2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Multi-GPU System Design with Memory Networks

Gwangsun Kim, Minseok Lee, Jiyun Jeong, John Kim
Department of Computer Science
KAIST
{*gskim, lms135, cjy9037, jjk12*}*@kaist.ac.kr*

***Abstract*—GPUs are being widely used to accelerate different workloads and multi-GPU systems can provide higher performance with multiple discrete GPUs interconnected together. However, there are two main communication bottlenecks in multi-GPU systems – accessing remote GPU memory and the communication between GPU and the host CPU. Recent advances in multi-GPU programming, including unified virtual addressing and unified memory from NVIDIA, has made programming simpler but the costly remote memory access still makes multi-GPU programming difficult. In order to overcome the communication limitations, we propose to leverage the *memory network* based on hybrid memory cubes (HMCs) to simplify multi-GPU memory management and improve programmability. In particular, we propose *scalable kernel execution* (SKE) where multiple GPUs are viewed as a single *virtual* GPU as a single kernel can be executed across multiple GPUs without modifying the source code. To fully enable the benefits of SKE, we explore alternative memory network designs in a multi-GPU system. We propose a GPU memory network (GMN) to simplify data sharing between the discrete GPUs while a CPU memory network (CMN) is used to simplify data communication between the host CPU and the discrete GPUs. These two types of networks can be combined to create a unified memory network (UMN) where the communication bottleneck in multi-GPU can be significantly minimized as both the CPU and GPU share the memory network. We evaluate alternative network designs and propose a *sliced* flattened butterfly topology for the memory network that scales better than previously proposed alternative topologies by removing local HMC channels. In addition, we propose an *overlay* network organization for unified memory network to minimize the latency for CPU access while providing high bandwidth for the GPUs. We evaluate trade-offs between the different memory network organization and show how UMN significantly reduces the communication bottleneck in multi-GPU systems.**

***Keywords*-Multi-GPU; Hybrid Memory Cubes; Memory network; Flattened butterfly**

I. Introduction

With the significant amount of computational capability available, modern GPUs (Graphics Processing Units) are not only used for graphics workloads, but are also commonly used with CPUs to accelerate general-purpose scientific, engineering and business workloads [1]. The GPU requires a significant amount of memory bandwidth to supply data to the large number of computing cores. To address this challenge, NVIDIA has announced future GPU systems with 3D stacked memory [2]. In this work, we address how future GPU systems can exploit 3D stacked memory, based on hybrid memory cubes (HMCs) [3], [4]. In particular, we focus on multi-GPU systems and explore the opportunities and challenges with a *memory network* [5], [6] interconnecting the memory modules.

One of the challenges in multi-GPU programming is data sharing between GPUs. The unified virtual addressing (UVA) [7] introduced by NVIDIA helps to simplify accessing data from remote GPU memory with a single virtual address space that is shared by the CPU and the GPUs. The unified memory [8] builds on UVA to further simplify programming as device memory allocation and data transfer are automatically managed. However, these approaches do not remove the communication bottleneck in multi-GPU systems, which includes data transfer between the host CPU and the GPUs as well as the high cost of accessing remote GPU's memory. To provide high bandwidth to GPUs, NVIDIA and IBM recently introduced NVLink [9] that provides high-bandwidth point-to-point channels between GPUs and the CPU.

In this work, we propose to achieve similar goals by introducing scalable kernel execution (SKE) and leveraging the memory network. SKE abstracts all of the GPUs in a system as a single *virtual* GPU to simplify programmability of multi-GPU systems. SKE leverages the same source code for single-GPU but takes advantage of the multi-GPU system by distributing the thread blocks or CTAs (Cooperative Thread Arrays) in a kernel across the multiple GPUs. Unlike prior software-based approaches which require a complex runtime system to analyze the memory access patterns and/or duplication of data, SKE requires a runtime system which simply distributes the kernels across the different GPUs and a single kernel can be seamlessly executed across all the GPUs.

SKE simplifies porting single-GPU workloads to multi-GPU systems but does not remove the communication bottleneck in the system. Thus, we propose leveraging a memory network to reduce the communication bottleneck. A *memory network* [5], [6] can be defined as a network that interconnects the memory modules (i.e., HMCs) that belong to different endpoint nodes (i.e., GPU or CPU nodes) together. Compared to the traditional multi-GPU system topology where GPUs were connected through PCIe channels with limited bandwidth to remote GPU memory [10], [11], the memory network provides GPUs with high bandwidth to the

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.55

remote GPU's memory – without having to communicate through the remote GPUs. We explore alternative memory network organization in a multi-GPU system that reduces the communication bottleneck between the CPU and the GPUs as well as the cost of accessing remote GPU memory.

One of the challenge in memory network is the relatively small number of channels available in the routers (or the HMCs) – e.g., current HMCs only have 8 channels. These channels are sufficient to create a low-radix topology such as a ring or a 2D mesh topology but result in higher latency and cost (network energy). As a result, the challenge is to scale the network with a high-radix topology [12] while providing high bandwidth to the HMCs. We show that with proper memory address interleaving for local HMCs directly connected to the GPUs, we minimize the need for local path diversity among the HMCs directly connected to the GPUs and propose the *sliced flattened butterfly* (sFBFLY) topology. To minimize latency for CPU in the memory network, we show how an *overlay* network organization can be exploited to minimize latency for CPU while still providing high bandwidth for GPUs.

In particular, the contribution of this work includes the following:

- We propose a scalable kernel execution (SKE) model for multi-GPU system to simplify programmability by using a collection of GPUs as a single *virtual* GPU.
- To overcome the communication bottleneck for multi-GPU systems and SKE, we explore the design space of alternative memory network organization, including GPU memory network (GMN), CPU memory network (CMN), and unified memory network (UMN).
- We exploit the communication patterns and memory mapping to propose a *sliced* flattened butterfly (sFBFLY) topology to provide high bandwidth and better scalability by removing local path diversity. To minimize latency for the CPU in a unified memory network, we propose an *overlay* network organization.

II. Background / Related Work

A. Conventional Multi-GPU Systems

Multi-GPU systems are classified into two categories [13] – a shared GPU system that consists of a single host with multiple GPUs directly connected via PCIe, and a distributed GPU system that consists of multiple hosts each with its own single or multiple GPUs. This work focuses on the shared GPU system, but our proposed techniques can be leveraged for the multiple GPUs within each node of a distributed GPU system. Traditionally, multiple GPUs in a single node are connected with PCIe switches as shown in Fig. 1(a), creating a star or tree topology [14]. Systems can be built using high-performance PCIe switches but the bandwidth among GPUs is limited to that of a single PCIe channel. For many memory-intensive workloads, this limited bandwidth can limit scalability. In order to avoid the bottleneck, the programmer needs to appropriately partition the data across GPUs and managing shared data among GPUs is a challenge. Recently, NVIDIA announced NVLink [9] for next generation GPUs (Fig. 1(b)) to provide high bandwidth between the CPU and multiple GPUs. We address the same problem in this work but with a different approach that leverages memory network.

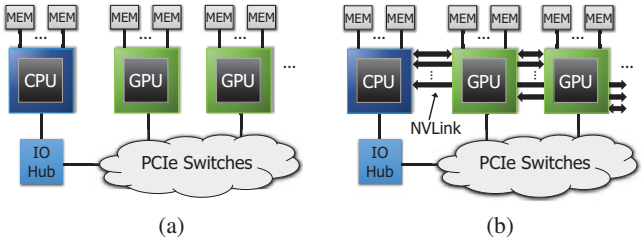


Figure 1. Multi-GPU system design based on (a) conventional PCIe interconnect and (b) recently proposed NVLink [9] for future NVIDIA GPUs.

B. Hybrid Memory Cubes and Memory Network

A hybrid memory cube (HMC) [4] is a 3D-stacked chip with multiple DRAM layers on top of a logic layer using Through-Silicon Vias to interconnect different layers (Fig. 2). Each DRAM layer is partitioned into independent segments and the segments in the same vertical slice are grouped into a *vault*. The vault includes a vault controller at the logic layer that controls access to the DRAMs and handles requests from external devices. Fig. 3 compares the conventional GDDR interface with the HMC interface. The GPU and HMCs communicate with packetized high-level request/response messages via I/O channels, whereas in GDDR, low-level DRAM commands are sent. To support a high throughput, high-speed signaling with SerDes (Serializer/Deserializer) is used for the I/Os. The logic layer provides switching capability to route incoming packets to vault memory controllers or I/O ports, if the packet's destination is another HMC. Thus, by leveraging an HMC as a router, we can create a memory network that interconnects multiple HMCs and connect it to CPUs or GPUs as shown in Fig. 2.

The memory-centric network (MCN) [5] based on HMCs was proposed to interconnect multiple CPUs to provide flexible processor bandwidth utilization. A Processor-centric network (PCN) dedicates some processor channels to processor-to-processor communication and others for processor-memory interface as commonly done in current system interconnects such as Intel QPI [15] or AMD HyperTransport [16]. In comparison, MCN dedicates all processor channels to connect to *local HMCs* (the HMCs directly connected to a given processor) and the HMCs from different processors are interconnected to form a *memory network*. Thus, all processor-to-processor packets as well as processor-to-remote memory packets are routed via the memory network through local HMCs. Although the

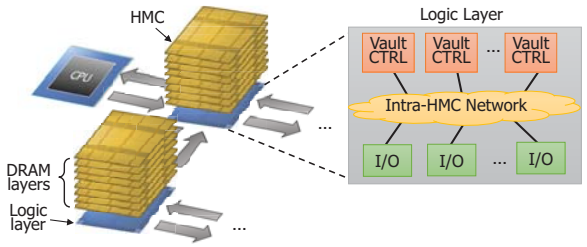


Figure 2. The Hybrid Memory Cube (HMC) architecture.

(a) (b)

Figure 3. Interface between GPU and (a) GDDR and (b) HMC memory devices.

memory network was proposed with HMCs, the concept of memory network is not necessarily limited to HMCs as other 3D stacked memory structures can also be leveraged in a memory network. Different topologies for HMCs have been proposed [17], [18], but they focus on single-processor systems and are not necessarily scalable to multi-GPU systems. Some GPUs support SLI [19] and Crossfire [20] that provides additional bandwidth among GPUs, but they are not available for GPGPU workloads. The NVLink [9] design for future GPUs provide higher bandwidth among GPUs but the topologies are limited to processor-centric network (PCN). In this work, we explore how the memory network can be leveraged in a multi-GPU system to overcome the communication bottleneck and explore alternative memory network organization.

C. Multi-GPU Programming

Scaling a GPGPU workload written for a single GPU to multiple GPUs is not necessarily trivial. The divide-and-conquer approach can be used to split a large problem into multiple sub-problems, and each sub-problem can be executed on one of the GPUs [13]. However, one challenge is that each GPU has its own memory attached to it and accessing a remote memory device is costly. In order to achieve high performance, manual division or duplication of shared data is required. Additional optimization effort is needed to overlap computation with data transfer, which requires data dependency to be tracked. The outputs of each GPU also have to be merged to produce the final result. Recent GPUs ease this difficulty by providing Unified Virtual Addressing with capability of GPUs to directly access another GPU's device memory without the intervention of a CPU [7]. However, GPU-to-GPU bandwidth is limited to that of a single PCIe channel and is significantly smaller than the bandwidth to access local device memory. Furthermore, unoptimized data transfer over PCIe can increase GPU idle time and synchronization delay [14].

Many libraries including GPUSs [21], CUBLAS [22], MAGMA [23], ArrayFire [24] and Maximus [25] have been developed for multi-GPU systems to improve programmability and provide high performance. However, these libraries are limited to a specific programming model or domain such as linear algebra and signal/image processing and thus are not sufficient for handling a diverse set of general workloads. There are other software approaches for general workloads based on existing multi-GPU hardware [26], [27], [28]. However, their limitation is that shared data have to be duplicated at each GPU, which increases data transfer time accordingly due to the PCIe bottleneck. There are different frameworks and libraries to help simplify multi-GPU programming [29], [30], [31], [32], [33], but they often require additional programmer effort to port single-GPU applications to multi-GPU systems. Recently, heterogeneous system architecture (HSA) [34] has been proposed to reduce the communication latency between CPU and GPU, and improve the programmability for heterogeneous workloads. However, HSA has mostly focused on single chip heterogeneous systems where the constraints are different from the multi-GPU systems that we explored. In the next section, we describe our proposed architecture that improves the programmability and removes the scalability bottleneck of current multi-GPU systems.

III. Scalable Kernel Execution (SKE)

We describe scalable kernel execution (SKE) which improves the programmability of multi-GPU systems by providing a collection of GPUs as a *virtual* GPU and does not require modifying the source code. By leveraging unified virtual addressing, we rely on a simplified runtime that distributes a kernel across the different discrete GPUs for SKE. However, SKE does not overcome the communication bottleneck in a multi-GPU system and we describe how the memory network (Section IV) helps to significantly reduce this bottleneck. We focus on a single kernel executed across multi-GPU system but SKE is not necessarily limited to a single kernel but can also be extended to support concurrent kernel execution [35] and dynamic parallelism [36]. We leave the extension of SKE to those features to be part of future work.

A. Kernel Execution Model

As described earlier, there are different approaches to leveraging multi-GPU systems but they require significant programmer effort to ensure correctness and/or high performance. To overcome these challenges, we propose *scalable kernel execution* (SKE). In SKE, a kernel is executed across multiple GPUs without any modification to the source code for a single GPU or any input from the programmer. A baseline kernel execution in a single GPU system is shown in Fig. 4(a). On current multi-GPU systems, a kernel needs to be partitioned into separate kernels, shown as Kernel A' in Fig. 4(b). This can be done manually or automatically

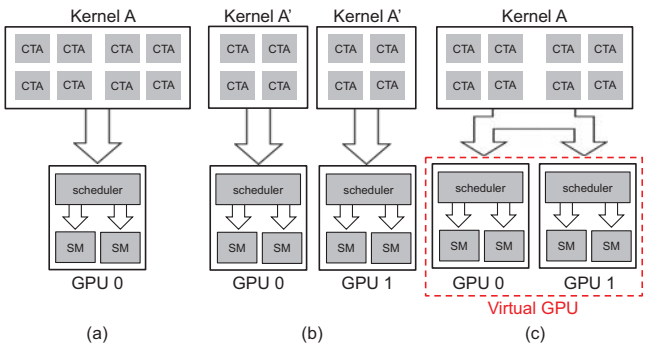


Figure 4. (a) Single GPU execution model, (b) multi-GPU system with Kernel A partitioned into two instances of Kernel A', and (c) the proposed SKE.

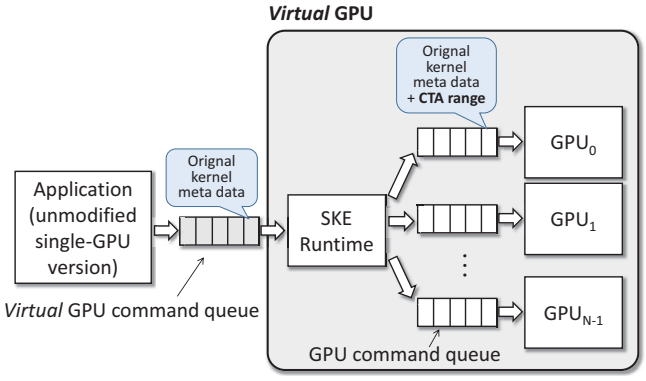


Figure 5. GPU command queues of the SKE runtime.

through different libraries or a runtime system. In comparison, SKE (Fig. 4(c)) enables a kernel to be executed across multiple GPUs without any partitioning of the kernel.

The SKE runtime system provides users with a view of a single *virtual* GPU that encapsulates multiple GPUs. Thus, instead of directly issuing kernel launch commands to the queues of multiple GPUs, the application launches unmodified kernels written for a single-GPU into the virtual GPU command queue as shown in Fig. 5. The runtime then generates multiple kernel launch commands, one for each GPU. In addition, CTA information is added to each command that specifies the range of CTAs assigned to the GPU. The GPU then executes the CTAs of the kernel within the specified range. The scheduling of the CTAs to the different GPUs is discussed in the following section.

B. CTA Scheduler

Although SKE provides a single virtual GPU, some of the resources are partitioned across the different physical GPUs. One such resource is the scheduler that distributes the CTAs to each core or stream multiprocessor (SM) in each GPU. For a single GPU system, the scheduler allocates all of the CTAs for a given kernel across all of the cores in the GPU (Fig. 4(a)). However, with virtual GPU, the CTA assignment needs to be done appropriately across the multiple GPUs to maximize performance and fully utilize all of the cores. A

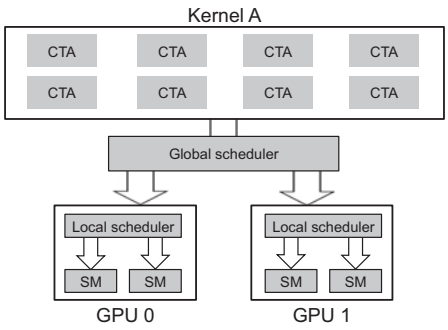


Figure 6. A two-level global scheduler that consist of a global and a local scheduler in a multi-GPU system.

global two-level CTA scheduler is shown in Fig. 6, where the centralized, global scheduler is responsible for distributing CTAs to each GPU, and a local scheduler distributes the CTAs within each GPU. However, such organization requires the global scheduler to be located either in the CPU or in one of the GPUs acting as a master GPU and can add significant overhead.

Instead of a complex centralized, two-level scheduler, we leverage a *static* CTA assignment policy across the different GPUs [26]. For n GPUs, the CTAs are partitioned into n groups and statically allocated to each GPU in chunks such that first $1/n$ CTAs are assigned to GPU0, the next $1/n$ CTAs to GPU1, and so on. For workloads with multi-dimensional CUDA grid, the CTA index is first flattened into one dimension before partitioning. We compared this assignment with fine-grained round-robin CTA assignment [37] across all cores of all GPUs and observed that our static assignment gives higher performance (8% overall) for our evaluated workloads due to memory access locality. In typical CUDA workloads with a regular memory access pattern, adjacent CTAs tend to access neighboring memory regions and thus, assigning them to the same GPU can improve cache access locality. In our evaluation, the cache hit rate was increased by up to 43% and 20% for L1 and L2 cache, respectively. Thus we assign CTAs to GPUs in chunks based on static assignment.

We also compared the static CTA assignment to a *dynamic* two-level scheduler that complements the static assignment with *CTA stealing* – after initial CTA assignment, once a core becomes idle, the global scheduler *steals* CTAs from other GPUs that have not started execution and schedules them in the idle core. However, based on our evaluation, the performance improvement by *dynamic* scheduling with CTA stealing was trivial (less than 1% improvement). The *dynamic* CTA scheduling would only increase overall performance if there was a significant CTA load-imbalance [37]. Since load-imbalance significantly decreases with large number of CTAs in a workload, static assignment was sufficient. For the rest of this work, we assume a static CTA assignment with SKE.

C. Memory Address Space Organization

With the memory network, a GPU can access any HMC in the system without going through other GPUs. Recent GPUs support virtual memory and thus, to support SKE, all of the GPUs in the system need to have the same copy of the page table to support the same virtual-to-physical address translation. The runtime system running in the CPU can keep the page tables in the different GPUs consistent, and the address translation can be done by MMUs (Memory Management Units) within each GPU [38], [39]. Thus, a request with a physical address from the GPU is injected into the memory network and routed appropriately to the destination HMC.

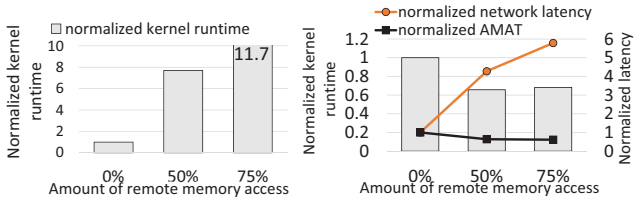
Another issue is how to map the physical address space to the different HMCs. In this work, we assume a page-based memory mapping as data are mapped at page granularity across the different HMC clusters – where a cluster consists of local HMCs for each GPU. Within each cluster, we use fine-grained cache line interleaving across the different HMCs for a given page to load-balance traffic [40]. Our memory mapping is not necessarily the most optimal mapping policy and it remains to be seen how to optimize memory mapping to increase locality in the memory network traffic and improve overall performance. The specific memory mapping used in our evaluation is described in Section VI-A.

D. Memory Hierarchy

Modern GPUs have atomic units in L2 cache to accelerate atomic memory operations [41]. This causes an issue when multiple GPUs share a memory network with SKE, since multiple atomic units exist; one within each GPU. To avoid incorrect behavior of atomic operations, we assume the atomic operations are not done in the GPU, but are moved to the HMC logic layer, near the vault memory controllers. HMCs [3] provide support for atomic transactions on the logic die and thus, we handle the atomic transactions in the HMCs instead of in the GPU. All atomic operations that occur to a cache line in L1 or L2 first evicts the line.

Current GPUs do not support cache coherence [1] between on-chip caches [43], [42] and CUDA memory models assumes relaxed memory consistency as data consistency across different CTAs within a kernel is not assumed [44]. In our multi-GPU system with SKE, we also assume the same relaxed memory consistency model; however, with caches distributed across the different GPUs, the memory model can be violated if write-back cache policy is used in the last level on-chip cache. As a result, for global memory access, we assume a write-through, write no-allocate policy for both L1 and L2.

[1]Recent work has investigated providing cache coherence in a single GPU [42]. Providing efficient cache coherence in the SKE-enable multi-GPU system is an interesting topic but we leave it as part of future work.



(a) (b)

Figure 7. Kernel runtime of `vectorAdd` on (a) an NVIDIA M2050 multi-GPU system and (b) simulated GPU memory network (Section IV-B2) as the amount of remote memory access is varied by distributing data across multiple GPUs.

IV. Multi-GPU System Interconnect

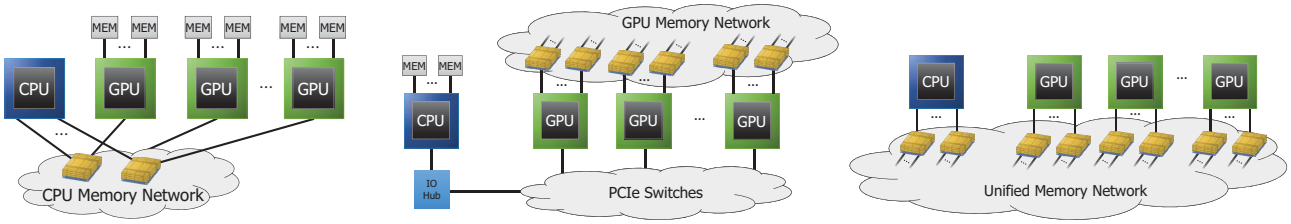
In this section, we describe the multi-GPU system architecture to support SKE efficiently. We first describe the overhead of current multi-GPU systems to support SKE because of the high cost of accessing remote memory. We then describe the alternative memory network organizations for multi-GPU systems.

A. Conventional Multi-GPU Systems

To illustrate the cost of accessing remote memory data, we evaluate `vectorAdd` [45] workload on single GPU while the data are distributed across the different GPU DRAMs. The results shown in Fig. 7(a) are based on an evaluation using multiple NVIDIA M2050 GPUs. The kernel is executed on a single GPU (e.g., GPU0 in Fig. 9(a)) while the data are distributed across 1, 2, or 4 GPU memories – e.g., if 2 GPU memories are used, 50% of the data are located on the local GPU and 50% of the data are located on remote GPU memory while for 4 GPUs, only 25% of the data are in local GPU memory. The performance is normalized to the execution time when all the data are located on a single local GPU memory. Since memory used is distributed across other GPUs, the performance degrades by up to 11.7× because of the high cost of remote memory access through PCIe.

However, unlike the conventional multi-GPU system that communicates through the PCIe switch and the remote GPU, the memory network enables direct sharing of the data and remote memory access does not significantly degrade performance. The result of executing the same `vectorAdd` on memory-network based multi-GPU system [2] is shown in Fig. 7(b). Compared to when all the data were placed at local memory modules, the workload performance actually *increased* with lower execution time when 50% of the data were placed at remote GPU memory. Since the data are distributed across more memory modules, the network latency increases with higher hop count – in this particular case, it results in approximately 50% increase in hop count. The actual average network latency increases is much higher, by approximately 4.3×, as there is more congestion in the

[2]Simulation methodology used is described in Section VI-A. For these results, we assume a 4-GPU/16-HMC system using the sliced flattened butterfly topology but only one of the GPUs is executing the workload.



(a) (b) (c)

Figure 8. Different multi-GPU system designs based on (a) CPU memory network, (b) GPU memory network, and (c) unified memory network.

network. However, the increased memory parallelism, with the larger number of memory banks from the additional remote memory modules, increase memory throughput and results in approximately 36% decrease in average memory access time, despite the increase in the network latency. Further increasing remote memory access ratio to 75% did not significantly further improve performance since GPU channels were already saturated.

B. Memory Network Organization

In this section, we describe the different memory network organization for multi-GPU systems and how they reduce the communication bottleneck while enabling efficient implementation of SKE. The three different types of memory network organization are shown in Fig. 8 and they are classified based on where in the system the memory network is leveraged.

1) CPU Memory Network (CMN):

In the CPU memory network organization (CMN), the CPU's memory is used to create a memory network (Fig. 8(a)). The memory network is not only used to interconnect the CPU with its local memory, but it is also used to connect to other GPUs as well. Thus, the memory network replaces the PCIe interconnect that was used for CPU-GPU communication and reduces the CPU-GPU communication overhead. However, each GPU is still directly connected to its own local memory, similar to the baseline. As a result, accessing a remote GPU's memory still requires having to go through the remote GPU first, but the PCIe bottleneck is removed with the memory network.

2) GPU Memory Network (GMN):

In comparison to CMN, a GPU memory network (GMN) has all of the GPUs' local memories interconnected together (Fig. 8(b)). The CPU-GPU interface remains identical to the conventional multi-GPU systems but GPUs' memories are used to create a memory network. As a result, accessing a remote GPU's memory is significantly different from the baseline or the CPU memory network organization. In either the baseline or the CMN, remote GPU's memory access required sending the request through the remote GPUs. For the conventional multi-GPU system, the request is sent through the PCIe channel to the remote GPU and then, access the remote GPU's memory as shown in Fig. 9(a). In comparison, the memory network enables a GPU to access remote GPU's memory directly through the memory network (Fig. 9(b)). Depending on the location of the remote memory, the request may traverse through one or more intermediate routers (or HMCs) before arriving at the destination memory module – however, both the PCIe channel and the remote GPU do not need to be accessed.

3) Unified Memory Network (UMN):

Both the CPU and the GPU memory network can be combined to create a single *unified* memory network (UMN) (Fig. 8(c)). In this network organization, both the CPU's memory and the GPU's memory are interconnected together in a single memory network. As a result, any memory module in the system can be accessed by traversing the memory network. Both the CMN and the GMN require explicit transfer of data from the CPU memory to GPU memory before executing kernels on the CPU. For the GMN (as well as the baseline), data transfer was done through the PCIe to the GPU, and then to the GPU's memory. For CMN, the PCIe was replaced with a memory network and thus, higher bandwidth can be provided for data transfer between the CPU and the GPU but the data still required being copied to each GPU's DRAM. However, with a single, unified memory network between the CPU and the GPU, data transfer is not necessarily required between the CPU memory and the GPU memory as the same memory network (and the memory modules) are shared between the CPU and the GPUs. One design decision is how to partition CPU and GPUs data across the different memory modules. One option is to explicitly partition the memory modules – e.g., CPU uses its local memory exclusively while GPU uses its local memory. However, given the availability of the memory network and virtual addressing, all of the physical memory can be shared by both the CPU and GPU and in this work, we assume this organization for the UMN.

Design Challenges: One of the clear advantages of a UMN organization is the removal of different types of interconnects in the system. While the baseline multi-GPU system requires having DDR memory bus, GDDR memory bus, and PCIe interface, the UMN only requires a single type of high-speed channel interface in the system. However, it does have the additional challenge of both the CPU and the GPU vendor to support the high-speed link interface for memory.

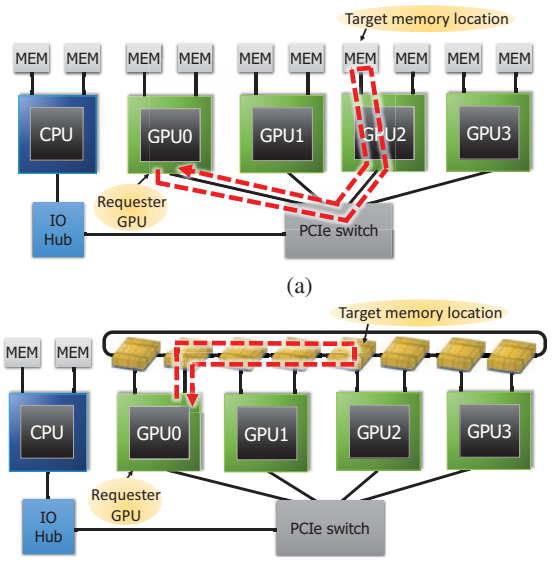


Figure 9. Comparison of remote memory access path for (a) conventional PCIe-based multi-GPU and (b) the GPU memory network (GMN). For simplicity, a ring topology is shown for the GMN.

V. Memory Network Architecture Topology

In this section, we explore the network topology for memory network in multi-GPU systems and in particular, focus on GMN and UMN network organization. We first explore the communication pattern in the SKE-enabled multi-GPU system and propose a *sliced* flattened butterfly topology that enables higher scalability while still providing high bandwidth to the GPUs. For the CPU network connectivity, we propose an *overlay* network architecture such that latency is minimized to the CPUs while the high bandwidth of the HMCs is still provided to the GPUs. In this work, we define *local* HMCs as the HMCs that are directly connected to a given GPU while the remaining HMCs are referred to as *remote* HMCs. The collection of local HMCs are also referred to as a *cluster*.

A. Network Characteristic

Prior system interconnect design with hybrid memory cubes (HMCs) [5] for a multi-CPU system is not necessarily optimal for multi-GPU systems. Since the network topology design is significantly impacted by the network traffic [46], we first look at the network traffic with our SKE-enabled multi-GPU system. We evaluate a 4 GPU-16 HMC system [3] and measured the amount of traffic between GPU-HMC pairs across different workloads. Fig. 10 shows the fraction of traffic from the different GPUs (y-axis) to the different HMC modules (x-axis). The results are shown for two workloads that have different characteristics.

For some workloads (e.g., `KMN`), the memory access across the different HMCs is nearly uniform as shown

[3]Evaluation methodology used is described in Section VI-A.

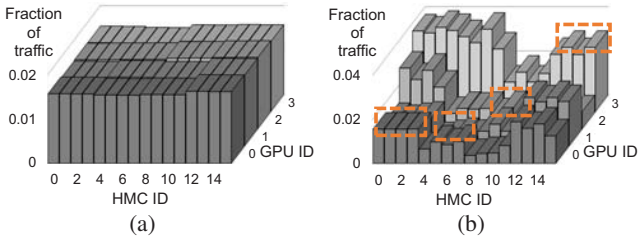


Figure 10. Distributions of traffic in a 4GPU-16HMC system for (a) `KMN` and (b) `CG.S` workloads.

in Fig. 10(a). This access pattern can be expected for many data-parallel workloads where memory is uniformly accessed. The uniform random memory access pattern (and the resulting random network traffic pattern) has significant impact on the network topology design since the traffic pattern itself load-balances the network – and thus, adaptive routing or path diversity in the network is not necessary [46]. However, for other workloads, such as `CG.S` (Fig. 10(b)), there is significant variance in traffic between the different HMCs nodes – e.g., some of the HMCs receive up to 11.7× more traffic than other HMCs. In this particular workload, the small input problem size resulted in this imbalance as there were not sufficient CTAs to be allocated to each GPU. [4] As a result, assuming uniform random traffic across all workloads is not necessarily valid.

However, by using the lower order bits to interleave across the local HMCs, the intra-cluster HMC traffic variance is significantly lower, compared with the inter-cluster HMC traffic variance. In Fig. 10(b), the intra-cluster traffic is highlighted in the dotted square box – e.g., GPU0's traffic to HMC0-3, GPU1's to HMC4-7, etc. Since we use fine-grained cache line-granularity interleaving among local HMC (Section III-C), memory traffic is balanced within a cluster.

B. Sliced Flattened Butterfly Topology

Kim et al. [5] has explored the design space of the memory network including the distributor-based dragonfly (dDFLY) and distributor-based flattened butterfly (dFBFLY). While dFBFLY provides better performance, dDFLY was preferred because of dFBFLY's limited scalability and higher cost. However, since GPUs require high bandwidth to the memory, dDFLY does not provide sufficient amount of bandwidth to remote GPU memory because of the limited connectivity – e.g., in Fig. 11(a), there is only a single channel directly connecting any two clusters. As a result, we consider dFBFLY as the baseline, but we show how it can be modified to reduce cost while improving scalability by leveraging the relatively random traffic pattern and reducing path diversity.

[4]With optimization efforts such as AMD HSA [34] to minimize GPU offloading overhead, GPU-accelerated workloads can benefit from offloading small computation as well.

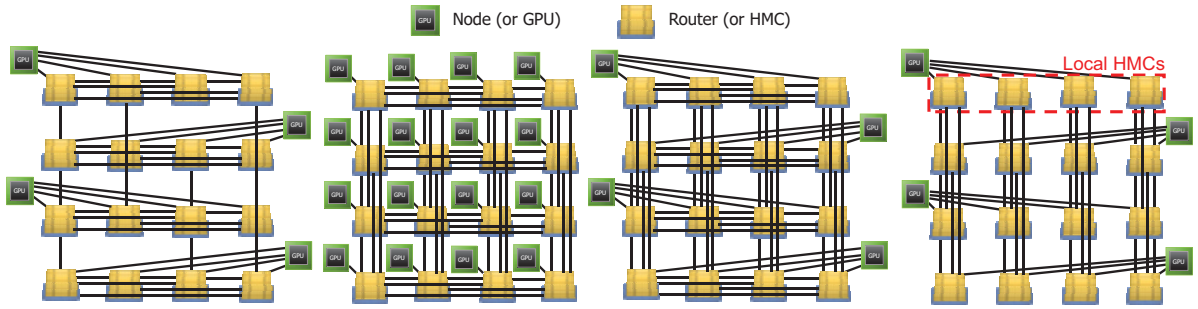


Figure 11. Different multi-GPU system design based on (a) distributor-based dragonfly (dDFLY) [5], (b) conventional flattened butterfly (FBFLY) [47], (c) distributor-based flattened butterfly (dFBFLY) [5], and (d) proposed sliced flattened butterfly (sFBFLY).

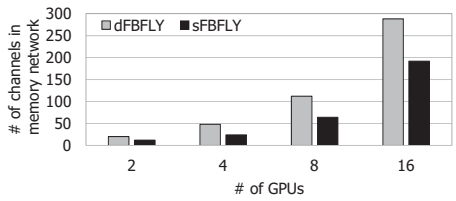


Figure 12. Comparison of the number of bidirectional channels for dFBFLY and sFBFLY.

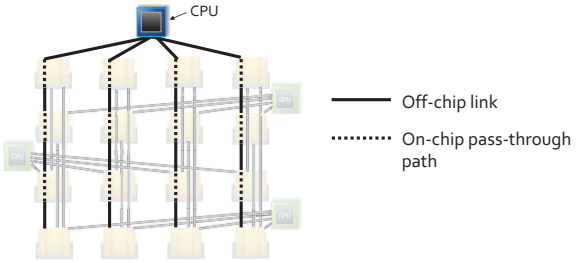


Figure 13. Memory network architecture of UMN with the CPU overlay connections.

A conventional 2D flattened butterfly topology [47] is shown in Fig. 11(b). All routers are fully connected within each row and each column. However, the number of nodes (or processors) are smaller than the number of the routers (or HMCs) in a memory network. Thus, *distribution* can be used to spread the bandwidth from a single node across multiple routers to create a distributor-based flattened butterfly (dFBFLY) [5] (Fig. 11(c)). Although dFBFLY can provide high performance, its scalability is limited with small number of router ports.

In this work, we optimize the dFBFLY and propose *sliced* flattened butterfly (sFBFLY). We exploit the traffic characteristics in multi-GPU systems described earlier; in particular, the balanced traffic and no need for non-minimal routing (or path diversity) within a cluster. The main difference, compared with dFBFLY, is the removal of channels within a cluster (indicated by the dotted box in Fig. 11(d)), where a node (or GPU) is distributed across the different routers (or HMCs). As a result, sFBFLY reduces cost with a smaller number of channels compared with dFBFLY by not providing non-minimal paths within a cluster. The minimal routing of sFBFLY between any GPU and HMC is identical to that of dFBFLY. If HMC-to-HMC packet existed, the packet would need to be routed through the GPU but such traffic does not occur in our system.

An important feature of sFBFLY is that it increases the scalability of the topology. Fig. 12 compares the number of channels in a memory network with dFBFLY and sFBFLY. Since no intra-cluster connectivity is required in sFBFLY compared to dFBFLY, sFBFLY reduces the number of channels within the memory network by 50% for a 4-GPU and 43% for an 8-GPU system. Thus, given the same number of channels, sFBFLY can scale to a larger memory network than dFBFLY. Compared to dDFLY, sFBFLY provides more inter-cluster bandwidth while reducing the total number of channels. The intra-cluster channels in the dDFLY topology are necessary to provide connectivity between any clusters in the system.

C. Overlay Architecture for UMN

For a unified memory architecture (UMN), in addition to the GPU nodes, the CPU also needs to be connected to the same network. Since CPUs are latency-sensitive, it is important to provide low-latency access in the network. Prior work on memory network design [5] showed that leveraging pass-through path across memory network can significantly reduce the packet latency. By bypassing the SerDes and the router datapath, an HMC can forward an incoming packet to a predetermined output port with minimal latency.

Since CPUs are latency-sensitive, we leverage pass-through path and propose an *overlay* memory network as shown in Fig. 13 to provide low-latency access from CPU to all HMCs. The dark-colored channels indicate the channels used by the CPU while the GPUs channels are light-colored. In this design, the CPU and its HMCs are connected to the memory network similar to GPUs, but serially connected pass-through paths for the CPU are overlaid on the memory network. While there can be path diversity in each slice depending on the topology, CPU packets take the serial pass-through path to minimize latency at low traffic load. This routing can increase the average hop count compared

Table I
SYSTEM CONFIGURATION

GPU	
Parameter	**Value**
# of cores	64 per GPU
# of HMCs	4 per GPU
Core	1024 threads, 8 CTAs, 32768 registers, 48 KB shared memory, SIMD width: 32
L1 cache	32 KB/core, 4-way, 128 B line
L2 cache	2 MB/GPU, 16-way, 128 B line
Core, Xbar, L2 clock	1400, 1250, 700 MHz
CPU	
Parameter	**Value**
Core	1 Out-of-Order core @ 4 GHz Issue width: 4, ROB size: 64
L1 I/D cache	64 KB, 4-way, 2-cycle latency
L2 cache	16 MB, 16-way, 10-cycle latency
Cache coherence	Directory-based MOESI
Cache line size	64 B
HMC	
Parameter	**Value**
HMC organization	8 layers × 16 vaults, 16 banks/vault
HMC memory size	4 GB
Memory scheduler	FR-FCFS [48], 16-entry request queue/vault
DRAM timing	tCK=1.25ns, tRP=11, tCCD=4, tRCD=11, tCL=11, tWR=12, tRAS=22

to a minimal routing with directly channels (e.g., sFBFLY), but the packet latency can be lower since per-hop latency can be significantly reduced with pass-through. If there is significant bandwidth demand from the CPU and result in the pass-through path from being congested, the CPU packet can take other paths depending on the routing algorithm.

VI. EVALUATION

A. Methodology

We modified GPGPU-sim [37] to model the multi-GPU system. The functionality of the SKE runtime system responsible for CTA assignment was also added. To model the host thread of CUDA workloads, the CPU was modeled with McSimA+ [49] while the cache hierarchy leveraged GEMS simulator [50]. The memory network was modeled with a cycle-accurate interconnection network simulator [51]. The parameters are given in Table I. For PCIe bandwidth, we assumed 15.75 GB/s to model 16-lane PCIe v3.0 channel. We used the interconnect energy model and parameters from [5] (2.0 pJ/bit and 1.5 pJ/bit for real and idle packet, respectively). We assumed each high-speed channel provides 20GB/s bandwidth in each direction, and 8 channels per CPU, GPU, and HMC. For HMC routers in the logic layer, we assumed 1.25 GHz, 4-stage pipeline, 3.2ns SerDes latency, 2 message classes with 6 VCs/class, and 512 B buffer-/VC. We used RW:CLH:BK:CT:VL:LC:CLL:BY memory address mapping (RW:Row, CLH:Column High, BK:Bank, CT:Cluster ID, VL:Vault, LC:Local HMC ID, CLL:Column Low, BY:Byte Offset). We also assumed 4KB page size and random page placement policy.

Throughout the evaluation, we assumed 4 GPUs and 4 HMCs per GPU (4GPU-16HMC) unless otherwise specified.

Table II
EVALUATED WORKLOADS

Abbr.	Input problem size	Name
BP	1M points	Back Propagation [52]
BFS	1M nodes	Breadth First Search [52]
SRAD	2K × 2K grids	Speckle Reducing Anisotropic Diffusion [52]
KMN	484K objects, 34 features	K-means [52]
BH	8K bodies	Barnes-Hut [53]
SP	100K clauses, 300K literals	Survey propagation [53]
SCAN	16M elements	Parallel prefix sum [45]
3DFD	1024×1024×4 grid	3D finite diff. comp. [45]
FWT	8M data	Fast Walsh Transform [45]
CG.S	Class S (1400 rows)	Conjugate Gradient [54]
FT.S	Class S (64 × 64 × 64)	Fast Fourier Transform [54]
RAY	1024×1024 screen	Ray Tracing [37]
STO	26MB file	Store GPU [37]
CP	512×256 grid, 100 atoms	Coulombic Potential [37]

Table III
EVALUATED MULTI-GPU ARCHITECTURES

Abbreviation	Configuration
PCIe	PCIe-based multi-GPU with memcpy
PCIe-ZC	PCIe-based multi-GPU with zero-copy
CMN	CMN-based multi-GPU with memcpy
CMN-ZC	CMN-based multi-GPU with zero-copy
GMN	GMN-based multi-GPU with memcpy
GMN-ZC	GMN-based multi-GPU with zero-copy
UMN	UMN-based multi-GPU (no copy)

Since we assumed 8 channels per GPU, each GPU was connected with two channels to each of its local HMCs with distribution. We assumed 2D connectivity for 4GPU-16HMC dFBFLY, and 4×4 2D FBFLY for each slice within 16GPU-64HMC sFBFLY. We used the workloads from NVIDIA CUDA SDK examples [45], Parboil [55], Rodinia [52], LonestarGPU [53], CUDA-ported NAS parallel benchmark suite [56], [54], and [37] as described in Table II without modification.

B. Memory Network Organization

In this section, we compare the alternative multi-GPU architectures, as summarized in Table III. For each interconnect except for UMN, we present performance with memcpy and zero-copy. With memcpy, the kernel blocks until data copy is performed between the CPU and GPU, whereas with zero-copy, data reside in CPU memory and are directly accessed by GPUs without copying. However, the interconnect between the CPU and GPUs can become bottleneck as its bandwidth can be much lower than GPU's memory access bandwidth. For the UMN, since the CPU and the GPU share the same HMCs, we assume no-copy.

Fig. 14 shows the kernel execution and memcpy time for different workloads. The UMN provides the highest performance for all workloads since it avoids memcpy overhead while providing high-bandwidth with memory network. In comparison, PCIe and PCIe-ZC resulted in the worst performance due to the PCIe channel bottleneck. The PCIe channel becomes bottleneck in not only performing

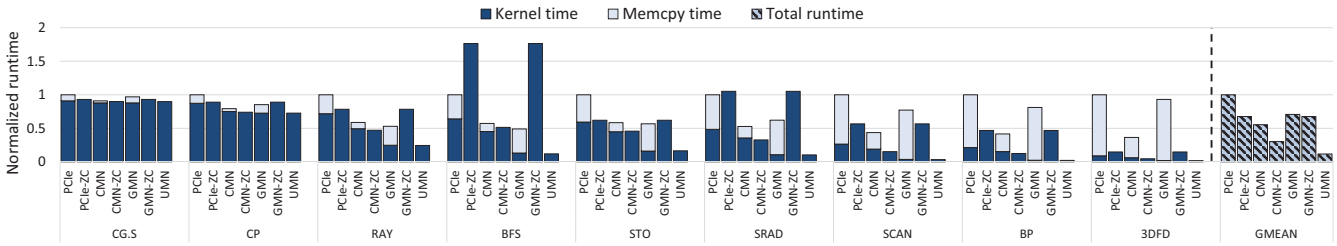


Figure 14. Breakdown of runtime in different multi-GPU designs based on PCIe, CMN, GMN and UMN.

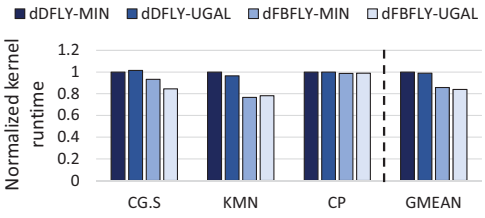


Figure 15. Performance comparison of minimal (MIN) and load-balanced (UGAL) routing on distributor-based dragonfly and flattened butterfly topology.

memcpy but also during kernel execution since we are assuming SKE and GPUs need to frequently access remote GPU memory or CPU memory (for zero-copy) which can aggravate the bottleneck. For example, with BFS, the PCIe-ZC increased the kernel runtime by 2.75× due to low remote memory access bandwidth compared with when the data were copied to GPU memory. For workloads such as 3DFD, BP, and SCAN, since the memcpy time is longer than kernel execution time, zero-copy resulted in shorter total runtime by avoiding memcpy.

There is a similar trend in the CMN and GMN as well, although the actual amount of benefit differs. The benefit of CMN over PCIe is the higher bandwidth between GPUs and the host, which lowers memcpy latency and also improves zero-copy performance. As a result, the CMN and CMN-ZC reduced total runtime over PCIe by 1.8× and 2.2×, respectively. The GMN also provides higher bandwidth among GPUs, which makes SKE more efficient, reducing kernel execution time by up to 8.8× for BP and 3.5× on average compared to PCIe. However, the single PCIe channel bottleneck still exists between the CPU and GPUs. As a result, the GMN-ZC provided the same performance as PCIe-ZC since the GPU memory was never accessed and the memory network did not make any difference. Overall, the GMN reduced the total runtime by 30% and 33% compared to PCIe. UMN further reduced the overall runtime by 8.5× by eliminating memcpy and providing high SKE performance compared to PCIe.

1) Impact of Adaptive Routing: We describe the performance benefit of intra-cluster adaptive routing in Fig. 15. We only show representative workloads since others showed a very similar pattern. Because of the relatively random access, many workloads including KMN and CP result in minimal improvement with adaptive routing – approximately only 1-2% performance improvement over the same topology with only minimal routing. For CG.S, there is 9.5% performance improvement by leveraging adaptive routing in dFBFLY as the workload showed traffic variance across the different HMCs.

2) Topology Evaluation: Fig. 16 compares the performance of different sliced memory network topologies, including sliced mesh (sMESH) and sliced torus (sTORUS). Similar to the sliced FBFLY, the local HMC cluster in these topologies are directly connected to the GPUs without any additional connectivity between the local HMC nodes. Since sMESH and sTORUS use a smaller number of channels per HMC compared to sFBFLY, in order to provide a fair comparison, we also evaluated sMESH-2x and sTORUS-2x where the number of channels within each slice was doubled to provide more bandwidth. For most of the workloads, sFBFLY provided a better or comparable performance to sMESH-2x and sTORUS-2x, which outperformed sMESH and sTORUS by providing a higher network bandwidth. Compared to sTORUS-2x, sFBFLY provides the same bisection bandwidth, but sFBFLY performed better due to lower average hop count.

Fig. 17 compares the energy consumption of different memory network topologies during kernel execution. Compared to sMESH and sTORUS, sMESH-2x and sTORUS-2x consumes more power due to additional channels but, by reducing overall kernel runtime, lowered energy by 6.8% and 4.8%, respectively. The sFBFLY further reduced the network energy by reducing runtime. Compared to sMESH, sFBFLY reduced network energy by up to 50.7% for BP and 20.3% on average. Thus, sFBFLY resulted in the highest performance while minimizing energy consumption.

Fig. 18 shows the performance of overlay network compared to sMESH and sFBFLY that do not use pass-through. We used 1CPU-3GPU-16HMC in this evaluation and we evaluated the only two workloads (CG.S and FT.S) that uses CPU for computation. For the other workloads, CPU does minimal computation and the use of the overlay network has minimal impact on performance. The performance gap between sMESH and the overlay is made by pass-through path that reduces per-hop latency. The overlay performed better than sFBFLY by significantly reducing per-hop latency even though the hop count was higher as discussed in Section V-C.

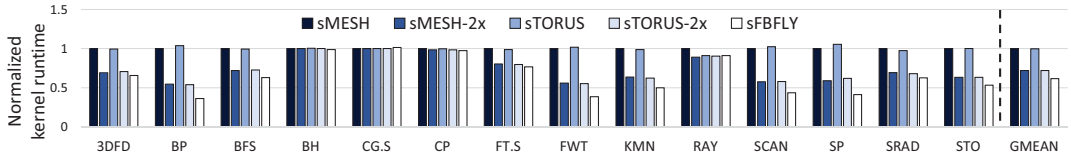


Figure 16. Performance comparison of different sliced network designs based on mesh, torus, and flattened butterfly.

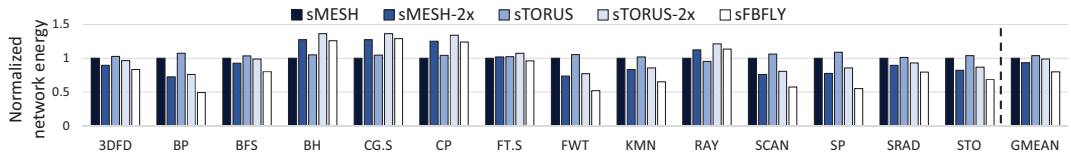


Figure 17. Energy comparison of different sliced network designs based on mesh, torus, and flattened butterfly.

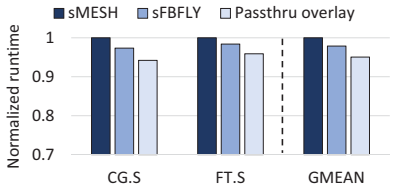


Figure 18. Host thread (CPU) performance with different UMN designs.

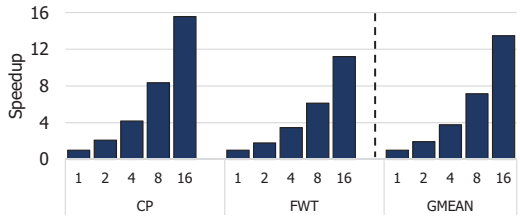


Figure 19. Speedup of kernel execution as the number of GPUs is increased on the horizontal axis.

3) Scalability: For scalability study, we increased the input problem size but because of the infeasible simulation time for some of the workloads, we only evaluated `3DFD`, `BP`, `CP`, `FWT`, `RAY`, `SCAN`, and `SRAD`. Fig. 19 shows the speedup for kernel execution as the number of GPUs is increased. We show the two workloads with the highest and the lowest scalability. On average, the evaluated workloads showed high scalability as the geometric mean of speedup was 13.5 for 16 GPUs. `CP` resulted in a near-ideal performance scaling as it is compute intensive and the increased memory network latency had little impact on performance. With 8 GPUs, the performance was 35% better than the ideal speedup due to the side effect of increased L2 cache hit rate as the number of CTAs scheduled to each core was reduced. The `FWT` showed the lowest speedup (11.2×) with 16 GPUs as the input problem size was not large enough to keep the cores busy.

VII. Conclusion

We explored how different memory network organizations can be used in multi-GPU systems to provide high memory bandwidth and improve programmability. We proposed *scalable kernel execution* that enables a single kernel to be executed across multiple GPUs without modifying source code or manually partitioning data across the different GPU devices. To overcome the communication bottleneck between the host CPU and the GPU as well as the high cost of accessing remote GPU memory, we proposed alternative memory network organization in multi-GPU systems including the unified memory network. We also proposed the *sliced* flattened butterfly topology for the memory network that exploited the communication pattern in multi-GPU systems and the *overlay* network to provide low latency for CPU in the UMN.

Acknowledgment

This work was supported in part by SK Hynix, the MSIP under the Mid-career Researcher Program through NRF (NRF-2013R1A2A2A01069132), and the ITRC support program supervised by the NIPA (NIPA-2014-H0301-14-1018).

References

[1] J. Owens *et al.*, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] "Nvidia to Stack up DRAM on Future Volta GPUs," http://www.theregister.co.uk/2013/03/19.

[3] "Hybrid Memory Cube Specification 1.0," Hybrid Memory Cube Consortium, 2013. [Online]. Available: http://www.hybridmemorycube.org/

[4] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," *Hot Chips 23*, 2011.

[5] G. Kim *et al.*, "Memory-centric System Interconnect Design with Hybrid Memory Cubes," in *Proceedings of PACT'13*.

[6] D. R. Resnick, "Memory Network Methods, Apparatus, and Systems," US Patent Application Publication US20100211721 A1, 2010.

[7] T. C. Schroeder, "Peer-to-Peer and Unified Virtual Addressing," GPU Technology Conference, NVIDIA, 2011.

[8] M. Harris, "Unified Memory in CUDA 6," GTC On-Demand, NVIDIA, 2013.

[9] D. Foley, "NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data," http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data.

[10] "HP ProLiant SL270s Gen8 Server Quickspecs," HP.

[11] "Dell PowerEdge C410x Rack Server Technical Guide," DELL.

[12] J. Kim *et al.*, "Microarchitecture of a high-radix router," in *Proceedings of ISCA'05*.

[13] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Processing of IPDPS'09*.

[14] P. Micikevicius, "Multi-GPU Programming," GPU Computing Webinars, NVIDIA, 2011.

[15] D. Ziakas *et al.*, "Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures," in *HOTI'10*.

[16] "HyperTransport I/O Technology Overview," The HyperTransport Consortium, Tech. Rep., June 2004.

[17] P. Rosenfeld, "Performance Exploration Of the Hybrid Memory Cube," Ph.D. dissertation, the University of Maryland, 2014.

[18] K. Sudan, "Data Placement for Efficient Main Memory Access," Ph.D. dissertation, the University of Utah, 2013.

[19] "SLI Best Practices," White Paper, NVIDIA, 2007.

[20] "ATI CrossFire Pro User Guide," White Paper, AMD, 2009.

[21] E. Ayguadé *et al.*, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of Euro-Par'09*.

[22] "CUBLAS Library User Guide," NVIDIA, 2012.

[23] S. Tomov *et al.*, "Dense linear algebra solvers for multicore with gpu accelerators," in *Proceedings of IPDPSW'10*.

[24] J. Malcolm *et al.*, "ArrayFire: a GPU acceleration platform," in *Proc. of SPIE Vol*, vol. 8403, 2012, pp. 84 030A–1.

[25] "NVIDIA Multi-GPU Technology," NVIDIA. [Online]. Available: http://www.nvidia.com/object/multi-gpu-technology.html

[26] J. Kim *et al.*, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proceedings of PPoPP'11*.

[27] J. Lee *et al.*, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of PACT'13*.

[28] T. Diop *et al.*, "DistCL: A framework for the distributed execution of opencl kernels," in *Proceedings of MASCOTS'13*.

[29] M. Strengert *et al.*, "CUDASA: Compute Unified Device and Systems Architecture," in *Proceedings of EG PGV'08*.

[30] C. de La Lama *et al.*, "Static Multi-device Load Balancing for OpenCL," in *Proceedings of ISPA'12*.

[31] U. Dastgeer *et al.*, "Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems," in *Proceedings of IWMSE'11*.

[32] J. Kim *et al.*, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of ICS'12*.

[33] R. Aoki *et al.*, "Hybrid opencl: Enhancing opencl for distributed processing," in *Proceedings of ISPA'11*.

[34] G. Kyriazis, "Heterogeneous system architecture: A technical review," AMD, 2012.

[35] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," White Paper, NVIDIA, 2009.

[36] S. Jones, "Introduction to Dynamic Parallelism," GPU Technology Conference, NVIDIA, 2012.

[37] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of ISPASS'09*.

[38] S. H. Duncan *et al.*, "Method and apparatus for providing peer-to-peer data transfer within a computing environment," US Patent Publication US7451259 B2, 2008.

[39] C. S. Case *et al.*, "Multi-client virtual address translation system with translation units of variable-range size," US Patent Publication US7334108 B1, 2008.

[40] E. Cooper-Balis *et al.*, "Buffer-on-board memory systems," in *Proceedings of ISCA'12*.

[41] D. B. Glasco *et al.*, "Cache-based control of atomic operations in conjunction with an external ALU block," U.S. Patent 8135926, 2012.

[42] I. Singh *et al.*, "Cache coherence for GPU architectures," in *Proceedings of HPCA'13*.

[43] P. Micikevicius, "GPU Performance Analysis and Optimization," GPU Technology Conference, 2012.

[44] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., 2010.

[45] "CUDA C/C++ SDK code samples," NVIDIA, 2011.

[46] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[47] J. Kim *et al.*, "Flattened Butterfly: a cost-efficient topology for high-radix networks," in *Proceedings of ISCA'07*.

[48] S. Rixner *et al.*, "Memory Access Scheduling," in *Proceedings of ISCA '00*.

[49] J. Ahn *et al.*, "McSimA+: A Manycore Simulator with Application-level+Simulation and Detailed Microarchitecture Modeling," in *Proceedings of ISPASS '13*.

[50] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[51] N. Jiang *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Proceedings of ISPASS'13*.

[52] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of IISWC'09*.

[53] M. Burtscher *et al.*, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of IISWC'12*.

[54] "High Performance Computing with GPUs," http://hpcgpu.codeplex.com/.

[55] J. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Center for Reliable and High-Performance Computing, 2012.

[56] D. H. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.