UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs

AMIR KAVYAN ZIABARI and YIFAN SUN, Northeastern University
YENAI MA, Boston University
DANA SCHAA, Northeastern University
JOSÉ L. ABELLÁN, Universidad Católica San Antonio de Murcia
RAFAEL UBAL, Northeastern University
JOHN KIM, KAIST
AJAY JOSHI, Boston University
DAVID KAELI, Northeastern University

35

In this article, we describe how to ease memory management between a Central Processing Unit (CPU) and one or multiple discrete Graphic Processing Units (GPUs) by architecting a novel hardware-based Unified Memory Hierarchy (UMH). Adopting UMH, a GPU accesses the CPU memory only if it does not find its required data in the directories associated with its high-bandwidth memory, or the NMOESI coherency protocol limits the access to that data. Using UMH with NMOESI improves performance of a CPU-multiGPU system by at least 1.92× in comparison to alternative software-based approaches. It also allows the CPU to access GPUs modified data by at least 13× faster.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Hardware** → integrated circuits;

Additional Key Words and Phrases: Unified memory architecture, memory hierarchy, graphics processing units, high performance computing

ACM Reference Format:
Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs. ACM Trans. Archit. Code Optim. 13, 4, Article 35 (December 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2996190

1. INTRODUCTION

In a growing number of platforms, accelerators are being leveraged to increase the performance of the applications and, at the same time, reduce energy and cost [Hameed et al. 2010]. Graphic Processing Units (GPUs) have become the accelerator of choice,

This work was supported in part by NRF-2013R1A2A2A01069132, NSF grants CNS-1525412 and CNS-1319501, and by the Spanish MINECO under grant TIN2016-78799-P.
Authors' addresses: A. K. Ziabari, Y. Sun, R. Ubal, and D. Kaeli, Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave, Boston, MA 02115; emails: {aziabari, yifansun, ubal, kaeli}@ece.neu.edu; Y. Ma and A. Joshi, Department of Electrical and Computer Engineering, Boston University, 8 Saint Mary's Street, Boston, MA 02215; emails: {yenai, joshi}@bu.edu; D. Schaa, Advanced Micro Devices (AMD), 1 AMD Pl, Sunnyvale, CA 94085; email: dana.schaa@amd.com; J. L. Abellán, Computer Science Department, Universidad Católica San Antonio deMurcia, Avenida Jerónimos, 135, 30107 Guadalupe, Murcia, Spain; email: jlabellan@ucam.edu; J. Kim, Department of Computer Science, KAIST, 291 Daehak-ro, Guseong-dong, Yuseong-gu, Daejeon, South Korea; email: jjk12@kaist.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1544-3566/2016/12-ART35 $15.00
DOI: http://dx.doi.org/10.1145/2996190

given their ability to process thousands of concurrent threads in a wide range of applications [Harrison and Waldron 2007; Vijaykumar et al. 2015; Harish and Narayanan 2007; Stuart and Owens 2011].

The class of applications that are currently best suited to run on a GPU are highly scalable and tend to have a large degree of data-level and task-level parallelism. As the size of the data sets and number of processing steps present in these applications continue to increase, we will quickly outgrow the computing resources provided by a single GPU. Efficient coupling of multiple discrete GPU devices is an attractive platform for processing large data sets, especially to the High-Performance Computing (HPC) community. A multi-GPU system can provide substantial time savings by providing higher processing throughput and enabling more flexible management of system resources such as memory bandwidth. Previous work [Schaa and Kaeli 2009; Kim et al. 2011; Nere et al. 2011] has shown that we can significantly improve the throughput of HPC applications when combining multiple GPUs, and, therefore, there has been an increased focus on how best to utilize and program multiple GPUs to address this trend in a range of applications [Kim et al. 2014a; Al-Saber and Kulkarni 2015; Cabezas et al. 2015; Mohanty and Cole 2007].

The major GPU vendors, NVIDIA and AMD, have also heavily invested in bundling multiple discrete GPUs into a unified system. There are many devices commercially available that utilize *dual* GPU devices. Among the most commercialized designs are NVIDIA Tesla Dual GPU Kepler K80 [NVIDIA 2015b], which includes two identical Tesla GK210 GPUs connected to each other through an on-board Peripheral Component Interconnect (PCI) switch, and AMD Radeon R9 295X2, which marries two AMD Radeon R9 Series GPUs on one card using Hypertransport link technology (a direct point-to-point link, similar to NVLink) [AMD 2014a]. Finally, Super Micro Inc. has introduced SuperWorkstation [SuperMicro 2016], which utilizes four GPUs. These GPUs are connected to each other through the PCIe 3.0 of the workstation itself but are managed (for load-balancing and multitasking) through NVDIA's multi-GPU Maximus Technology [NVIDIA 2012].

In comparison to single-GPU architectures, multi-GPU architectures introduce new challenges. One of them is efficient data sharing and memory management between multiple GPUs. Given our vision to enable multiple GPUs to collaboratively execute the same application, the GPUs should be able to share the data required by that application. One common approach used to simplify access to application data and improve the performance is to provide a shared address space across the host (CPU) and devices (GPUs). One of the first moves toward memory unification was NVIDIA's *Unified Virtual Addressing* [NVIDIA 2015a]. This solution provides explicit Application Program Interface (API) function calls, *cudaHostAlloc* and *cudaMemCpy*, that allow the user to manage data allocated on either the CPU or the GPU(s). As shown in Figure 1(a), using *cudaHostAlloc*, the user can *pin* data used by the GPU to the host (CPU) memory (we will refer to the CPU's memory as host memory throughout the rest of this article). Any GPU access to that data is performed using a *zero-copy* operation [Harris 2013]. 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.

One major drawback of this approach is the underutilization of the GPU's high memory bandwidth. In *zero-copy*, accesses are made to the host memory. This is while today's GPUs have a large memory space and can provide bandwidth that is much higher than the bandwidth present on host memory [Agarwal et al. 2015] (i.e., *High-Bandwidth Memory* (HBM) interface, targeted for GPUs, provides more than 100GB/s bandwidth per Dynamic Random-Access Memory (DRAM) stack, with multiple stacks integrated per chip [AMD 2015b]). Another drawback of *zero-copy* is the use of pinned pages to keep the data used by the GPU in the host memory until the end of the

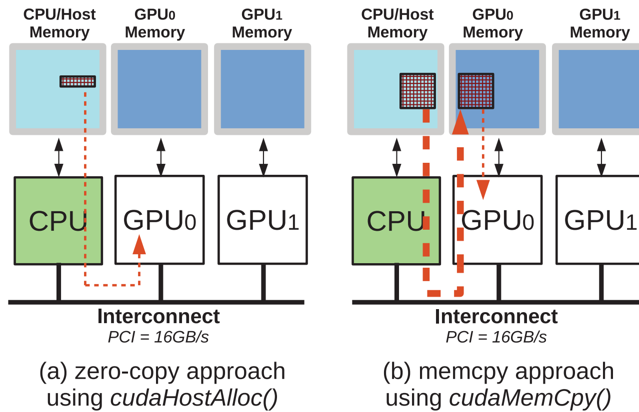


Fig. 1. Current approaches used to manage memory for GPU applications.

GPU kernel execution. This constrains the amount of physical memory available to the operating system.

As an alternative, CUDA's *cudaMemCpy* API function call allows the user to explicitly copy data from the host memory to the GPU memory, as shown in Figure 1(b). Using this *memcpy* approach, the GPU program can leverage the high-bandwidth memory provided on a GPU. Also, memory pages are not required to remain pinned in the host memory. However, a copy of the application's entire address space is performed between the CPU memory and the GPU memory (transfers between devices are usually at the granularity of OS pages [Lake 2014]), which takes a long time to complete. Until the data copy is finished, kernel execution is stalled on the GPU. While it is possible in some cases to overlap kernel execution and data transfers through software (using multiple contexts or streams), it still comes with the cost of some added programming complexity. Finally, similarly to *zero-copy*, this approach also relies on the user to manage the data copies explicitly, thereby further increasing programming complexity.

Delivering an efficient memory management system for a multi-GPU platform is a challenge, especially if management is the responsibility of the programmer. To remove this burden, NVIDIA introduced *Unified Memory* (UM) [Harris 2013], which enhances the Unified Virtual Address by 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 sometimes degrade the performance of applications when both the CPU and the GPU share data during program execution. These performance issues are reported by Li et al. [2015] and Pai [2014] 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 variables that are 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 [Negrut 2014; NVIDIA 2015a], leading to slower memory operations and performance degradation. Third, software UM always assumes that the GPU data is modified [Pai 2014]. This means that on every synchronization between the CPU and the GPU, all the GPU data need 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. A software

UM implementation relies heavily on synchronization to provide coherency between the CPU and the GPU, and, as a result, many redundant data transfers are performed only to provide coherency. This is another challenge present in systems that employ one or more GPU devices.

These inefficiencies in user-based memory management (*zero-copy* and *memcpy*) or the software-based Unified Memory are not limited to frameworks based on CUDA. Starting from version 2.0, OpenCL also provides Shared Virtual Memory to allow pointer sharing between the host program and the kernel, allowing for similar *zero-copy* and *memcpy* memory management. Similarly to software-based UM, in the current version of OpenCL driver, memory copy is still used as the synchronization mechanism. HSA, which is utilized by the latest 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 merit of avoiding data copies is 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 [Sun et al. 2016].

In this article, we present a novel hardware-based approach to manage UM that is precisely designed to avoid these redundant data transfers. 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 the Scalable Kernel Execution (SKE) runtime [Kim et al. 2014a] 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.

This unique vision for memory management provides a single logical view of memory between the CPU and GPU devices as well. Our proposed approach, named Unified Memory Hierarchy (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. Our SMD design can redirect accesses to the memory where the data reside, whether the address is in the main memory of the GPU or the host memory. The SMDs can also maintain coherency information of the data that are 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 are 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 are 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 leverage the NMOESI coherence protocol [Schaa 2014; Ubal and Kaeli 2015], originally designed for Accelerated Processing Units (APUs or fused system). NMOESI provides coherence between a CPU and a GPU that are on the same die, reducing synchronization costs between devices. Using NMOESI alongside our Unified Memory Hierarchy, a synchronization operation results in a transfer of modified data by the GPU devices (unlike the software-based UM mechanism implemented by the CUDA runtime), while unmodified data are not needlessly copied back again to the CPU.

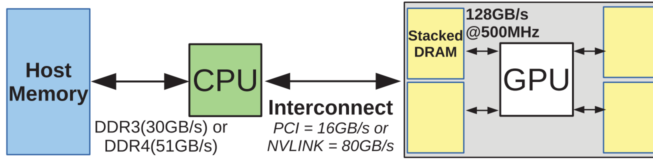


Fig. 2. Logical view of the interconnects between the CPU and the GPU and their associated memories.

The contributions of this work include the following:

—We perform a thorough study of the limitations of the *zero-copy* and *memcpy* approaches, considering systems with multiple GPUs and the SKE [Kim et al. 2014a].
—We describe a novel hierarchical unified memory architecture for systems that utilize any number of GPU devices. Our UMH keeps the memory management process 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. We modify NMOESI to explore the benefits of variable sized sub-blocks in different levels of the memory hierarchy, allowing our system to achieve better performance with little additional cost.

2. TARGET SYSTEM AND EVALUATION FRAMEWORK

We consider an x86-based architecture for the CPU [Shanley 2010] and AMD's Southern Islands architecture as our baseline GPU device. We 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 800MHz [AMD 2014b]. The HD 7850 is designed for general-purpose computing [AMD 2012].

The Radeon HD 7850's CUs can execute 256 threads from one workgroup at a time. CUs are equipped with L1 caches that are connected to four L2 caches using a crossbar network. The application address space is interleaved across the L2 units, addressable on a cache line granularity [AMD 2012], and each L2 unit is connected to a separate main memory controller.

We use three-diminsional- (3D) stacked DRAM memories for the main memory of the GPU architecture. 3D-stacked DRAM memory is used in the design of AMD's next-generation high-performance memory. 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 Field-Programmable Gate Arrays (FPGAs)) are moving toward this direction [AMD 2015a; Dorsey 2010; Gupta 2015]. Figure 2 provides a logical view of the interconnections between the CPU and one GPU and their corresponding memories. Our designs and analysis are inspired by the HBM technology [AMD 2015a; Standard 2013] but are also applicable to packetized die-stacked memory interfaces (e.g., Hybrid Memory Cubes—see Section 5). Two different interconnection technologies, PCIe [Lawley 2014] and NVLink [NVIDIA 2014], are considered in this work.

We use Multi2Sim 4.2 simulator [Ubal et al. 2012] as part of our evaluation framework. We modified Multi2Sim to model the SKE runtime workgroup scheduler proposed by Kim et al. [2014a] for OpenCL applications. The SKE scheduler allows multiple discrete GPUs to be viewed as a single virtual GPU by the host program. We have also extended Multi2Sim to implement our proposed unified memory hierarchy between

Table I. Workloads from the AMD APP SDK (The Number in Parentheses Indicates the Input Argument to the Benchmark)

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

the CPU and multiple GPU devices. Additionally, Multi2Sim is enhanced to model the main memory system of the CPU and the GPU devices. This is accomplished by modeling the components of the memory system, that is, controller, channels, and stacked DRAMs (banks). The NMOESI protocol is also extended to support varying block sizes in one memory hierarchy. We evaluate applications from the AMD-APP SDK benchmark suite [AMD 2016] (see Table I). This suite covers a wide spectrum of memory access patterns, optimized to AMD's Southern Islands GPUs.

3. MEMORY MANAGEMENT USING UNIFIED MEMORY HIERARCHY

As described in Section 1, 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 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 Unified Memory model, as presented by NVIDIA, performs the management through the graphics driver and CUDA runtime libraries [Harris 2013], transparent to the user. With NVIDIA UM, the data are 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 [2014] and Li et al. [2015] studied this issue using multiple micro-benchmarks.

An example is described in more detail in Figure 3. 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, then this memory transfer was not required. The second transfer copies the initialized *z* to GPU memory again, even though this value is not needed by the GPU kernel. The last redundant transfer is

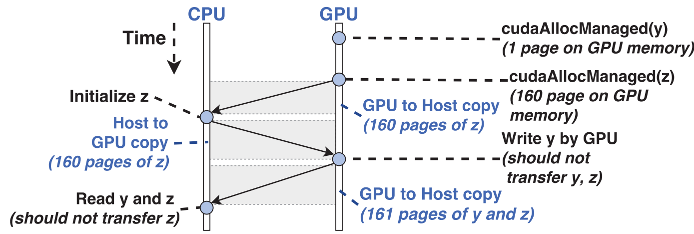


Fig. 3. The timeline of a CUDA application on a CPU-GPU system, managed by software-based UM.

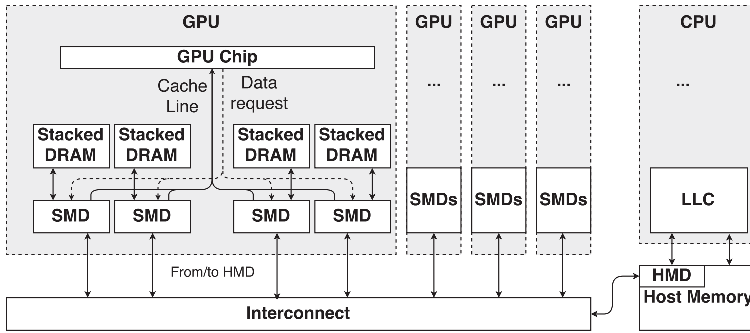


Fig. 4. Block diagram showing our Unified Memory Hierarchy design with SMDs, using four GPU devices.

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).

Here, we offer a hardware solution that can better manage memory in the system. Our hardware-based 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 4. 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), then the request is redirected to the host memory, which holds the entire dataset at the start of the application. The requested data are 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, applications (e.g., *MM, RS*, and *SC*—see Table I) 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), then a number of applications (e.g., *DCT, DH, Hist*) 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.

Our novel UMH solution pairs a 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 4 shows how SMDs are interconnected in a system with four GPUs. Our hardware UMH can resolve the issues observed in the software-based unified memory. First, the data are 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.

3.1. 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 multiGPU systems, and the following solutions are only alternatives for multi-GPU systems, but only if users manually change their host program using API calls.

3.1.1. ZeroCopy Approach for Multi-GPU Systems. Considering the *zero-copy* operations described in Section 1, a request from an 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).

3.1.2. 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 are 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. [2014a]. We compare this *memcpy* approach for CPU-multiGPU systems with our UMH approach in Section 6.

3.2. 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 [Boudier and Sellers 2011; Abramson et al. 2006] leverage an Input/Output Memory Management Unit (IOMMU) that 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 [Kim et al. 1995]. Alternatively, Pichai et al. [2014] propose dedicating a single 128-entry four-port TLB with non-blocking and PTW scheduling logic *for each CU*. Power et al. [2014] 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 the information is 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.

4. 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" [NVIDIA 2015a]. 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 [Pai 2014]) between devices, in order to maintain a consistent view of the data.

By equipping each stacked DRAM with an SMD that 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.

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* [Schaa 2014; Ubal and Kaeli 2015] 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 multicore 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.

4.1. 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 *coherency 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 II shows the complete set of possible state transitions for NMOESI using the representation introduced by Martin [2003]. Shaded cells represent the changes

Table II. State Transitions Defined by the NMOESI Protocol

	Processor action			Incoming request		
Block state	*load*	*store*	*n-store*	Eviction	Read request	Write request
N	hit	write request → **M**	hit	writeback → **I**	-	send data → **I**
M	hit	hit	hit	writeback → **I**	send data → **O**	send data → **I**
O	hit	write request → **M**	hit	writeback → **I**	send data	send data → **I**
E	hit	hit → **M**	hit → **N**	→ **I**	send data → **S**	send data → **I**
S	hit	write request → **M**	hit → **N**	→ **I**	-	→ **I**
I	read request → **S** or → **E**	write request → **M**	read request → **N**	-	-	-

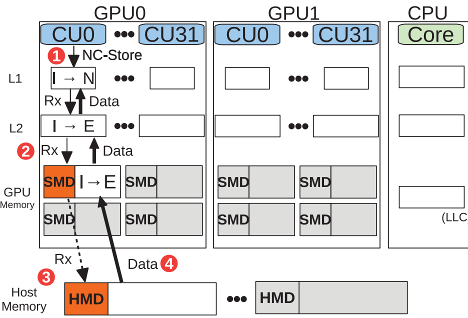
needed to support the N-state. The left column shows all possible states, the middle three columns represent actions triggered by processing memory requests, and the right three columns show requests initiated internally in the memory hierarchy.

The shaded column labeled *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 *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 the N state generates a write-back to the lower-level cache.

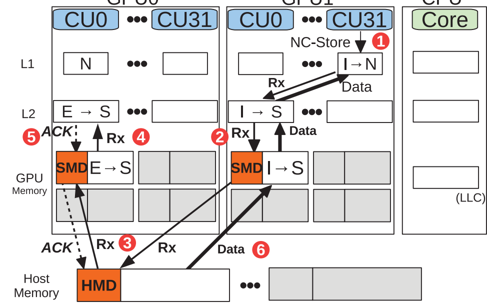
One attractive feature of the NMOESI protocol is that it is already a superset of the MOESI protocol. This means every state of the MOESI protocol (which is generally used for multicore CPUs) is already covered by the NMOESI protocol. In a multicore CPU, each cache line in the L1 and L2 caches requires 3 bits to maintain the MOESI states (five states). Supporting NMOESI on the CPU caches does not require any additional bits since the 3 bits can cover up to eight states (NMOESI has only six states). From the NMOESI prospective, 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 II 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 [AMD 2012; Schaa 2014]. 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.

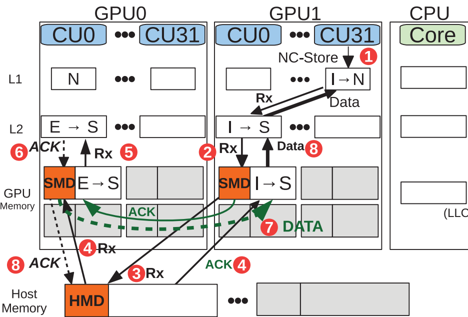
4.1.1. Example of Utilizing NMOESI in a CPU-multiGPU System. Figure 5 presents an example of how a system with two GPUs can leverage NMOESI to manage coherency between devices and the CPU. This example illustrates one of the most commonly used features of the NMOESI protocol in our work. Other NMOESI features include the following: cooperative execution, non-coherent operations for the CPU, and maintaining true coherency for the GPU [Schaa 2014]. NMOESI is utilized to support a system with any number of GPUs, based on our Unified Memory Hierarchy design. In this example,



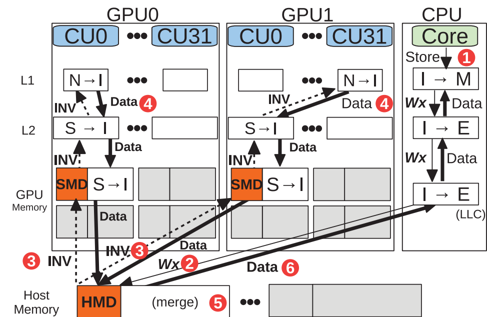
***(a) NC-store*: Read request misses in L1, L2, and SMD, and hits in host memory. Host memory has the copy of the data so it returns the data to the requesting SMD. The data are passed to L1 and modified non-coherently.**



***(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.**



***(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 are directly provided by GPU0, reducing the traffic on the host memory.**



***(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.**

Fig. 5. Leveraging NMOESI in CPU-multiGPU system.

all the compute units and cores within the GPU and CPU system request the same cache line.

As shown in Figure 5(a), *CU0* of the *GPU0* issues a non-coherent write operation for the cache line (❶). The request traverses the GPU memory hierarchy down to reach the appropriate SMD (❷). The SMD intercepts the request and performs a lookup in its directory. Since the data are not present in the GPU's memory (and hence in the directory), the request is forwarded by the SMD to host memory (❸). A second component, the *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 (❹).

As shown in Figure 5(b-1), when *CU31* of *GPU1* issues a **non-coherent** store operation to the same cache line (❶), the request is intercepted by the SMD (❷) and redirected to the appropriate HMD (❸). 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* (❹). 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 are in non-coherent state). The L2 cache in *GPU0* only responds with an acknowledgment of a receipt of the update (❺), while the HMD provides the data to *CU31* of *GPU1* (❻).

In Figure 5(b-1), the HMD device is responsible for providing the data to the requesting SMD of *GPU1* (❻) 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 that for *zero-copy*, it can still be high. Alternatively, peer-to-peer transfer between GPU devices can be leveraged (Figure 5(b-2)) to allow the GPU that owns the data (the cache line is in the *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* (❸). The HMD requests the SMD of *GPU1* to listen for data from *GPU0* (❹), and the SMD from *GPU0* ultimately provides the data to the requesting SMD (❼). 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 (❽). Our analysis suggests that on a multiGPU system with 4 GPUs, the performance improvement achieved by peer transfer is 5% on average, in comparison to a system without the peer transfer.

Figure 5(c) shows how a CPU core can access data that are modified by the GPU devices. By performing an *exclusive* store (❶), the state of the cache line changes to *Modified* or *M*. When performing this exclusive operation, a write request (❷) is made through the hierarchy to the shared host memory. The HMD holds information about the sharers of the requested address and sends invalidation ❸ to these sharers (the appropriate SMDs of the two GPUs). Each SMD forwards the invalidation request to the higher-level L1 cache (through L2), which holds the data in the non-coherent state. Each L1 cache (❹) responds to the invalidation with data through the memory hierarchy. The data are received from different GPUs and merged (❺) by the HMD using a byte-mask, which will be discussed later in this section. The data are invalidated in the cache levels of both GPUs during this process. The HMD forwards the merged data to the CPU (❻). These data are 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.

4.1.2. 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 multiGPU 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 [Standard 2013]. If the HBM is used for GPUs, then 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. This extra bit per byte amounts to 12.5% overhead in terms of DRAM memory space.

If such space is not provided, then 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 tradeoff 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 tradeoffs of representing multiple bytes per bit are discussed by Schaa [2014], and it is shown that 4 bytes per bit is a good tradeoff since the performance degradation of the entire AMDAPP SDK suite does not exceed 2%. 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%.

4.2. 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 dataset 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 were 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 3.1), the modified data in the cache hierarchy of each GPU need to be flushed (i.e., Final-Flush) to the host memory directly and do not require a copy from GPU memory to the host memory. In the *memcpy* approach (see Section 3.1), the modified data have 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. Similarly 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*.

4.2.1. 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 are 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, 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

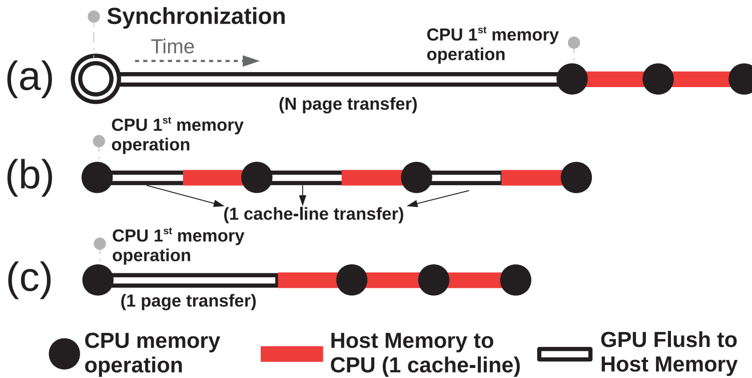


Fig. 6. The flushing mechanisms provided by NMOESI for Unified Memory Hierarchy. (a) A synchronization is performed before the CPU is allowed to use the computed data by the GPU; (b) a compiler-assisted load flushes only the single cache line of the computed data that CPU requires; (c) same as (b) but a larger granularity of the data is flushed from the memory of the GPU devices to the host memory.

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 6 shows how the CPU can retrieve data from the GPU using our unified memory hierarchy and NMOESI. Figure 6(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 6(b) shows an alternative 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 6(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 6.

5. DESIGN OF SMD AND HMD

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 are available in the stacked DRAMs, each SMD should be equipped with a Static Random-Access Memory (SRAM) lookup table. The SMD decodes the requested address and, based on the tag, locates the row buffer where the data reside. 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, then 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 [Ubal and Kaeli 2015; Schaa 2014]. In our target GPU (AMD's 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.

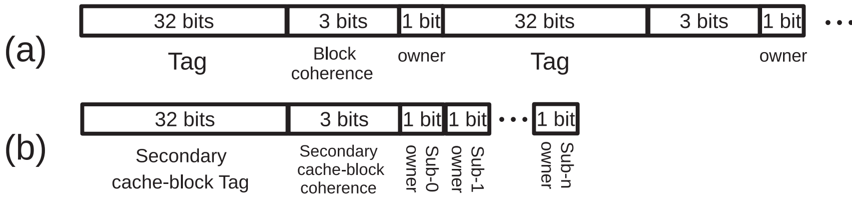


Fig. 7. Possible formats for the entries in the SMD. (a) Typically, each 64B cache line requires 36 bits of associated data, making the required space unmanageable. (b) We implement sub-blocking for a directory-based coherence protocol. A block of n 64B cache lines only requires 35 bits for the tag and coherence data, and each of the n cache lines only requires 1 bit to identify whether its owned by the upper level.

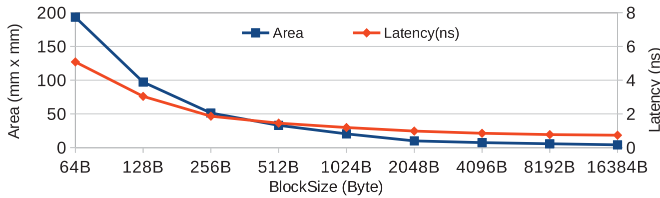


Fig. 8. The impact of selecting a larger secondary block-size on area and look-up latency of the SRAM table for 1GB stacked DRAM.

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 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 among four separate stacked DRAMs (see Section 4). So the tag is 32 bits long. Figure 7(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 [Jevdjic et al. 2014; Loh and Hill 2011; Kim et al. 2014b].

Instead of storing each cache line in the SRAM separately, we can combine cache lines in a larger secondary cache block. This technique is know as sub-blocking [Kadiyala and Bhuyan 1995]. 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 7(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 8 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 [Shivakumar and Jouppi 2001]. 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.

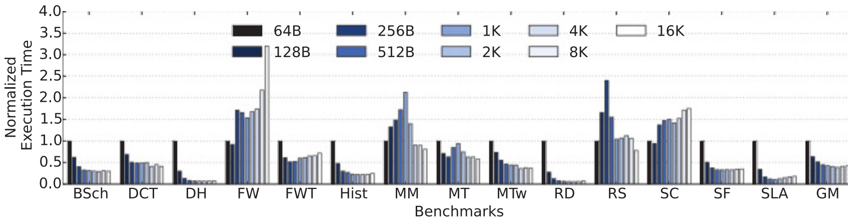


Fig. 9. Design space exploration of different secondary cache-block sizes and its impact on the performance of the CPU-multiGPU system.

Figure 9 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. 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 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 10 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 do not reside in the stacked DRAM to the lower levels of the memory hierarchy.

5.1. 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, then the access generated to HMD has to be able to trigger a page table walk in host memory. We suggest implementing a hardware PTW for the host memory, since (1) a hardware PTW will deliver 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) [Pichai et al. 2014]. So an 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

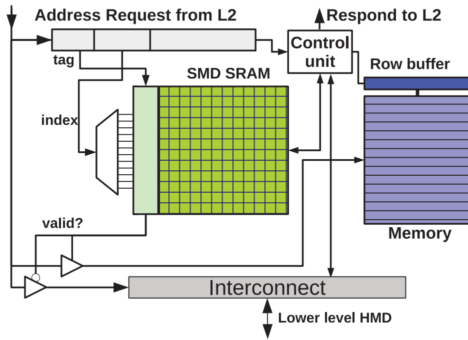


Fig. 10. Design of an SMD component for each stacked DRAM.

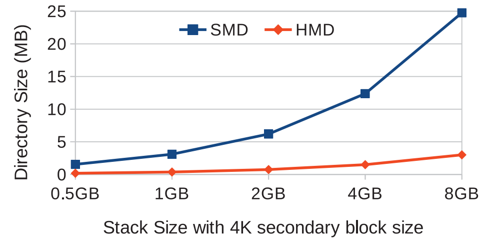


Fig. 11. The size of each HMD and SMD for different stack sizes in a system with two HMDs and four GPUs.

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 does 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.

5.2. Design of the HMD

The HMD slightly differs from the SMD in the GPUs. HMD is a SRAM-based memory, similar to the SMD design. It can either be 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 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 are partitioned between these two memory modules (and two HMDs), so half of the SMDs in the entire system (two of four 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 11 shows the required size of the HMD and SMD components (in bytes) for different stack sizes for a system with four GPUs and two 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 0.5ns, 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.

5.3. Required Modifications

Our implementation also involves two small modification to the GPU memory system. First, as shown in Figure 5, 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. 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 7mm^2, and so adding four SMD components to any chip increases the overall area of that chip by more than 28mm^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 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 12). We considered 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

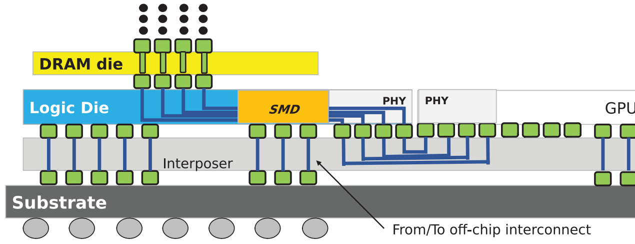


Fig. 12. The physical placement of the SMD on HBM.

Table III. Simulation Parameters

GPU specs.		GPU Memory				CPU-multiGPU UMH System	
Clock Frequency	800	Frequency	1200	# of GPUs	1/2/4	PCIe BW	16GB/s
Compute Units	16	Partitions	4	# of SMDs	4/8/16	NVLink BW	80GB/s
SIMD Width	16	DRAM controller	FR-FCFS	# of HMDs	2	Secondary Block	4KB
Workgroup Size	256	L1-L2 Interconnect	Ideal	Stack Size	1GB	Network size - 1GPU	2 × 2SMD-1HMD
Wavefront Size	64	Memory Bus Width	32B	Stack/GPU	4	Network size - 2GPU	2 × 4SMD-1HMD
Fabrication	28nm	Capacity	4GB	Die-stack Memory	GDDR5	Network size - 4GPU	2 × 8SMD-1HMD
L1 cache Size	16KB	L2 Cache Size	128KB	CPU memory	DDR4	Workload Scheduling	SKE

built-in memory controller of the stacked DRAM on the logic die and use the built-in router to communicate with off-chip devices.

6. 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 III outlines the simulation parameters considered. In order to evaluate the UMH, *zero-copy*, and *memcpy* approaches (see Section 3.1), we utilize the SKE runtime model to provide the image of a single virtual GPU for multiple GPUs [Kim et al. 2014a]. Using SKE, the workgroups from the same application are assigned to multiple GPUs in a round-robin fashion.

6.1. Performance and Scalability

Figure 13 compares the runtime of the AMDAPP SDK applications on a system with one, two, and four GPUs. For our UMH design, we support peer-transfers between the GPUs to further reduce the access overhead on the host memory (see Section 4.1). 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.

6.1.1. 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 are 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.

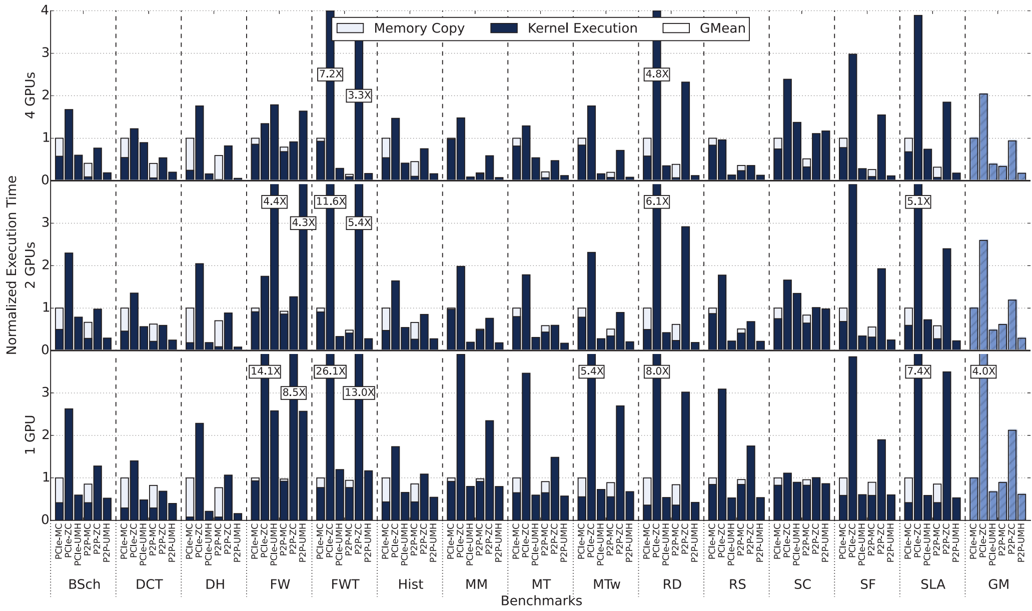


Fig. 13. Breakdown of runtime in a system with one, two, and four GPU devices, when using PCIe or NVLink peer-to-peer connections (P2P) for *mem-copy* (MC), *zero-copy* (ZC), or the UMH approach. The results for each system is normalized to the execution time of application on a system that uses MC and PCIe.

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 one, two, and four GPUs, respectively.

6.1.2. 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 four GPUs has 2.3× speedup in comparison to a similar system with one GPU. This speedup is 1.5× and 1.76× 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 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 *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 *memcpy* method is not as substantial as with the UMH approach.

6.1.3. Outliers. As shown in Figure 13, UMH outperforms all the other methods. However, in the *FW* and *SC* applications, we encounter outliers to this trend. The *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 four 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

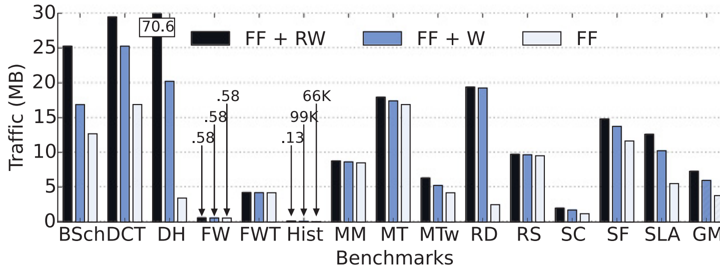


Fig. 14. The amount of data that is flushed from caches to the GPU memories during Final Flush (FF) and written back to the host memory. (R) denotes the data structures only read by GPU, and (W) denotes the data structures that GPU writes into. Software-based UM, *memcpy*, and *zero-copy* synchronize the CPU by performing FF+WR, FF+W, and FF, respectively. UMH performs dynamic synchronization.

line of that data. In this case, the interconnect becomes the bottleneck for the UMH approach. On the other hand, the *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.

6.1.4. 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 transfered over the network) but remains a necessary overhead to make the implementation of the UMH possible.

6.2. Dynamic Synchronization in UMH

As described in Section 4.2, 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, on 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 14 shows the amount of data that are flushed from the GPU and the amount that are 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 FF and then writes back the write-buffers (W) to the host memory), and (3) *zero-copy* (which only uses the FF). 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, then 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.

7. RELATED WORK

MultiGPU systems. In previous work, Kim et al. [2013, 2014a] designed a memory network with a sliced flattened butterfly topology for the interconnect between the GPUs, while 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 (and we showed here), there can be significant bandwidth demands on the CPU memory, making the pass-through path a potential bottleneck.

Coherency Protocols for APU systems. Past work has proposed novel coherency protocols for APU systems [Power et al. 2013; Sinclair et al. 2015; Komuravelli et al. 2015]. Many of these protocols can be leveraged (with some hardware modifications) to work with our UMH design (NMOESI was initially designed for APUs). In general, it is necessary for UMH to have a specific coherence protocol that takes into consideration the programming model of the GPU, including non-coherent accesses.

Reorganizing GPU memory placement. If the GPU's stacked DRAMs are placed at the same level as the CPU memory, then a flat non-uniform memory access (flat-NUMA) organization can be created to leverage the high bandwidth of the GPU memory, which will reduce bandwidth contention on the CPU memory [Bolotin et al. 2015]. But coherency can only be maintained with intelligent data migration [Agarwal et al. 2015]. 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.

DRAM cache for GPUs. A separate line of research focuses on DRAM caching in CMPs and servers [Jevdjic et al. 2013, 2014; Loh and Hill 2011; Qureshi and Loh 2012]. Similarly, Kim et al. [2014b, 2014c] 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 the 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.

8. CONCLUSIONS

In this work, 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.

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× in terms of synchronization time between the CPU and all the GPU devices. Additionally, architecting our vision of a UMH enables us to achieve a speedup of 1.92× and 5.38× (on average) over alternative *memcpy* and *zero-copy* approaches for a system with four discrete GPU devices.

REFERENCES

Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel virtualization technology for directed I/O. *Intel Technol. J.* 10, 3 (2006).

N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 354–365. DOI:http://dx.doi.org/10.1109/HPCA.2015.7056046

Nabeel Al-Saber and Milind Kulkarni. 2015. SemCache++: Semantics-aware caching for efficient multi-GPU offloading. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, 255–256. DOI:http://dx.doi.org/10.1145/2688500.2688527

AMD. 2012. AMD Graphics Cores Next (GCN) Architecture. White paper. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

AMD. 2014a. AMD Launches World's Fastest Graphics Card. Retrieved from http://www.amd.com/en-us/press-releases/Pages/fastest-graphics-card-2014apr8.aspx.

AMD. 2014b. AMD Radeon HD 7800 Series Graphic Cards. (2014). Retrieved from http://www.amd.com/en-us/products/graphics/desktop/7000/7800.

AMD. 2015a. High Bandwidth Memory. Retrieved from http://www.amd.com/en-us/innovations/software-technologies/hbm.

AMD. 2015b. High-Bandwidth Memory (HBM): Reinventing Memory Technology. (2015). https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.

AMD. 2016. AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK). Retrieved from http://developer.amd.com/sdks/amdappsdk/.

E. Bolotin, D. Nellans, O. Villa, M. O'Connor, A. Ramirez, and S. W. Keckler. 2015. Designing efficient heterogeneous memory architectures. *IEEE Micro* 35, 4 (July 2015), 60–68. DOI:http://dx.doi.org/10.1109/MM.2015.72

Pierre Boudier and Graham Sellers. 2011. MEMORY SYSTEM ON FUSION APUS: The Benefits of Zero Copy. AMD, June 2011. Web. Nov. 11 2016. http://developer.amd.com/wordpress/media/2013/06/1004_final.pdf.

Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei W. Hwu. 2015. Automatic parallelization of kernels in shared-memory multi-GPU nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, New York, NY, 3–13. DOI:http://dx.doi.org/10.1145/2751205.2751218

Patrick Dorsey. 2010. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency. *Xilinx White Paper: Virtex-7 FPGAs* (2010), 1–10.

NVIDIA Gupta, Sumit. 2015. NVIDIA Updates GPU Roadmap; Announces Pascal. Retrieved from http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 37–47. DOI:http://dx.doi.org/10.1145/1815961.1815968

Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. Springer-Verlag, Berlin, 197–208.

NVIDIA Harris, Mark. 2013. Unified Memory in CUDA 6. Retrieved from http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

Owen Harrison and John Waldron. 2007. AES encryption implementation and analysis on commodity graphics processing units. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'07)*. Springer-Verlag, Berlin, 209–226. DOI:http://dx.doi.org/10.1007/978-3-540-74735-2_15

D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 25–37. DOI:http://dx.doi.org/10.1109/MICRO.2014.51

Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 404–415. DOI:http://dx.doi.org/10.1145/2485922.2485957

M. Kadiyala and L. N. Bhuyan. 1995. A dynamic cache sub-block design to reduce false sharing. In *Proceedings of the 1995 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'95)*. 313–318. DOI:http://dx.doi.org/10.1109/ICCD.1995.528827

Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE Press, 145–156.

Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. 2014a. Multi-GPU system design with memory networks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, 484–495. DOI:http://dx.doi.org/10.1109/MICRO.2014.55.

Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. *SIGPLAN Not.* 46, 8 (Feb. 2011), 277–288. DOI:http://dx.doi.org/10.1145/2038037.1941591

Jesung Kim, Sang Lyul Min, Sanghoon Jeon, Byoungchu Ahn, Deog Kyoon Jeong, and Chong Sang Kim. 1995. U-cache: A cost-effective solution to synonym problem. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*. IEEE, 243–252.

Youngsok Kim, Jaewon Lee, Jae-Eon Jo, and Jangwoo Kim. 2014b. GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 546–557. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835963

Y. Kim, J. Lee, D. Kim, and J. Kim. 2014c. ScaleGPU: GPU architecture for memory-unaware GPU programming. *IEEE Comput. Arch. Lett.* 13, 2 (July 2014), 101–104. DOI:http://dx.doi.org/10.1109/L-CA.2013.19

R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. 2015. Stash: Have your scratchpad and cache it too. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 707–719. DOI:http://dx.doi.org/10.1145/2749469.2750374

Adam Lake. 2014. Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics. Intel 2014, Web. Nov. 11 2016. https://software.intel.com/sites/default/files/managed/f1/25/opencl-zero-copy-in-opencl-1-2.pdf.

Jason Lawley. 2014. Understanding performance of PCI express systems. Xilinx, October 28, 2014. web. Nov. 11, 2016. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

Wenqiang Li, Guanghao Jin, Xuewen Cui, and S. See. 2015. An evaluation of unified memory technology on NVIDIA GPUs. In *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 1092–1098. DOI:http://dx.doi.org/10.1109/CCGrid.2015.105

Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 454–464. DOI:http://dx.doi.org/10.1145/2155620.2155673

Milo M. K. Martin. 2003. *Token Coherence*. Ph.D. Dissertation. University of Wisconsin–Madison.

Siddharth Mohanty and Murray Cole. 2007. Autotuning wavefront applications for multicore multi-GPU hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores (PMAM'14)*. ACM, New York, NY, Article 1, 9 pages. DOI:http://dx.doi.org/10.1145/2560683.2560689

Dan Negrut. 2014. Unified Memory in CUDA 6.0: A Brief Overview. Retrieved from http://www.drdobbs.com/parallel/unified-memory-in-cuda-6-a-brief-overvie/.

A. Nere, A. Hashmi, and M. Lipasti. 2011. Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS'11)*. 906–920. DOI:http://dx.doi.org/10.1109/IPDPS.2011.88

NVIDIA. 2012. NVIDIA Maximus System Builder's Guide. Retrieved from http://www.nvidia.com/content/quadro/maximus/di-06471-001_v02.pdf.

NVIDIA. 2014. Whitepaper: Nvidia NVLink high-speed interconnect: application performance. NVIDIA Nov. 2014, web Nov. 11. 2016. http://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf.

NVIDIA. 2015a. NVIDIA CUDA C Programming Guide: Version 7.5. (2015). Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/

NVIDIA. 2015b. Tesla K80 GPU Accelerator. Retrieved from https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf.

Sreepathi Pai. 2014. Microbenchmarking Unified Memory in CUDA 6.0. Retrieved from http://users.ices.utexas.edu/sreepai/automem/.

Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. *ACM SIGPLAN Not.* 49, 4 (2014), 743–758.

Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 457–467. DOI:http://dx.doi.org/10.1145/2540708.2540747

Jonathan Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 568–578.

Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, 235–246. DOI:http://dx.doi.org/10.1109/MICRO.2012.30

Dana Schaa. 2014. *Improving the Cooperative Capability of Heterogeneous Processors*. Ph.D. Dissertation. Northeastern University.

D. Schaa and D. Kaeli. 2009. Exploring the multiple-GPU design space. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*. 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2009.5161068

Tom Shanley. 2010. *X86 Instruction Set Architecture*. Mindshare Press.

Premkishore Shivakumar and Norman P. Jouppi. 2001. *Cacti 3.0: An Integrated Cache Timing, Power, and Area Model*. Technical Report. Technical Report 2001/2, Compaq Computer Corporation.

Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*.

JEDEC Standard. 2013. High bandwidth memory (HBM) dram. *JESD235* (2013).

J. A. Stuart and J. D. Owens. 2011. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 1068–1079. DOI:http://dx.doi.org/10.1109/IPDPS.2011.102

Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

NVIDIA SuperMicro. 2016. Revolutionising High Performance Computing with Supermicro Solutions Using Nvidia Tesla. Retrieved from http://goo.gl/2YEKIq.

Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

Rafael Ubal and David Kaeli. 2015. The Multi2Sim Simulation Framework: A CPU-GPU Model for Heterogeneous Computing. Retrieved from www.multi2sim.org.

Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A case for core-assisted Bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 41–53. DOI:http://dx.doi.org/10.1145/2749469.2750399

Received May 2016; revised August 2016; accepted September 2016