (electrical 2D-mesh = E; photonic hybrid = P) and \( N \) refers to channel width in bytes (16, 32 and 72). Speedup and ED\(^2\)P results are normalized to \( E^{-16} \).

Our study for P-HYB-32 is 43\%, and for P-HYB-16 is 17\%, when compared to E-MESH-72. By reducing the channel width to 32B or 16B, we reduce the power consumption, and therefore, the ED\(^2\)P in our hybrid designs. The ED\(^2\)P for P-HYB-32 and P-HYB-16 is reduced by 3\% (for both) when compared to E-MESH-72. This means both P-HYB-32 and P-HYB-16 are marginally better than the Cmesh.

One important feature of the hybrid design is its effect on memory intensive applications. If applications extensively utilize the memory hierarchy (such as conv, dct, dwthaar, mtwist, largescan and sobel), this will favor the photonic hybrid design. For these memory intensive applications, the P-HYB-16 design enjoys a 26\% average performance speedup, and a 13\% average reduction in ED\(^2\)P, as compared to E-MESH-72.

Since these applications scale well with the number of GPU units, P-HYB-32 and P-HYB-72 provide significant reductions in ED\(^2\)P compared against E-MESH-72 (34\% and 40\%, respectively). The reported speedups for P-HYB-32 and P-HYB-72, running the memory-intensive applications, are on average 1.6\times and 2.2\times when compared to E-MESH-72, respectively. Our results clearly show that for future GPU systems that will execute memory intensive workloads, a photonic hybrid design provides the best ED\(^2\)P solution.
4.5 Summary

In this section, we combined our knowledge of silicon-photonic link technology and GPU architecture to present a GPU-specific photonic hybrid NoC (used for communication between L1 and L2) that is more energy efficient than the electrical NoC. Our proposed hybrid design uses MWSR for L1-to-L2 communication and SWMR for L2-to-L1 communication. Our simulation-based analysis shows that applications that are bandwidth sensitive can take advantage of a photonic hybrid NoC to achieve better performance, while achieving an energy-delay\(^2\) value that is lower than the traditional electrical NoC. In the AMD Southern Islands GPU chip with 32 CUs, our proposed photonic hybrid NoC increases application performance by up to 6\(\times\) (2.7\(\times\) on average), while reducing ED\(^2\)P by up to 99\% (79\% on average).

We also evaluated the scalability of the photonic hybrid NoC for a GPU system with 128 CUs. When the 128-CU GPU system runs memory intensive applications, we can achieve up to 3.43\(\times\) (2.2\(\times\) on average) performance speedup, while reducing ED\(^2\)P by up to 99\% (82\% on average), as compared to an electrical mesh NoC.\(^1\)

\(^1\)An earlier version of this work was presented in a paper by the author.\(^{[127]}\)
Chapter 5

Unified Memory Hierarchy for multi-GPU Systems

As mentioned in the Chapter [1] an alternative way to a single scaled GPU design for addressing the high demand of the GPU applications, is to unite multiple GPU devices together, providing the illusion of a single larger system. Efficient coupling of multiple discrete GPU devices can provide substantial time-saving by providing higher processing throughput and enabling more flexible management of system resources such as memory bandwidth. However this multi-GPU architecture introduces a new challenge in terms of memory management between the various devices (whether CPU or GPU), in comparison to a single GPU system. Among these challenges are 1) providing an efficient cross-device communication and data sharing method, 2) providing coherency between multiple GPU devices and the CPU device, and 3) reducing the cost of synchronization.

Our approach to improving the memory management in CPU-multiGPU systems is to utilize a novel hardware-based approach. Our design establishes a hierarchical structure between the CPU and one or more GPUs, treating the GPU’s main memory banks as cache units. This is very similar to how traditional single-GPU systems execute their programs. In traditional systems with one GPU the host program offloads the compute to that GPU. Similarly, in our system an external runtime library (e.g., SKE [28]) provides the view of a single virtual GPU from many GPUs in the system, and offloads the compute workload to many GPUs. Also as a single GPU has the guarantee that it can find its data in the global memory, our hierarchical design also guarantees that our multiple GPUs can find their required data in this memory hierarchy. The worst case scenario is that the data is in the host memory, the last level of the hierarchy.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

This vision for memory management provides a single logical view of memory between the CPU and GPU devices as well. Our proposed approach, named [UMH], introduces hardware to carry out the actual copy, transparent to the user. This hierarchy is constructed by introducing Stacked Memory Directories (SMDs) for each GPU’s memory bank. The SMD design can redirect accesses to the memory where the data resides, whether the address is in the main memory of the GPU or the host memory. The SMDs should maintain coherency information of the data that is stored in GPU memory.

This hierarchical view avoids redundant transfers by limiting the number of transfers from the CPU to the GPU memory to only those cache lines that are requested by the GPU. The transferred data is cached in GPU memory for the GPU to use in the future. Similarly, we need a more efficient mechanism to give the CPU coherent access to the data that is computed by the GPU. Coherent access is supported by incorporating a Host Memory Directory (HMD) component in each controller in the host memory. The HMD tracks the addresses accessed by the GPUs, and jointly with SMDs, can ensure the coherency of the host memory shared between one or more GPUs and the CPU.

To further support coherency, we should leverage a coherence protocol that provides coherence between a CPU and a GPU that are on the same die, reducing synchronization costs between devices. By using the coherence protocol alongside the Unified Memory Hierarchy, a synchronization operation results only in a transfer of modified data by the GPU devices, while unmodified data is not needlessly copied back again to the CPU.

There are three main concepts that involved in this proposed work: 1) reorganizing the memory of different GPUs and the host device to form a hierarchy, 2) a coherence protocol for managing coherence between CPU and multiple GPUs, and 3) using DRAM memory units as caches. So in the next section we describe the related work to these concepts in the realm of GPU-based systems, as well as prior work on designing multiGPU systems.

5.1 Related Work

5.1.1 MultiGPU systems

In previous work, Kim et. al. [67] designed a memory network with a sliced flattened butterfly topology for the interconnect between multiple CPU cores. Based on this efficient design, they also proposed a Unified Memory Network (UMN) that takes into account the management of
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

memory between GPU devices and the CPU [28]. While the sliced flattened butterfly maintains the connectivity between different DRAMs of the multiple GPU memories, the connection between CPU and the GPUs is maintained through a pass-through path. This design supports a no-copy approach, where every access to the CPU memory goes directly through the network topology to the CPU. While this network reduces the communication cost between devices, as the authors commented themselves, there can be significant bandwidth demands on the CPU memory, making the pass-through path a potential bottleneck. Their design has the additional challenge of relying on both the CPU and the GPU vendors to support the same high-speed link interface for memory. The UMN is not considered a hardware memory management solution, since it leverages the support of existing software solutions provided by the CUDA runtime library and the driver.

Kim’s work [28] is not the only prior work that builds on a software runtime to enhance application execution using multiple GPUs. Multi-GPU based MapReduce (MGMR) [131] is another framework, developed on top of the CUDA runtime. MGMR is software-based and does not require changes to the network infrastructure and memory organization to support multiple devices. Instead, it distributes the computation between GPUs based on the capacity of the GPUs to achieve load balancing and increase performance. MGMR also takes into account the host memory (which can have much higher capacity) as part of the framework to store the large data associated with Big Data applications. One major issue in multi-GPU systems is the impact of atomic operations (see Section 5.2 for more detail). Instead of handling the atomic operation in memory (which leads to serialization of different memory requests), MGMR replaces these atomic operations with parallel alternative operations to achieve better performance.

NMF-mGPU [132] targets non-negative matrix factorization as the application to execute on multi-GPU systems. Instead of accessing the memory of different devices through the conventional methods (zero-access, memcpy, or GPUDirect – See Section 2.5), they use Message Passing Interface (MPI). In their case, the input matrix is distributed across existing devices and each portion is processed as described for a single device. Each GPU has to manage a smaller problem. Another executable file operates multiple devices through MPI. This method also imposes some new overheads. Each device requires a full copy of input matrices, and a collective-communication step is necessary after performing each update in order to keep coherence among all local replicas. Otherwise, during the stage where the computed data is copied back to the CPU device, the resulting data can be overwritten by different GPUs, providing only a portion of the desired output. Application specific frameworks that targets multiple GPU devices are becoming increasingly popular, where each framework is optimized to provide the best performance for the target applica-
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

MAPS-Multi [137], is an automatic multi-GPU partitioning framework that distributes the workload based on the underlying memory access patterns. The framework consists of host- and device-level APIs that allow programs to efficiently run on a variety of GPU and multi-GPU architectures. The framework implements several layers of code optimization, device abstraction, and automatic inference of inter-GPU memory exchanges. MAPS-Multi is based on concepts of partitioning the global address space (which we leverage in memcpy approach for multi-GPU systems – see Section 5.2.1.2). We describe the limitations of this partitioning scheme later in this dissertation.

Similar to SKE, SKMD [138] is another runtime library for executing single kernel on multiple GPU devices. However, it aims to tackle more challenges in the CPU-GPU collaboration than SKE which only statically distribute the workloads. For instance one of the challenges in multi-GPU systems is that the data-parallel kernels with irregular memory access patterns are hard to partition over multiple devices. Another challenge is making partitioning decisions when the GPU devices are not homogeneous. SKMD aims to address these challenges by 1) using a performance prediction model that predicts the execution time of kernels based on offline profile data, 2) a code transformation methodology that distributes data and merges the results of computation on different GPUs in a seamless and efficient manner regardless of the data access pattern, and 3) a runtime system that transparently executes the data-parallel kernels on multiple GPUs.

gScale [139] takes a slightly different approach in addressing the challenges of computation on multiple GPU devices, specially when they in a computing cluster. gScale treats each GPU as a virtual GPU (vGPU), and provides each vGPU instance with a private shadow graphics translation table (GTT) to break the limitation of global memory space. gScale copies vGPU’s private shadow GTT to physical GTT along with the context switch. The private shadow GTT allows vGPUs to share an overlapped range of global memory space. However, it is non-trivial to make the global memory space sharable, because global memory space is both accessible to CPU and GPU. gScale implements a ladder mapping mechanism and a fence memory space pool to let CPU access host physical memory space serving the graphics memory, which bypasses the global graphics memory space.

Finally, Matz et. al. [140] explore multi-GPU programming when the LLVM compiler infrastructure is leveraged. In their approach, they collect information on data dependencies and memory access patterns of the host and kernel using static code analysis. Then they merge the results of analysis performed on the host code and kernel code, to choose an appropriate partitioning
strategy. Finally, they apply the code transformations to implement the chosen partitioning between 
CPU and multiple GPU devices, and insert appropriate function calls into the code to dynamic 
runtime libraries to achieve this. The final product is a tool that allows a user to write a single- 
device program that utilizes an arbitrary number of GPUs, either within one machine boundary or 
distributed at cluster level.

5.1.2 Organizing CPU-GPU memory placements

As mentioned earlier, our work aims to form a memory hierarchy to manage the memory 
between multiple GPU devices, and the host CPU device. Different types of physical organization 
of the memory have been explored before.

If the GPU’s DRAMs are placed at the same level as the CPU memory, a flat non-uniform 
memory access (flat-NUMA) organization can be created to leverage the high bandwidth of the 
GPU memory. Such a configuration should reduce bandwidth contention for CPU memory. Bolotin 
et, al. [141] described this type of memory architecture for CPU devices that had two types of 
memory, stacked-DRAM (HBM or HMC) and external DRAM (DDR4). The logical organization 
of an additional stacked-DRAM memory layer was analyzed in terms of bandwidth efficiency and 
l latency. In this design, the OS manages the flat-NUMA organization. Coherency can only be 
managed by intelligent data migration.

Spafford et, al. [142] analyzed a new approach for creating a NUMA architecture between 
multiple GPUs, using external I/O hubs to connect the GPU devices together, and to the CPU device. 
The integration of multiple GPUs into hubs introduces complex performance phenomena including 
non-uniform memory access and contention for shared system resources. They quantified these 
effects and presents some guidance on programming strategies to maximize performance in multi- 
GPU environments.

Agarwal et, al. [143] extended the NUMA design to Cache-Coherent NUMA (CC- 
NUMA) for CPU-GPU systems, to unlock the bandwidth of the GPU memories. They examined 
the best balance for accesses either through cache-coherence or page migration. Their work examined 
the effect of counter-based metrics to determine when to migrate pages from the CPU to GPU. 
It showed that such counter-based metrics are insufficient for finding an optimal migration policy 
to exploit GPU memory bandwidth. They showed that combining reactive migration with virtual 
addressing through Transfer Lookup buffers, allows for aggressive prefetching of pages to migrate 
between different memories of the NUMA architecture and results in increased GPU performance.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

The need for a page migration policy which is aware of internal components of the CC-NUMA architecture (such as the utilization of the interconnecting medium) will allow the system to outperform either CPU and GPU (when they are used in isolation). However, migrating data between memory units of multiple CPU and GPU devices requires more complex software, and will introduce software-based memory management overhead to the system.

BW-AWARE [61] is a type of page placement strategy for GPUs. This type of page placement is application agnostic, bandwidth-aware placement, aiming to maximize GPU throughput by balancing page placement across the memories based on the aggregate memory bandwidth available in a system. It is build upon developing a compiler based profiling mechanism that provides programmers with information about GPU application data structure access patterns, in combination with simple program-annotated hints about memory placement.

Selectively caching the GPU’s required data in its own memory may eliminate the need of cache coherence, while improving the GPU performance in a NUMA architecture with a CPU and a GPU. Agarwal et. al. [144] designed a system that coalesces the requests from GPU to a remote memory such as CPU’s DRAM, to reduce the inefficiencies of the interconnection network. Data migration from CPU to GPU only occurs for read-only data. They also increase the efficiency of the design by introducing an additional component to the CPU to handle GPU requests with sub-cache line granularity (when only a portion of a cache line is required). This reduces the contention both on the bandwidth of the memory and the bandwidth of the interconnecting medium.

5.1.3 Coherency Protocols

One of the novelties in our design is to allow the memory of the GPU devices to be treated as cache modules, with host memory as the Last Level Cache (LLC) shared between all of these devices. This design bears some similarities with APU devices, where a LLC is shared between the CPU and one GPU. Therefore, many existing protocols that are proposed for APU devices can be also utilized in our design. It is important to note that these protocols can not be used as-is, and require a variety of hardware modifications to work with UMH. While we introduce some of these proposed coherency protocols in this section, we only focus on enhancing the NMOESI [20, 145] protocol (see Section 5.3 for more detail), enabling its integration in a UMH. Past work has proposed novel coherency protocols for heterogeneous CPU-GPU systems.

Heterogeneous System Coherence (HSC) was developed by Power et. al. [34] to mitigate the affect of coherence traffic generated by contemporary coherency protocols that were de-
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

signed for chip multiprocessors. Because of the high-bandwidth nature of heterogeneous systems, using coherence mechanisms created for CPU architectures is difficult. Providing full coherency necessitates that each GPU must deal with a large number of parallel requests. Many applications generate tens of thousands of requests in parallel to avoid stalling the GPU. The number of required resources to handle this many parallel requests is impractical for a real machine. Region coherence \cite{alisafaee2012spatiotemporal,zebchuk2007framework,cantin2005improving} exploits coarse-grained sharing patterns among processors, only requiring a subset of the memory requests to be broadcast to all cores. Given that GPGPU applications employ mostly coarse-grained sharing between the CPUs and GPUs, the concept of region coherence lends itself well to the requirements of hardware coherence in heterogeneous systems.

HSC is a directory-based hardware coherence on heterogeneous CPU-GPU systems that adds region buffers to both CPU and GPU L2 caches to track the regions over which the CPU or GPU currently hold permission. This reduces the required bandwidth by making most of the coherent requests to be perceived as incoherent accesses.

The conventional GPU-only coherence schemes are based on Valid/Invalid (V/I) protocols. They perform full cache invalidations, dirty data flushes, and force-execution on remote processing elements during synchronizations \cite{35}. All of these features are inefficient for many existing GPU workloads. So a Heterogeneous-Race-Free (HRF) \cite{146} model was proposed to associate synchronization accesses with a scope that indicates the level of the memory hierarchy where the synchronization should occur. For example, a synchronization access with a local scope indicates that it synchronizes only the data accessed by the thread blocks within its own CU (which share the L1 cache). The Heterogeneous System Architecture (HSA) Foundation \cite{147} and OpenCL 2.0 \cite{148} have adopted a similar scoped-based synchronization model.

The introduction of scopes is an efficient solution to the problem of fine-grained GPU synchronization, but it comes at the cost of programming complexity. Although HRF is a very well-defined model, it cannot hide the inherent complexity of using scopes. So Sinclair et. al. \cite{35} utilize DeNovo \cite{149}, originally proposed for CPUs, to develop DeNovo+HRF. DeNovo+HRF does not require writer-initiated invalidations or directories, but does obtain ownership for written data. Due to its use of ownership on writes, DeNovo+HRF is able to exploit reuse of written data and synchronization variables across synchronization boundaries.

VIPS \cite{150} (Valid/Invalid - Private/Shared) is a protocol with V/I states, while the private and shared data can be differentiated by a flag (P/S). VIPS uses write-back for private data, and write-through for shared data, as well as delayed write-through through coalescing to reduce
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

write-through overhead. The VIPS is designed for multi-core architectures. So Koukos et. al. [151] introduces VIPS-G. VIPS-G uses VIPS-M protocol [150] (which is VIPS for flat placement of architectures, but uses a page granularity), and VIPS-H [152] (which is VIPS proposed for hierarchical placement of homogeneous cores), in conjunction with each other. These protocols are based on self-invalidation (during synchronization), and both rely on P/S classification of data blocks. This classification is performed by the OS using the page table and the TLBs. Dirty private blocks, update the next level using write-back (upon replacement), while shared blocks use write-through and self-invalidate at synchronization points.

Software Assisted Hardware Coherence (SAHC) [153] is built on the idea that the system software (operating system and runtime) often has semantic knowledge about sharing patterns of data across the CPU and the GPU. This knowledge can be utilized to provide cache coherence across throughput-oriented GPUs and latency-sensitive CPUs in a heterogeneous processor. SAHC proposes a hybrid software-hardware mechanism that judiciously uses hardware coherence only when needed while using software’s knowledge to filter out most of the unnecessary coherence traffic.

5.1.4 DRAM cache for GPUs

Placing a faster DRAM technology (such as stacked-DRAMs) to act as a cache unit for a slower but larger DRAM technology (such as DDR4), has been explored for CMPs and CPU servers. Universally known as DRAM caching, the stacked-DRAMs can be viewed as a very large cache unit.

A typical cache is designed to exploit temporal locality, so the common cache block sizes are limited to 64- to 128-bytes. For large DRAM caches, 64-byte blocks would require huge tag storage, therefore the tag should be embedded alongside with the data in the DRAM itself [154, 155]. However, embedding tags in the DRAM results in multiple DRAM accesses per request, as well as higher hit and miss latencies.

Loh et. al. [156] enable a more time efficient method to cache conventional block sizes in large die-stacked DRAMs. Their work consists of two main ideas. First, they implement compound-access scheduling. By scheduling the tag and data accesses as a compound access, they allow for faster hits. Second, they make misses faster by utilizing a MissMap, which is a hardware structure that keeps track of the presence or absence of cache blocks in the stacked-DRAM. However, this design still suffers from high overhead that is associated with storing the tags for each individual
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

64B (or 128B) cache blocks. These optimizations only partially reduce the high hit latency, because of the need for several operations to be performed within the DRAM row buffer. Additionally, the co-allocation of tags and data requires better data placement policies that diminish DRAM locality. It also requires a way to determine the presence of a block in the cache prior to accessing the tags, as well as additional multi-megabyte storage for that purpose. This additional storage is placed on the critical path of every accesses. Finally, block-based designs fall short of exploiting abundant spatial locality.

Alternatively, page-based caches can be used in the design of DRAM caches. Page-based caches (which we leverage in our work as well) increase the block-size to a page, reducing the tag storage overhead. Since the size of fetched data is usually equal to or larger than a row buffer in the slower DRAM technology (as we show quantitatively in Section 5.4), it allows for maximum DRAM access efficiency. It also exploits spatial locality of the stored data (the data that forms the page). Woo et. al. [157] show that paged-based caching provides an order of magnitude more hits for CMP applications in comparison to block-based DRAM caching with DRAMs of the same size. Unfortunately, page invalidation is costly when using page-based DRAM caching, since an entire block is invalidated whether its contents are used or not (later we show how our design avoids this problem in multi-GPU systems).

Footprint Cache [158] is an efficient die-stacked DRAM cache design for server processors. It also allocates data at the granularity of a page, but identifies and fetches only those blocks within a page that will be touched during the page’s residency in the cache (i.e., the page’s footprint). In doing so, the Footprint Cache eliminates the excessive off-chip traffic associated with page-based designs, while preserving their high hit rate, small tag array overhead, and low lookup latency. The Footprint Cache tightly couples the footprint prediction mechanism with the tag array. The footprint predictor uses the information from the tag array to learn page footprints, storing the footprints into a history table upon page evictions and using the footprint information upon page misses to fetch only the accessed blocks (and not the entire page). The prediction history, shared by all the tag tiles, is kept as a separate tiled structure, called the Footprint History Table (FHT). This introduces some additional costs including the cost of the structure, as well as the cost of mispredictions within the critical pass of the accesses. Due to high spatial locality of the GPU applications (as millions of threads are accessing consecutive elements of the same stored memory objects), it is actually more beneficial to cache the entire page rather than caching blocks, based on the prediction.

The Unison Cache [159] is another stacked-DRAM cache design. It bears some similarity to the Alloy Cache [160], an approach that incorporates the tag metadata directly into the stacked
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

DRAM to enhance scalability of the stacked-DRAM sizes. However, unlike the Alloy cache, pagesized caching is used in this design. Storing the metadata directly in the stacked-DRAM also removes the need for a SRAM memory to store metadata. While this is useful for CMP and server designs, it is not always beneficial for GPU-based systems. In the work presented in this dissertation, we use the SRAM storage for metadata of the pages stored in the DRAM caches. We keep the cost of the individual SRAM components lower, by using a larger number of smaller stacked-DRAMs as banks. Additionally, we partition the program’s address space between these banks. This method serves GPU applications better as it balances the load from the GPU applications over different stacks.

Similarly, Kim et al. [161, 162] proposed the use of GPU memory as a cache level for CPU memory. Tag Miss Handlers (TMHs) are used for each controller to fetch data from CPU memory. A page versioning mechanism is used to keep a copy of the data stored in GPU memory consistent with the data in the CPU memory. However, due to the lack of directories to maintain coherency, the CPU can only update data between kernel executions (using a new page version), which prevents CPU-GPU cooperative execution. In order to ensure the GPU writes are seen by the CPU, the user can configure the TMHs (via the host program) to update the CPU memory on each GPU write (write-through), which introduces excessive communication latencies (PCIe latency is added to each write). This management scheme is also not transparent to the user. TMH can use synchronization flushes (as traditional memory management schemes), which adds significant latency before CPU can start processing the data. Using our SMDs to support cache coherence protocols, solves all these issues, and also allows us to leverage from multiple discrete GPU devices in one system.

5.2 Memory Management using Unified Memory Hierarchy

As described in Section 2.5 in user-level memory management approaches (i.e., zero-copy and memcpy), the GPU chip uses the off-chip interconnect to retrieve data from the memory of another device. The memcpy approach performs transfers with higher latency versus zero-copy, and also, it blocks the kernel until data transfer through the interconnection medium is complete. Furthermore, the user is responsible for synchronizing the data between the host and device memory. This synchronization requires an additional memory copy operation.

The Runtime Unified Memory, as presented by NVIDIA, performs the management through the graphics driver and CUDA runtime libraries [56], transparent to the user. With NVIDIA UM,
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

Figure 5.1: The timeline of a CUDA application on a CPU-GPU system, managed by software-based UM.

the data is always allocated in the memory of an active GPU. Therefore, unlike memcpy, there is no initial memory copy to move the data from the host memory to GPU memory. The use of Unified Memory also enables the CPU to make changes to the data. The problem with the current software-based implementation is the large number of redundant memory copies between the host memory and GPU memory. Pai et. al. [38] and Li et al. [37] studied this issue using multiple micro-benchmarks.

In Figure 5.1 we provide a more detailed example. Three sets of redundant data copies are identified in this example. The first redundant transfer is from the GPU to host memory in order to initialize a data structure, $z$, which is 160 pages in size. However, if $z$ has already been allocated in host memory, this memory transfer was not required. The second transfer copies the initialized $z$ to GPU memory again, even though this value is no longer needed by the GPU kernel. The last redundant transfer is the final transfer of $z$ from GPU memory to host memory, even though $z$ was initialized by the CPU in the host memory and never used by the GPU. The main drawbacks of using software-based UM include: 1) the significant number of redundant memory copies, and 2) lack of support for CPU-multiGPU systems (i.e., systems with a single CPU and multiple discrete GPU devices).

HSA, which is utilized by the most recently released commercial GPUs by AMD, features a Unified Memory Space to improve programmability by allowing the GPU to use memory spaces allocated by the malloc or new functions/methods. However, the merits of avoiding data copies are only applicable to fused CPU-GPU devices, and redundant data copies are still being performed by the underlying HSA runtime (and consequently the device drivers) on discrete GPUs [55].

A hardware solution can provide several benefits since it is much easier to keep track of data movements between the memories via hardware. Hardware can actively monitor the memory organization in the system and can carry out the requested memory operations, regardless of the
physical location where the data may reside. Hardware can also be easily equipped to work seamlessly in systems that have multiple GPUs. This is while software solutions (usually through runtime and drive) imposes additional costs for keeping tracks of the data, or migration. Also the software operations are blocking (the devices are blocked until the software process is complete). The reason software is not currently capable of supporting multi-GPU systems, or it performs extensive data copies for synchronizing the CPU and GPU devices can be attributed to incapability of the software to handle GPU page faults.

Here, we offer our hardware solution that can better manage memory in the system. Our hardware-based Unified Memory Hierarchy (UMH) approach extends the typical memory hierarchy of a GPU device (which consists of L1s, L2s, and GPU memory) by adding an additional level. We treat the GPU’s main memory as another level of cache for the host memory, as shown in Figure 5.2. By adopting this design, a request from the L2 will first interrogate GPU memory. If the GPU memory does not hold the data (equivalent to a cache miss), the request is redirected to the host memory which holds the entire data set at the start of the application. The requested data is then sent back to the GPU memory, where it resides to serve any future accesses. With this solution, the hardware can keep track of the data in the system and can carry out transfers between the CPU and GPU memories.

By caching the requested data in the GPU memory, many applications can benefit from the temporal locality. Also if GPU memory receives multiple consecutive cache lines with each request (we consider the practicality of a larger transfer granularity between devices in Section 5.4), a number of applications can potentially benefit from spatial locality present. This also reduces the number of requests to the host memory. This reduction in the number of requests applies to all GPU
devices in a CPU-multiGPU system. This is important, especially since each request will traverse the interconnect, which if shared, can degrade overall system performance.

The UMH solution pairs a Stacked Memory Directory (SMD) component with each memory controller of the GPU, which 1) intercepts load and store requests to the GPU memory, 2) issues a data request in case a miss occurs in the GPU memory, and most importantly 3) stores coherency related information to maintain coherency between a CPU and multiple GPU devices in the system. Figure 5.2 shows how SMDs are interconnected in a system with four GPUs. The hardware UMH can resolve the issues observed in the software-based unified memory. First, the data is always allocated in host memory, so initialization of the data by the CPU comes for free. Second, UMH allows multiple GPUs to use the host memory as their shared main memory, so it is easy to leverage the UMH in systems with multiple GPU devices. This is not currently supported by software-based UM. Third, UMH supports the implementation of directory-based coherence protocols, which also leads to a significant reduction in coherency traffic between devices.

5.2.1 Alternative Memory Management Approaches for CPU-multiGPU systems

We evaluate our design with two alternative solutions. As stated earlier, there are no user-transparent UM solutions for multi-GPU systems, and the following solutions are only alternatives for multi-GPU systems, but only if users manually change their host program using API calls.

5.2.1.1 ZeroCopy approach for multi-GPU systems

Considering the zero-copy operations described in Section 2.5, a request from a L2 cache in a GPU can traverse the PCIe bus (or other interconnects) to the host memory to retrieve the data, instead of acquiring the data from the memory of the GPU. This can overwhelm the bandwidth of the host memory, while the high bandwidth of GPU memory can remain under-utilized. In a system that has multiple GPUs, there can be even a larger demand for data stored in the host memory because all of the L2 caches on every GPU are only accessing the host memory for data. This may lead to saturation of the host memory’s bandwidth and result in high interconnect latency (multiple GPUs will compete to read data from host memory).

5.2.1.2 MemCpy approach for Multi-GPU system

An alternative solution that will effectively utilize the high-bandwidth memory of the GPU in CPU-multiGPU systems is to explicitly allocate separate data ranges of the application’s
address space on separate GPUs, using memcpy software API calls. This means no two GPUs have the same data in their memories. By distributing data evenly across the memories of multiple GPUs, we can allow the GPU to access some of the required data through its high-bandwidth memory. However if the GPU requires the data that is present in another GPU’s memory, it will need to issue a zero-copy request to that GPU. This solution was examined by Kim et. al. \cite{28}, and will be compared to UMH in this dissertation.

5.2.2 Unified Memory Hierarchy and Paging

Our UMH design has been specifically tailored to ease memory management between a CPU and one or more GPU devices by sharing a unified address space. To support a virtual unified address space in a multi-GPU system, we have to consider a virtual-to-physical translation mechanism within the memory hierarchy. Fortunately, current state-of-the-art GPUs already account for the virtual-to-physical translation.

AMD and Intel \cite{163,164} leverage an Input/Output Memory Management Unit (IOMMU) which contains large Translation Look-aside Buffers (TLBs) and Page Table Walkers (PTWs) for address translation. The IOMMU is placed along side the memory controller. The main advantage of virtually-addressed GPU caches is that performing an address translation is not necessary until a cache miss occurs. Nonetheless, this scheme has many issues related to address synonyms and homonyms that lead to significant performance degradation \cite{165}. Alternatively, Pichai et al. \cite{166} propose dedicating a single 128-entry 4-port TLB with non-blocking and PTW scheduling logic for each CU. Power et al. \cite{167} also propose to use a TLB for each CU, while using a shared page walk cache and a shared PTW between the CUs.

Our UMH design can fit perfectly with either case. An address received by the SMDs indexes into addresses that are already translated from virtual to physical addresses, and hence the UMH design can leverage the programmability benefits of virtual memory with no additional cost.

As mentioned earlier, The SMD and HMD are the main components in our UMH design. The main design decisions for implementing SMD and HMD involve creating solutions that minimize the required storage for the tag and coherency information. However, we need to identify, beforehand, what is the information that needs to be stored for UMH in the these directories. In the next section, we describe the concept of coherency in a CPU-multiGPU system, and briefly introduce the coherence protocol utilized by our UMH, identifying the amount of information we need to store in the UMH directories.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

5.3 Coherency in UMH

As stated in the CUDA C programming guide, Unified Memory attempts to optimize memory performance, but maintaining coherence between global processors (CPU and GPU) is “its primary requirement, ahead of performance” [32]. Prior evaluations of software-based UM have also shown that runtime issues exist due to the large number of memory transfers (many of which are redundant [38]) between devices, in order to maintain a consistent view of the data.

By equipping each stacked-DRAM with a SMD which contains a cache directory, we can support a directory-based cache coherency protocol. The SMDs become responsible for tracking the location of each cache line within the GPU memory and maintaining necessary coherence-related information. Equipping each stacked-DRAM with independent SMDs allows them to behave as independent cache modules.

Provided with a hierarchical view of memory, a system with a single CPU and multiple discrete GPU devices can be viewed as a fused system. As opposed to an Accelerated Processing Unit (APU) that has an integrated CPU and a single GPU, our design provides a memory hierarchy for a system with a CPU and multiple discrete GPUs. Also known as a coupled architecture [168], the CPU cores and GPU compute units of an APU system have their own dedicated L1 and L2 cache levels, but share a last level cache before reaching the main memory. Our hierarchical design presents the same logical view and memory organization to both the CPU and multiple GPU devices, as does an APU, except we configure system memory to work as shared system memory for the entire system, and the GPU memory hierarchy is extended one level deeper, where the GPU’s stacked-DRAMs act as another cache level. The issue is especially important for GPU applications that exhibit a write-once access pattern [169]. For example, in case of MOESI, when one GPU device tries to modify a subset of data within a cache line (for example an integer value is written to bytes 0-3 of a cache line) while another device owns a version of a cache line (for example the bytes 4-7 of the cache line is modified), the protocol pushes the owner to evict the cache line fully to the shared memory (CPU memory) through the slow PCIe (or alternative) interconnect, and pass the new cache line to the modifying GPU, before the store (to bytes 0-3 of the cache line) can be performed.

Today’s GPU devices consider the notion of non-coherent operations. In these systems, a store instruction generates a non-coherent access, while an atomic store operation is coherent. The NMOESI protocol [20, 145] additionally introduces non-coherent states to support non-coherent memory accesses, while using the M state of the MOESI protocol (a common coherency standard protocol for multi-core CPU architectures) for coherent store operations. With a non-coherent
access to a cache line from a compute unit, data will transition to the non-coherent state (a non-exclusive state), which means it is not required to maintain the latest state of the data, unless it is specifically required for synchronization.

5.3.1 Leveraging The NMOESI Protocol

In a traditional coherent memory system, load and store operations translate to non-exclusive and exclusive accesses, respectively. Adding the N-state, a non-coherent store will be treated as a non-exclusive access, meaning that the non-coherent access generates the same coherence traffic as a load access, and only needs to ensure that no other cache has the same block in the exclusive state.

The key benefit of supporting non-exclusive memory requests is that we can reduce the coherence traffic as compared to exclusive requests. An exclusive coherent access from a compute unit must invalidate all copies of that cache line in the memory hierarchy. This guarantees that no other compute unit can modify a copy of that block at the same time. This is while a non-exclusive non-coherent store only updates the copy of the cache line that is local to the compute unit. Supporting non-exclusive accesses allows each compute unit to make modifications to the cache line data without having to send out any coherence update/invalidation to other compute units.

Table 5.1 shows the complete set of possible state transitions for NMOESI, using the representation introduced by Martin et al. [170]. Shaded cells represent the changes needed to support the N-state. The left column shows all possible states, the middle three columns represent actions triggered by processing memory requests, and the right three columns show requests initiated internally in the memory hierarchy.
The shaded column labeled \textit{n-store} shows the consequence of a cache line being updated by a non-coherent store. For this particular operation, a coherence-related request (read-request) is sent only if the cache block does not exist in the cache (I state). This is while for \textit{store} operations, which are exclusive coherent stores, the cache has to respond with coherence-related messages if the cache line is in one of four states \( O, S, I \) or \( N \). The first row describes the possible transitions for a block in state \( N \), which looks very similar to the transitions in the row for state \( S \). The only difference between these state transitions is that an eviction and a write request initiated for a block in \( N \) state generates a write-back to the lower-level cache.

One attractive feature of the NMOESI protocol is that it is already a superset of the MOESI protocol. This means every state of the MOESI protocol (which is generally used for multicore CPUs) is already covered by the NMOESI protocol. In a multicore CPU, each cache line in the L1 and L2 caches requires 3 bits to maintain the MOESI states (5 states). Supporting NMOESI on the CPU caches does not require any additional bits since the 3 bits can cover up to 8 states (NMOESI has only 6 states). From the NMOESI perspective, there is no difference between CPU and GPU cache units, and the only thing that matters is the state of the cache blocks within cache units. The state transitions described in Table 5.1 account for all possible NMOESI state transitions of a cache block, independent of whether the cache block is used by the CPU or the GPU. This allows our UMH design to be compatible with systems that utilize a multicore CPU as the host.

NMOESI only focuses on coherency, and does not provide any ordering by itself. Providing an ordering for a CPU-multiGPU system requires additional hardware/software support. GPUs generally exhibit relaxed consistency, which is easy to support \cite{1,20}. The strictest memory model for a multicore CPU in the CPU-multiGPU system is sequential consistency. Being the superset of the MOESI protocol has another practical benefit for NMOESI in regards to managing consistency. Any mechanism that supports consistency together with the MOESI protocol on the CPU side can also support consistency with NMOESI.

### 5.3.1.1 Example of utilizing NMOESI in a CPU-multiGPU system.

Figure 5.3 presents an example of how a system with two GPUs can leverage NMOESI to manage coherency between devices and the CPU. This example illustrates one of the most commonly used features of the NMOESI protocol in our work. Other NMOESI features include: cooperative execution, non-coherent operations for the CPU, and maintaining true coherency for the GPU \cite{20}. NMOESI is utilized to support a system with any number of GPUs, based on our Unified
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

(a) (a) NC-store: Read request misses in L1, L2 and SMD, hits in host memory. Host memory has the copy of the data so it returns the data to the requesting SMD. The data is passed to L1 and modified non-coherently.

(b) (b-1) NC-store: Same as in (a), starting from a different GPU. Read request misses in L1, L2 and SMD, and hits in host memory. Host memory receives the data from GPU0 and provides it to the requesting SMD.

(c) (b-2) NC-store (Alternative): In case a peer transfer is possible, the request from GPU1 to HMD updates the coherency fields (sharers, owners), but the data is directly provided by GPU0, reducing the traffic on the host memory.

(d) (c) Store: CPU Core Write request misses in L1, L2, and LLC. It reaches the Host Memory. HMD, which keeps track of the sharers/owners of the data, invalidates GPU copies, receives and merges the modified GPU data, and returns it to the requester.

Figure 5.3: Leveraging NMOESI in CPU-multiGPU system.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

Memory Hierarchy design. In this example, all the compute units and cores within the GPU and CPU system request the same cache line.

As shown in Figure 5.3(a), CU0 of the GPU0 issues a non-coherent write operation for the cache line (1). The request traverses the GPU memory hierarchy down to reach the appropriate SMD (2). The SMD intercepts the request and performs a lookup in its directory. Since the data is not present in the GPU’s memory (and hence in the directory), the request is forwarded by the SMD to host memory (3). A second component, the Host Memory Directory (HMD), resides between the memory controller, and the DRAM of the host memory. The HMD receives the request, locates the requested data, and directs it to the requesting SMD (4).

As shown in Figure 5.3(b-1), when CU31 of GPU1 issues a non-coherent store operation to the same cache line (1), the request is intercepted by the SMD (2) and redirected to the appropriate HMD (3). The HMD has been updated by the sharers of the cache line (the SMD of GPU0) in the previous request, so it performs a read request to the SMD of GPU0 (4). However, unlike the transitions in the MESI or MOESI coherence protocols, this transaction only updates the coherence information for the cache line in the cache units of GPU0 (identical to the same transactions that a load access initiates), and the L1 cache of CU0 in GPU0 is not required to provide the latest data to its L2 cache or SMD, or to the requester GPU1 (the data is in non-coherent state). The L2 cache in GPU0 only responds with an acknowledgment of a receipt of the update (9), while the HMD provides the data to CU31 of GPU1 (6).

In Figure 5.3(b-1), the HMD device is responsible for providing the data to the requesting SMD of GPU1 (6) during the read (resulting from a non-exclusive write request). As the number of GPU devices in the design increases, the load on the HMD devices can increase as well. The HMD devices have to consistently provide the data to multiple requesting SMDs. While this pressure on the HMD is significantly less than for zero-copy, it can still be high. Alternatively, peer-to-peer transfers between GPU devices can be leveraged (Figure 5.3(b-2)) to allow the GPU that owns the data (the cache-line is in E, O or M states) to provide the data to the requesting GPU, reducing the load on the HMD. In this case, the request from the SMD of GPU1 to the HMD is forwarded to the SMD of GPU0 (9). The HMD requests the SMD of GPU1 to listen for data from GPU0 (4), and the SMD from GPU0 ultimately provides the data to the requesting SMD (7). Upon finishing the data transfer, the SMD from GPU0 will provide the necessary information to update the sharer field of the HMD, while the SMD for GPU1 provides the data to upper cache levels (8).

Figure 5.3(c) shows how a CPU core can access data that is modified by the GPU devices. By performing an exclusive store (1), the state of the cache line changes to Modified or M. When
performing this exclusive operation, a write request (2) is made through the hierarchy to the shared host memory. The HMD which holds information about the sharers of the requested address, and sends invalidation (3) to these sharers (the appropriate SMDs of the two GPUs). Each SMD forwards the invalidation request to the higher-level L1 cache (through L2), that holds the data in the non-coherent state. Each L1 cache (4) respond to the invalidation with data through the memory hierarchy. The data is received from different GPUs and merged (6) by the HMD using a byte-mask, which will be discussed later in this section. The data is invalidated in the cache levels of both GPUs during this process. The HMD forwards the merged data to the CPU (6). This data is now coherent and ready for modification by the CPU. The CPU requires synchronization to have coherent read access to the updated data. We discuss the possible options for synchronization later in this section.

5.3.1.2 Merging Using a Byte-mask

One hardware cost of supporting non-coherence is the added dirty byte-mask required to combine non-coherent modified blocks. This support is required for GPU systems since they allow for non-coherent access to data. This byte-mask is needed since non-coherent data will need to be merged at some point to provide a coherent view of the data. For multi-GPU systems, where multiple GPUs can have non-coherent access to the same data, this hardware support is necessary as well. This feature is very typical in GPU design, for instance, the HBM technology features a multi-purpose bit for each byte of data in the die-stacked memory [63]. If the HBM is used for GPUs, this bit can be used to store the write data byte-mask. The alternative purpose of this bit is to store the error correction code (ECC). This extra bit per byte amounts to 12.5% overhead in terms of DRAM memory space.

If such space is not provided (or provided, but used for ECC), additional space is required for the byte-mask. To reduce this space overhead, multiple bytes of data can be represented with a single dirty byte-mask bit, instead of using 1 bit per byte. The trade-off of working at a coarser granularity is that smaller or non-aligned accesses cannot be accounted for, and therefore, these accesses must use regular coherent stores (e.g., if each bit in the byte-mask represents two bytes, then stores to a single byte must be coherent). The performance trade-offs of representing multiple bytes per bit are discussed by Schaa [20], and it is shown that 4 bytes per bit is a good trade-off since the performance degradation of the entire AMDAPP SDK suite does not exceed 2%. In fact, Schaa disassembles the AMD’s Southern Islands binaries, showing that AMD has made the same
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

observation, as store instructions for 1 and 2 bytes have their coherence bit set (to bypass the L1 and work directly with the L2), and larger sized accesses (i.e., multiples of 4) do not [20]. With a 1-bit byte-mask per 4 bytes of data, the byte-mask memory space overhead for the GPU and the host memories amounts to 3.12%.

5.3.2 Reduction in Synchronization Cost

Synchronization between the CPU and the GPU (or multiple GPU devices) is one of the main requirements of the memory management system. Each memory management technique uses a different mechanism to synchronize data between the GPU and the CPU. A software-based UM uses extensive copies to provide consistency between the GPU and the CPU devices. This means each synchronization requires the entire modified data set to be flushed from the GPU caches to the GPU memory (known as the Final-Flush), and then copied to the host memory. Additionally, the data that was not changed by the GPU may also be copied back from GPU memory to the host memory, even if the host memory has the exact same copy of the data.

In the zero-copy approach (See Section 5.2.1) the modified data in the cache hierarchy of each GPU needs to be flushed (i.e., Final-Flush) to the host memory directly, and does not require a copy from GPU memory to the host memory. In the memcpy approach (See Section 5.2.1) the modified data has to be flushed (i.e., Final-Flush) to the GPU memory after kernel execution. Then the host application copies the data from the GPU memory to the host memory. This copy involves specific buffers that are filled with the modified data of the executing kernel.

In our approach, an easy way to synchronize the CPU and GPU devices is to flush the modified contents of the GPU memory hierarchy (states M and N) at the final stage of the kernel execution. Similar to the method performed by the zero-copy approach, a final flush pushes the contents of the memory modules to the host memory, so later they do not require any data copies. The main difference between our UMH approach and the zero-copy approach is, when using UMH, the contents of the L2 are first flushed to the DRAM memory of the GPU, and then the modified contents of GPU memory are flushed to the host memory.

While this method has a lower cost than synchronization with the memcpy and software UM approaches, it is only comparable to or slightly worse than the zero-copy approach. However, leveraging both the NMOESI protocol and our UMH allows for a unique and fast way of enabling concurrency between the GPUs and the CPU which avoids flushing and extensive data copying. We name this approach the dynamic synchronization.
5.3.2.1 Dynamic Synchronization in UMH

We can leverage compiler assistance with our UMH design to allow the CPU to dynamically synchronize only the cache lines associated with the data that it requires at this current instance in order to continue application execution, instead of copying the application’s entire data space. The compiler should be aware of whether the CPU is requesting an address that is non-coherent (its data is used by the GPU), so it can produce an exclusive access request to this address, followed by a load. One possible implementation of the exclusive access request is to generate a store to this address with a size of zero bytes, which flushes all copies of the cache lines associated with that address (i.e., that are in state $N$) from all cache units and GPU memories within the memory hierarchy of all the GPU devices.

Our CPU-multiGPU hierarchy can greatly benefit from this feature. As we discuss later in Section 5.4, we choose to use a granularity larger than a single cache line for data transfers between the host memory and the GPU memory. We refer to this granularity as the secondary block size. For every compiler-assisted dynamic synchronization of a single cache line, a larger secondary cache block is synchronized between the two devices, allowing the CPU to benefit from spatial locality.

Figure 5.4 shows how the CPU can retrieve data from the GPU using our unified memory hierarchy and NMOESI. Figure 5.4(a) shows the timeline for CPU accesses to the data, resulting from a flush of all the GPU’s non-coherent data to the host memory. Figure 5.4(b) shows an alter-
native timeline if dynamic synchronization of a single cache line is used, and how CPU accesses lead to flushing of a single cache line from the hierarchy of the GPU device(s). Figure 5.4(c) is similar to the timeline for dynamic synchronization, but the synchronization is performed at a coarser granularity. We show the benefits of using this approach to reduce the synchronization cost in Section 5.5.

5.4 Design of SMD and HMD

5.4.1 Design of SMD

One responsibility of the SMD is to redirect the requests that are issued from upper levels of the memory hierarchy (e.g., the GPU’s L2 cache) to the host memory whenever the GPU’s stacked-DRAMs do not hold the requested data. In order to know whether the requested data is available in the stacked-DRAMs, each SMD should be equipped with a SRAM lookup table. The SMD decodes the requested address, and based on the tag, locates the row buffer where the data resides. The SMD maintains tag information for each cache line stored in the stacked-DRAM. Each incoming request is decoded into the three typical cache indexing fields: 1) the tag, 2) the index, and 3) the offset. The SMD module looks up the cache line stored in the decoded index. If a cache line is found and the tag matches, the cache line is available in the DRAM, so the request is a hit. Otherwise, the request is redirected to off-chip host memory.

The other responsibility of the SMD is to maintain coherence information related to each cache line in order to maintain the coherency between the CPU and one or more GPU devices. For the NMOESI protocol, each cache block requires 3 bits to represent the six states (Non-coherent, Modified, Owned, Exclusive, Shared, or Invalid), \(\lceil \log_2(n+1) \rceil\) bits are required to identify the owner field where \(n\) is the number of caches in the upper level), and \(n\) additional bits are required for the sharer field \([145, 20]\). In our target GPU, the AMD’s Radeon 7850 from Southern Islands GPU architecture, each L2 cache is connected to a dedicated memory controller, so each cache line in the SMD requires a 1 bit owner field, and no sharer field.

Given the large size of the stacked DRAM, we can store a very large number of cache lines in DRAM. For example, a 1GB stacked-DRAM can store 16M 64B cache lines. Since the SMD has to maintain a tag for each stored cache line, the memory required for storing the tag becomes unmanageable. This is due to the size of the tags in a 64-bit architecture. The directory for a direct-mapped, inclusive, cache can hold 16M sets. The size of the tag field, \(t\), is calculated using
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

Figure 5.5: Possible formats for the entries in the SMD. (a) Typically, each 64B cache line requires 36 bits of associated data, making the required space unmanageable. (b) We implement sub-blocking for a directory-based coherence protocol. A block of \( n \) 64B cache lines only requires 35 bits for the tag and coherence data, and each of the \( n \) cache lines only requires 1 bit to identify whether its owned by the upper-level.

The following equation: \( t = 64 - s - o - p \), where \( s \) is 24 bits, which is the required number of bits to represent the sets, \( o \) is the cache-line offset (6 bits), and \( p \) is for the extra 2 bits due to partitioning the memory space between 4 separate stacked-DRAMs (this is the number of stacked-DRAM used for our target architecture, the AMD Radeon HD 7850). \( p \) can change based on the number of available stacks). So the tag is 32 bits long. Figure 5.5(a) identifies the information that has to be stored in a single entry of the SMD. The space required to hold 36 bits per cache line is 72MB for a single 1GB stacked-DRAM! This large amount of data cannot be stored in the SRAM memory, so the alternative is to determine methods to store the tag alongside the cached data within the row buffers of the stacked-DRAM, which reduces the effective capacity of the stacked-DRAM [159, 171, 161].

Instead of storing each cache line in the SRAM separately, we can combine cache lines in a larger secondary cache block. This technique is known as sub-blocking [172]. This allows us to manage memory at a coarser granularity (e.g., a 4KB page), which will reduce both tag overhead and coherence information overhead significantly. Using this approach, we only need to maintain coherence information for a secondary cache block. As shown in Figure 5.5(b), the only information required within the entry is the secondary cache block tag, and the owner bit per cache line to identify whether the cache-line (sub-block of the secondary cache-block) is owned by the upper-level L2.

Figure 5.6 presents the effects of varying the secondary block size on area and latency of the SMD’s SRAM, as obtained by Cacti in 32nm technology [173]. The smaller secondary cache block size requires larger SRAM lookup tables. As we increase the size of the secondary cache block (and consequently the number of cache lines it contains), the area and lookup latency of the lookup table is reduced.
Figure 5.6: The impact of selecting a larger secondary block-size on area and look-up latency of the SRAM table, for 1GB stacked-DRAM.

Figure 5.7: Design space exploration of different secondary cache-block sizes, and its impact on the performance of the CPU-multiGPU system.

Figure 5.7 presents the impact of different secondary cache block sizes on the execution time of applications run on a CPU-multiGPU system with four discrete GPUs, with NVLink connections between devices. We evaluate applications from the AMDAPPSDK benchmark suite [19] (See Table 5.2). This selection covers a wide spectrum of memory access patterns. As we increase the size of the secondary block size to 4K, we see a general performance improvement for most of the applications (as seen in the Geometric Mean (GM) column). The main reason for the performance improvement is that a larger chunk of data is moved from the slow DDR host memory to the very fast stacked-DRAM of the GPU. Therefore, a large number of accesses from the GPU to the SMD find their data in this faster memory module due to spatial locality. However, as we increase the block size from 4KB to 8KB and 16KB, we see that performance starts to degrade. This is due to the interconnection network latency. Larger secondary cache blocks take longer to traverse the interconnect between the CPU and the GPU (i.e., the serialization latency of a 8K page on a link with 16 bytes per cycle bandwidth is 512 cycles, while a 1K page traverses the same link in 64 cycles).

Outlier applications that suffer from the change in the secondary block size are Floyd-Warshall (FW) and Fast-Walsh Transform (FWT). Both applications possess poor scalability (in
Table 5.2: Workloads from the AMD APP SDK. The number in the parentheses indicates the input argument to the benchmark.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Application</th>
<th>Access Pattern</th>
<th>Access Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSch</td>
<td>Black Scholes (2097152)</td>
<td>Streaming</td>
<td>Spatial (store)</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transforms (1024×1024)</td>
<td>Streaming</td>
<td>Spatial (load/store)</td>
</tr>
<tr>
<td>DH</td>
<td>1D Haar Wavelet Transform (2097152)</td>
<td>Streaming</td>
<td>Spatial (load/store)</td>
</tr>
<tr>
<td>FW</td>
<td>Floyd-Warshall (256)</td>
<td>Memory Intensive</td>
<td>Irregular, Mostly sparse (Temporal/Spatial regions)</td>
</tr>
<tr>
<td>FWT</td>
<td>Fast Walsh Transform (262144)</td>
<td>Compute Intensive</td>
<td>Irregular, Sparse</td>
</tr>
<tr>
<td>Hist</td>
<td>Histogram (2048×2048)</td>
<td>Streaming</td>
<td>Spatial (load)</td>
</tr>
<tr>
<td>MM</td>
<td>Matrix Multiplication (1024×1024×1024)</td>
<td>Memory Intensive</td>
<td>Temporal/Spatial</td>
</tr>
<tr>
<td>MT</td>
<td>Matrix Transpose (1024×1024)</td>
<td>Streaming</td>
<td>Spatial (load), Temporal (store)</td>
</tr>
<tr>
<td>MTw</td>
<td>Mersenne Twister (131072×4)</td>
<td>Streaming</td>
<td>Spatial (load/store)</td>
</tr>
<tr>
<td>RD</td>
<td>Reduction (2097152)</td>
<td>Streaming</td>
<td>Spatial (load/store)</td>
</tr>
<tr>
<td>RS</td>
<td>Radius Sort (2097152)</td>
<td>Memory Intensive, Iterative</td>
<td>Temporal/Spatial (load/store)</td>
</tr>
<tr>
<td>SC</td>
<td>Simple Convolution (512×512)</td>
<td>Memory Intensive</td>
<td>Temporal/Spatial (load), Spatial (store)</td>
</tr>
<tr>
<td>SF</td>
<td>Sobel Filter (1024×768×3)</td>
<td>Iterative</td>
<td>Spatial, Temporal via iteration (load/store)</td>
</tr>
<tr>
<td>SLA</td>
<td>Scan Large Arrays (2097152)</td>
<td>Streaming</td>
<td>Spatial (load/store)</td>
</tr>
</tbody>
</table>

terms of performance) as we increase the block size since they exhibit irregular access patterns and lack spatial locality. Increasing the secondary cache block size just increases the load latency (and interconnect traffic). Applications such as Matrix Multiplication (MM) and Radix Sort (RS) load data from addresses in memory that have large strides. So performance is improved as the secondary cache block becomes large enough (as large as the stride) to effectively prefetch the next required data.

Based on these results, we set the secondary block size to 4KB. Figure [5.8] shows the final design of our SMD component. In this figure, the directory control unit within the SMD updates the coherency information for the incoming blocks, updates the state of the sub-blocks based on the coherent requests received by the L2 cache, and redirects requests to data that does not reside in the stacked-DRAM to the lower levels of the memory hierarchy.

5.4.2 Page faults and Eviction from GPU memory

In our evaluation, we pin GPU data to the host memory so page faults do not lead to a replacement of the page that is being used by the GPU. However, our design can handle page faults if the required support is provided. If pinned pages are not utilized, the access generated to HMD has to be able to trigger a page table walk in host memory. We suggest implementing a hardware Page Table Walker (PTW) for the host memory, since 1) a hardware PTW will deliver
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

Figure 5.8: Design of a Stacked Memory Directory (SMD) component for each stacked-DRAM.

Figure 5.9: The size of each HMD and SMD for different stack sizes, in a system with 2 HMDs and 4 GPUs.

better performance than a software version of the same, and 2) a hardware PTW does not need to run OS code (presently, GPUs cannot run OS code) [166]. So a HMD miss can trigger a page walk in host memory through the PTW, but miss handling does not require significant hardware support, since the last level TLB of the CPU device can also trigger page faults. In the case of a page fault, prior to page replacement, all the secondary cache blocks associated with the replaced page have to be evicted from every GPU in the system and flushed to the host memory.

While the cost of evicting a secondary cache block (which requires invalidation of all the cache lines within the cache block) can be identified as a concern, our analysis suggests that the invalidation of large secondary cache blocks has minimal impact on performance (no secondary cache block is evicted from the GPU memory until flush time, if page pinning is employed). The main reason is due to the large size of GPU memory, which ensures that an eviction will not occur due to a capacity miss. Additionally, a large number of sets are available through the SMD, which helps to avoid conflict misses, since secondary cache-blocks are rarely assigned to the same set.

5.4.3 Design of the HMD

The host memory directory (HMD) is slightly different from the SMD in the GPUs. HMD is a SRAM-based memory, similar to the SMD design. It can either be designed on-board with the host memory, or placed non-intrusively, between the host memory and the communication medium interconnecting other devices. Unlike SMDs, HMDs require both sharer and owner fields. The
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

HMD has \( n \) bits to track the sharers of the blocks with the secondary block sizes. The \( n \) is the number of SMDs that receive their data from that HMD. For example, if our system has two separate memory modules for the host memory, the data is partitioned between these two memory modules (and 2 HMDs), so half of the SMDs in the entire system (2 out of 4 of the SMDs on each GPU) are connected to each HMD. The HMD also requires \( \lceil \log_2(n + 1) \rceil \) bits per secondary cache block to identify which SMD is the owner of that block. However no coherency state bits are required for the secondary cache blocks. More importantly, there are no tag fields for the cache lines in the HMD as the data should be physically available in the CPU memory.

Figure 5.9 shows the required sizes of the HMD and SMD components (in bytes) for different stack sizes, for a system with 4 GPUs and 2 HMDs. As we increase the size of the stacks, more blocks can be fit in a single stack, so we will need more space in both the SMDs and HMDs to store the associated data of each block. However the increase in size of the HMD is not as significant as in the SMD components due to the reasons noted above. By adding the HMD to the design of the host memory, now two accesses are required to the host memory. But since one access is to the SRAM-based HMD, which has a latency of less than 0.5\( \text{ns} \), it does not impact the performance of the host memory access. This low latency feature of HMD is due to its size, which is much smaller than SMD.

5.4.4 Required Modifications

Our implementation also involves two small modification to the GPU memory system. First, as shown in Figure 5.3, the last level cache of the CPU becomes a coherency point. This means we require a new directory for the LLC to maintain the coherency information of the data shared by the CPU and GPU devices. However, the size of this directory is very small (7KB), since the LLC itself is a small memory unit (2MB).

Second, each stacked-DRAM caches a larger 4KB secondary cache block from the host memory. In the case where we choose a smaller interleave factor (e.g., 1K) for the L2 caches and the memory controllers of the GPU, the same page can be cached in all four stacked-DRAMs, even though the L2 caches will only request 1/4 of cache lines within that 4KB block. By changing the interleave factor in the L2 and memory controllers to match the granularity of the secondary cache block, we allow only a single stacked-DRAM to buffer a single secondary cache block. This change is also necessary to allow all the cache lines within a secondary cache block to have the same tag, which translates directly to an index in the directory. Our analysis shows that the impact of using
a different interleave factor for the L2 caches is not significant. The performance degradation for switching from a 64B interleave to 4KB in a system with one GPU is 5.4%, which is compensated by the benefits of the UMH, as presented in the evaluation section.

5.4.5 Discussion on SMD Placement

The GPU’s on-chip memory controllers are in charge of issuing DRAM commands to the stacked-DRAM banks. The SMD should communicate with the memory controller in order to receive requests from other components (i.e., L2 caches, or other devices in the system) and to send data. So the most convenient placement for the SMD is on-chip, alongside the memory controller. However, our SMD occupies $7 \text{ mm}^2$, and so adding four SMD components to any chip increases the overall area of that chip by more than $28 \text{ mm}^2$.

Alternatively, the SMD can be integrated into the base logic die of the stacked-DRAM. The area overhead of including the SMD on the base logic die is not substantial, especially considering the large area of the logic die (which is approximately the same size as one DRAM die). There are two types of 3D-stacked memory designs that have been the most commercially successful, each with their own base logic die layout: 1) a 3D-stacked memory in 2.5D die-stacking technology, where stacked-DRAMs connect to the chip with DRAM address, command, and data buses through a silicon interposer, and 2) a 3D-stacked memory with a packetized memory interface (such as those provided on Hybrid Memory Cubes (HMCs)), where the stacked-DRAMs can be connected to the chip via a high-speed signaling channel.

For an HMD-based implementation where an optional base logic die can exist, the SMD can be placed on this logic die, which is located between the memory controller (on-chip) and the stacked memory banks. In this case, the memory controller can be aware of the SMD component, and help the SMD with off-chip communication. Alternatively, the SMD can be equipped with a basic router of its own, and use this router to directly communicate with off-chip components.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

through the silicon interposer. The SMD is required to intercept DRAM commands from the memory controller, and using the information stored in its SRAM, issue new DRAM commands. We considered this approach for our design (Figure 5.10 – previous page). We also take into account the longest possible latency (in addition to SMD latency itself) for each request from SMD to the memory bank, to account for these timing differences. Every request from the SMD closes the currently active row, opens the target row, and performs the read or write operation. Alternatively, for packetized memory interfaces such as HMC, our SMD can be integrated with the built-in memory controller of the stacked-DRAM on the logic die, and use the built-in router to communicate with off-chip devices.

5.5 Evaluation

In this section we present our evaluation results and compare our UMH approach to alternatives in terms of performance, scalability, and the coherence cost. Table 5.3 outlines the simulation parameters considered. In order to evaluate the UMH, zero-copy, and memcpy approaches (see Section 5.2.1), we utilize the Scalable Kernel Execution (SKE) runtime model to provide the image of a single virtual GPU for multiple GPUs [28]. Using SKE, the workgroups from the same application are assigned to multiple GPUs in a round-robin fashion.

5.5.1 Performance and Scalability

Figure 5.11 (previous page) compares the runtime of the AMDAPP SDK applications on a system with 1, 2, and 4 GPUs. For our UMH design we support peer-transfers between the GPUs to further reduce the access overhead on the host memory (see Section 5.3.1). This feature, generally increases the performance of the applications on multi-GPU systems. Our analysis suggests that,
Figure 5.11: Breakdown of runtime in a system with 1, 2, and 4 GPU devices, when using PCIe or NVLink peer-to-peer connections (P2P) for mem-copy (MC), zero-copy (ZC), or Unified Memory Hierarchy (UMH) approach. The results for each system is normalized to the execution time of application on a system that uses MC and PCIe.

on a multi-GPU system with 4 GPUs, the performance improvement achieved by peer-transfer is 5% on average, in comparison to a system without the peer-transfer, as shown in Figure 5.12. For all of the benchmarks, the PCIe connection is clearly a bottleneck. As we move to a faster peer-to-peer network (e.g., NVLink), we quickly see the benefits of a higher bandwidth point-to-point interconnect.

5.5.1.1 Performance

In our analysis, the zero-copy approach for the CPU-multiGPU system (denoted by ZC) exhibits the lowest performance for almost every application. This is due to the lower bandwidth of the host memory. The host memory bandwidth becomes saturated (all the memory requests from the L2 caches of the GPUs are accessing this memory), making this component a bottleneck during kernel execution.

The memcpy approach for the CPU-multiGPU system leverages the higher bandwidth
available from the GPU memory. However, there are two issues impacting the performance of this approach. 1) Copying data to the GPU memory prior to the execution, and back to the host memory after the execution hurts the overall performance, and 2) the application data is evenly distributed across the memories of multiple GPUs, so the L2 caches of one GPU have to constantly access the memory of other GPUs through the interconnect for the segments of data that reside in those memories.

In contrast, UMH achieves the best performance, using either peer-to-peer or shared interconnects, because it benefits from the high bandwidth of the GPU memory, and exploits the spatial and temporal locality of the accessed data. The speedup observed with UMH is on average 54%, 112%, and 92% better than memcpy (the best alternative), for systems with 1, 2, and 4 GPUs, respectively.

5.5.1.2 Scalability

As we increase the number of GPUs in the system, the workload is distributed to more GPUs (and CUs), which leads to less pressure on the L1 and L2 caches of each individual GPU. Therefore, we observe better performance for all three methods as the number of GPUs increases. Using UMH, a system with 4 GPUs has $2.3 \times$ speedup in comparison to a similar system with 1 GPU. This speedup is $1.5 \times$ and $1.76 \times$ for the zero-copy and the memcpy approaches, respectively. UMH performs better because the pressure on the host memory is significantly reduced, and the traffic is distributed across the main memory of multiple GPUs. So as we increase the number of GPUs, each GPU performs their tasks without encountering memory bottlenecks. The zero-copy approach enjoys less benefits when increasing the number of GPUs since the host memory is a
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

bottleneck. Every access from the L2 caches of all the GPUs are made to the host memory. As we increase the number of GPUs in the system with the \textit{memcpy} approach, more requests from a single GPU have to traverse the interconnect to access memory on another GPU. Therefore, the speedup achieved by the \textit{memcpy} method is not as substantial as with the UMH approach.

5.5.1.3 Outliers

As shown in Figure 5.11, UMH outperforms all the other methods. However, in the \textit{FW} and \textit{SC} applications we encounter outliers to this trend. The \textit{FW} workload is a memory intensive application with largely irregular and sparse memory accesses (they exhibit no clear locality patterns). The zero-copy approach used in a system with 4 GPUs outperforms our UMH approach since these requests access host memory and only retrieve a single 64B cache-line. However, when using the UMH method, each request retrieves 4KB data, while the GPU only uses a single 64B cache-line of that data. In this case, the interconnect becomes the bottleneck for the UMH approach. On the other hand, the \textit{SC} application possesses high spatial and temporal locality. But we can only leverage this locality if the same GPU performs the convolution of one strip (512×512 matrix has 32 512×16 strips). With a 16×16 mask, 512 workgroups are required to perform the convolution on a strip. Since we assign the workgroups to compute units of different GPUs, we effectively load 4KB blocks of data to the GPU memory such that only a portion of each is used by the compute units from that GPU.

5.5.1.4 Coherent traffic

The amount of coherency information traversing the interconnect between the SMDs and HMDs is insignificant (0.08% on average, with a maximum of 1.2% coherency traffic for FW, as compared to the amount of the data transferred over the network), but remains a necessary overhead to make the implementation of the UMH possible.

5.5.2 Dynamic Synchronization in UMH

As described in Section 5.3.2, the UMH can also leverage from the dynamic synchronization of the data between the CPU and GPU devices. During dynamic synchronization, only a single block of data (here, a 4KB block) is flushed from the GPU device, and is sent to the CPU, upon its request. This method avoids redundant flushing, and the transfer of the entire address space to the host memory. This allows the CPU to start working on the computed data as soon as possible.
Figure 5.13: The amount of data that is flushed from caches to the GPU memories during Final Flush (FF), and written back to the host memory. (R) denotes the data structures only read by GPU, and (W) denotes the data structures that GPU writes into. Software-based UM, `memcpy`, and `zero-copy` synchronize the CPU by performing FF+WR, FF+W, and FF, respectively. UMH performs dynamic synchronization.

Figure 5.13 shows the amount of data that is flushed from the GPU, and the amount that is written back to the host memory, for the three approaches: 1) software-based UM (which uses Final-Flush (FF), and writes back possibly both read and write data (WR) to the host memory), 2) `memcpy` (which uses Final-Flush (FF), and then writes back the write-buffers (W) to the host memory), and 3) `zero-copy` (which only uses the Final-Flush). However, with UMH on each GPU, a maximum of 4KB is transferred from the dedicated L2 to the SMD (and associated DRAM stack), and 4KB from this SMD to the host memory. This means 32KB (4 GPU × 4K+4K) is transferred from the GPUs to host memory. If we employ write-back L1 caches, the 16 L1’s for each GPU have to flush a maximum of 4KB to the L2 as well, increasing the dynamic synchronization data to 288KB. UMH speeds up the start of CPU operations on the GPU’s computed data by at least 13×, 20×, and 24× in comparison to `zero-copy`, `memcpy`, and software-based UM, respectively.

5.6 Discussion on UMH

In this chapter, we have proposed a bold new vision for memory management in systems with a CPU and one or more GPU devices. Our design supports seamless data transfer across all the devices, and, at the same time, creates a hierarchical view between the stacked-DRAM of the GPUs and the host memory. This makes it possible for these discrete devices to have a shared view of a unified memory that is managed by hardware, and allows for coherency between the GPU devices and the CPU.
CHAPTER 5. UNIFIED MEMORY HIERARCHY FOR MULTI-GPU SYSTEMS

We realized this design by incorporating multiple memory directory components in the design of the GPU and the host memories, and by leveraging the NMOESI coherence protocol. We were able to achieve a speedup of at least $13 \times$ in terms of synchronization time between the CPU and all the GPU devices. Additionally, architecting our vision of a unified memory hierarchy (UMH) enables us to achieve a speedup of $1.92 \times$ and $5.38 \times$ (on average) over alternative memcpy and zero-copy approaches, for a system with 4 discrete GPU devices.

UMH can be further enhanced by combining our work on silicon-photonics with this new unified memory hierarchy. This class of high-end multi-GPU system should be attractive to emerging applications in machine learning and data analytics. \[1\]

---

\[1\]This chapter interpolates material from an earlier paper by the author [174].
Chapter 6

Concluding Remarks

6.1 Conclusion

In this thesis we have accomplished three goals. First, we presented our solutions to improving the global memory of the GPU, in terms of energy-efficiency and performance. In Chapters 3 we performed a design space exploration of different Network-on-Chips for the GPU. Based on this layout-based evaluation, we identified the best existing topologies that can deliver the best performance for a GPU. We found that the performance of our designs is close to a system with ideal network. Later, we identified the behavior of the GPU application, and based on the existing trend in the GPU application traffic, tailored an energy-efficient alternative electrical NOC. We introduced our CmeshX2-asym design which provides comparable performance to the best baseline design, Cmesh, but consumes 65% less power. Additionally, we have shown that the performance of this design is 70% of the performance of an ideal interconnection network L1 caches and L2 caches of a GPU.

Second, we realized the design of high-throughput low-latency silicon-photonic based NoC for the GPU architecture in Chapter 4. We tailored the design of this NoC to meet the needs of the GPU architecture. Our GPU-specific photonic hybrid NoC (used for communication between L1 and L2) was shown to be more energy efficient than the electrical NoC crossbars. Our proposed hybrid design, constructed from using MWSR for L1-to-L2 communication and SWMR for L2-to-L1 communication, achieved better performance (2.7× on average), while achieving an energy-delay value that is lower than the traditional electrical NoC, when used in the AMD Southern Islands GPU chip with 32 CUs. We also extended the implementation of this network, examining its the scalability. The design of the photonic hybrid NoC was shown to be very scalable, when
placed on a forward-looking GPU with $4 \times$ the number of compute units, in 14 nm technology. When the 128-CU GPU system runs memory intensive applications, we can achieve up to $3.43 \times$ (2.2 $\times$ on average) performance speedup, while reducing $E^{2}P$ by up to 99% (82% on average), as compared to an alternative electrical mesh NoC.

In Chapter 5, we examined another challenging aspect of the global memory, which was the memory management between the CPU and the GPU, or multiple discrete GPU devices. Our aim was to reduce the inefficiencies in the global memory of existing memory systems that manage multiple discrete GPU devices. There were three main challenges. First, we tackled the problem of sharing data across multiple GPUs devices, and the CPU. We showed that current GPU programming frameworks are limited in their support in terms of memory management in a system with more than 1 GPUs. We considered shared memory models provided by both OpenCL and CUDA. They either use the GPU’s dedicated memory, and provide the data access through copying the required contents back and forth between GPUs, or use GPUDirect. Shared memory copies result in a large number of remote accesses from the GPU, which leads to saturation of the interconnection network (PCIe) and the host memory. Partitioned copies of the physical address space to different GPU memories resulted in transferring a large array of data that might not have been used by the GPU. Depending on the application, the traffic associated with remote accesses made by a GPU could still be an issue (if the required data resides in the memory of other GPU devices). We showed that we need a better user-transparent memory management scheme, since both of these approaches have flaws. Therefore we implemented UMH that allows GPU memories to act as cache units to the CPU memory. So the GPU only caches the data that is needed, without any extra communication overhead to any other remote GPU devices.

Second, we described how providing coherency between CPU and multiple GPU devices with the current infrastructures (such as CUDA’s runtime for NVIDIA, and HSA’s runtime for AMD) result in many redundant data transfers. We equipped our UMH design with SMD and HMD units that are also able to support different coherency protocols. By enabling heterogeneous-specific coherency protocols on top of UMH, we were able to avoid problems caused by data sharing between multiple devices in a CPU-multiGPU system.

Finally, we showed that the current methods of synchronization between GPUs and a CPU can become very costly, and currently has to wait for execution of the application on the GPU to complete. By utilizing the NMOESI coherency protocol and the support of the compiler to issue dynamic synchronization requests, we were able to significantly reduce the cost of synchronization, and also allow the synchronization to happen as soon as the CPU requests the modified data of the
CHAPTER 6. CONCLUDING REMARKS

GPU (and not at the end of the execution).

The complete implementation of UMH on a multi-GPU system with 4 discrete GPUs, resulted in a speedup of at least $13 \times$ in terms of synchronization time between the CPU and all the GPU devices. Additionally, UMH allowed us to achieve a speedup of $1.92 \times$ and $5.38 \times$ (on average) over alternative memcpy and zero-copy approaches, for a system with 4 discrete GPU devices.

6.2 Future Work

For the future, we plan on extending our work in two separate branches. First, UMH still uses remote memory accesses to the host memory. Although the number of accesses to the host is significantly reduced, we still require large pages (4 KB, as shown in our work) to be copied back and forth between the GPU memory and the CPU memory. If we utilize a truly shared memory system between CPU and multiple GPU devices, where there are no remote accesses, we can enhance the performance of the GPU significantly. This can be achieved by placing many small high-bandwidth memory units on the same interposer between the CPU and multiple GPUs, allowing these devices to share these resources. We can achieve better performance if we partition data on these shared memory units, and schedule workgroups between different GPUs based on the requirements of the applications. The cost of accesses to these shared memory units can be significantly reduced if a better link technology, such as a high-bandwidth low-latency silicon-photonic link, is utilized to create a superior topology for the interconnecting medium. We are also designing a silicon-photonic link based interconnect, that is dynamically managed to provide extremely high-bandwidth low-latency interconnect between devices during long, but sparse, communication intervals (when a page is copied from the CPU to the GPU).

Second, we would like to enhance the GPU’s scheduling in such a way that each compute unit receives workgroups based on the data that is close by. This is called memory-affinity scheduling, where a workgroup is assigned to a compute unit only if the cost associated with retrieving its data is not significant. For example, if each cache in the GPU holds a known subset of data in its L1 cache, we would want to assign workgroups based on the location of their associated data. With this enhancement, we can significantly reduce the traffic between the L1 and L2 caches, as well as the traffic between the GPU and the CPU memory.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


137


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


