GPU-SM: Shared Memory Multi-GPU Programming

Javier Cabezas
Barcelona Supercomputing Center, Spain
javier.cabezas@bsc.es

Marc Jordà
Barcelona Supercomputing Center, Spain
marc.jorda@bsc.es

Isaac Gelado
NVIDIA Corporation
Santa Clara, CA, USA
igelado@nvidia.com

Nacho Navarro
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center, Spain
nacho@ac.upc.edu
nacho.navarro@bsc.es

Wen-mei Hwu
University of Illinois at Urbana-Champaign
Urbana, IL, USA
w-hwu@illinois.edu

ABSTRACT

Discrete GPUs in modern multi-GPU systems can transparently access each other's memories through the PCIe interconnect. Future systems will improve this capability by including better GPU interconnects such as NVLink. However, remote memory access across GPUs has gone largely unnoticed among programmers, and multi-GPU systems are still programmed like distributed systems in which each GPU only accesses its own memory. This increases the complexity of the host code as programmers need to explicitly communicate data across GPU memories. In this paper we present GPU-SM, a set of guidelines to program multi-GPU systems like NUMA shared memory systems with minimal performance overheads. Using GPU-SM, data structures can be decomposed across several GPU memories and data that resides on a different GPU is accessed remotely through the PCI interconnect. The programmability benefits of the shared-memory model on GPUs are shown using a finite difference and an image filtering applications. We also present a detailed performance analysis of the PCIe interconnect and the impact of remote accesses on kernel performance. While PCIe imposes long latency and has limited bandwidth compared to the local GPU memory, we show that the highly-multithreaded GPU execution model can help reducing its costs. Evaluation of finite difference and image filtering GPU-SM implementations shows close to linear speedups on a system with 4 GPUs, with much simpler code than the original implementations (e.g., a 40% SLOC reduction in the host code of finite difference).

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems); D.1.3 [**Programming Techniques**]: Parallel Programming

General Terms

Measurement, Performance

Keywords

Shared memory machines, GPGPU, I/O interconnects

GPGPU 8 , February 7, 2015, San Francisco, CA, USA

1. INTRODUCTION

Many HPC systems install several discrete GPUs to accelerate computations rich in data parallelism, due to their outstanding computing performance [2] and energy efficiency [1]. Moreover, as CPU and GPU are integrated into the same chip (e.g., Intel Ivy Bridge [18], AMD APU [4], NVIDIA K1 [6]), multi-GPU nodes are expected to be more common in future systems. Therefore, programs need to be adapted to exploit all GPUs.

Current GPU programming models, such as CUDA [25] and OpenCL [19], make multi-GPU programming a tedious and error-prone task. These models present GPUs as external devices with their own private memory, and programmers are in charge of replicating shared data and explicitly communicating GPUs. These memory transfers must be overlapped with computation to minimize their overhead, thus increasing the complexity of the code.

Modern NVIDIA GPUs provide the capability of accessing the memories of all the GPUs connected to the same PCI Express root complex [3]. They present a single Unified Virtual Address Space (i.e., UVAS) that includes all the GPU memories in the system and the host memory. Thanks to these features, CUDA kernels can access any memory in the system through regular `load/store` instructions. While these features have been available for some time, their utilization has been mainly restricted to accelerate bulk data transfers between GPU memories and to enable better integration with I/O devices. Only a few works exploit remote memory accesses in multi-GPU computations [28]. However, we argue that they can be used in many other types of computations.

In this paper we perform an exhaustive performance analysis of remote memory accesses over PCIe and their viability as a mechanism to implement the shared memory model in

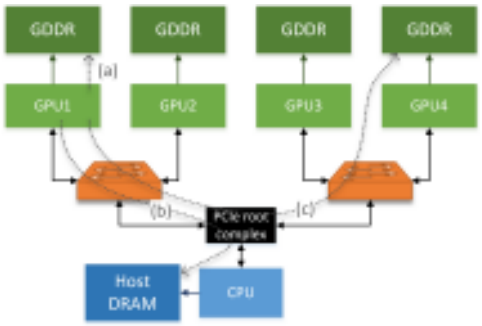


Figure 1: Multi-GPU NUMA system targeted in this paper.

multi-GPU systems. Different PCIe revisions (i.e., 2.0 and 3.0) and topologies are studied. We show that the highly-multithreaded GPU execution model helps to hide the costs of remote memory accesses. Other features introduced in the latest GPU families such as read-only caches can minimize the amount of accesses to remote GPUs. The importance of the thread block scheduling and its relation with data decomposition is also discussed, as they determine how remote memory accesses are distributed along kernel execution.

We also present GPU-SM, a set of programming guidelines for CUDA applications that define how to program multi-GPU systems using the shared memory model. Thanks to the weak memory model assumed in GPU programming models, kernel's computation grid can be safely decomposed to be run on several GPUs. Thus, thread blocks can access any data regardless its location, removing the need for replication of shared data and explicit data transfers between GPU memories. We port a finite difference and an image filtering computations to GPU-SM obtaining a performance similar to the highly-optimized original versions on a real 4-GPU system, while greatly simplifying the host source code.

The contributions of this paper are: (1) The first extensive performance analysis of the remote memory access mechanism on different multi-GPU systems. (2) A set of programming guidelines to program multiple GPUs as a shared memory system. (3) An evaluation of our proposal using high-performance implementations of the finite difference method and image filtering computations.

2. MULTI-GPU ARCHITECTURES

In this paper we focus on multi-GPU systems based on discrete GPUs since there are no systems with multiple integrated GPUs available yet. We use NVIDIA GPUs because they provide the features required to implement the shared memory model.

Our base system is composed of several discrete GPUs connected to the system through a PCI Express (i.e., PCIe) interconnect (Figure 1). Since the Fermi microarchitecture [17], NVIDIA GPUs can access other memories through the PCIe interconnect (i.e., GPUDirect[3]) without host code intervention. At the same time, they implemented a single Unified Virtual Memory Address Space (i.e., UVAS) for all the GPUs in the system. The goal of UVAS is to allow every object in the system, no matter which physical memory it resides in, to have a unique virtual address for use by application pointers. Combining the two features allows regular `load/store` instructions to transparently generate local or remote requests, based on the virtual address of the data being accessed.

GPUs access their local memory (arc a) with full-bandwidth (e.g., $\sim$ 200 GBps in GDDR5). GPUs can access other memories in the system through the PCIe interconnect: host memory (arc b) or another GPU memory (arc c). Remote accesses can traverse any PCIe switch found between the client and the server GPUs. However, in systems with multiple CPU sockets, if the target address resides in a GPU memory connected to a different root complex, the inter-CPU interconnect (HyperTransport/QPI) must be traversed, too. Unfortunately, current systems do not support routing remote GPU↔GPU requests over the inter-CPU interconnect, and we restrict our analysis to systems with a single CPU socket. Both CPU memory and the inter-GPU interconnects (e.g., PCIe 2.0/3.0) deliver a memory bandwidth which is an order of magnitude lower than the local GPU memory (e.g., $\sim$ 12 GBps in PCIe 3), thus creating a Non-Uniform Memory Access (i.e., NUMA) system. Future interconnects will help reducing the memory bandwidth gap (NVLink [5] will deliver up to 100 GB/s).

NVIDIA GPUs implement a cache memory hierarchy with a first level composed of a non-coherent private cache per SM (i.e., Streaming Multiprocessor), and a shared second level cache. L1 caches are write-through and, therefore, modifications from different SMs to the same cache line are consolidated in the L2 cache. The Kepler family of GPUs added a second private cache per SM for read-only data. Remote accesses cannot be cached in the regular memory hierarchy because modifications from different GPUs to the same cache line could produce coherence problems, but input data can be safely cached in the read-only (R/O) cache.

2.1 GPU Programming Model

GPUs are typically programmed using a Single Program Multiple Data (SPMD) programming model, such as NVIDIA CUDA [25] or OpenCL [19] (We use the CUDA naming conventions in the rest of the paper). This model allows programmers to spawn a large number of threads that execute the same program, although each thread might take a completely different control flow path. All these threads are organized into a *computation grid* of groups of threads (i.e., thread blocks). Each thread block has an identifier and each thread has an identifier within the thread block, that can be used by programmers to map the computation to the data structures. CUDA provides a weak consistency model: memory updates performed by a thread block might not be perceived by other thread blocks, except for atomic and memory fence (GPU-wide and system-wide) instructions. This is a key feature in order to implement the shared memory model: when a kernel is decomposed to be executed on multiple GPUs, the same consistency model is provided for thread blocks running on different partitions.

Each thread block is scheduled to run on an SM, and threads within a thread block are issued in fixed-length groups (i.e., warps). Each thread has its own set of private registers and threads within the same thread block can communicate through a shared user-managed scratchpad, and using synchronization instructions. The number of thread blocks that can execute concurrently on the same SM depends on the number of threads and other resources needed by each block (i.e., scratchpad memory and registers). Hence, the utilization of these resources must be carefully managed to achieve

full utilization of the GPU. In the Kepler and Maxwell families of GPUs, up to 64 warps can run concurrently on the same SM.

The GPU is a passive device that executes asynchronous commands pushed by the host code: kernel launches and memory transfers. GPUs provide several command queues which are abstracted as *CUDA streams*. Commands pushed to a stream are executed in order, but commands from different streams can execute in any order and, if there are enough available resources, concurrently. Therefore, several streams must be used to overlap computation and data transfers. Barrier operations can be also pushed to streams to guarantee inter-stream command ordering.

2.1.1 Multi-GPU Programming

In order to run applications on several GPUs, programmers typically decompose computation and data so that each GPU only accesses its local memory. If there are regions of data that are accessed by several GPUs, programmers are responsible of replicating and keeping them coherent through explicit memory transfers. Using the UVAS, programmers could achieve a continuous representation of a data structure that is distributed across several GPUs by mapping contiguous pages to alternate GPU memories. However, CUDA does not provide any means to control how virtual addresses are mapped to physical memory, and allocations are bound to a single GPU. Thus, a distributed data structure is composed of non-contiguous allocations on different GPUs.

3. EXPERIMENTAL METHODOLOGY

3.1 Hardware Setup

We use two systems with different PCIe revisions: 2.0 and 3.0.

- *System A* contains a quad-core Intel i7-930 at 2.8 GHz with 16 GB of DDR3 memory, and 3 NVIDIA Tesla K20 GPU cards with 6 GB of GDDR5 each, connected through PCIe 2.0 in x16 mode. The PCIe topology is 2+1, with two GPUs connected to the same PCIe switch, and the third one to a second switch.
- *System B* contains a quad-core Intel i7-3820 at 3.6 GHz with 64 GB of DDR3 memory, and 4 NVIDIA Tesla K40 GPU cards with 12 GB of GDDR5 each, connected through PCIe 3.0 in x16 mode. The PCIe topology is 2+2, exactly like in Figure 1.

Both machines run a GNU/Linux system, with Linux kernel 3.16 and NVIDIA driver 340.24. Benchmarks were compiled using GCC 4.8.3 and NVIDIA CUDA compiler 6.5 for GPU code. Execution times are measured using the CUPTI profiling library that provides support for sampling and nanosecond timing resolution.

3.2 Microbenchmarks

In order to characterize the hardware platform, we developed a set of microbenchmarks with different computation and memory access patterns. These microbenchmarks are implemented using a single parametrized GPU kernel that reads from memory, performs floating point operations on the input data and writes the result to memory. Each kernel launch is passed local and remote allocations for both input and output data. The available parameters are:

- Computational intensity: amount of floating point instructions performed on each input datum (FLOPs/ datum). This is implemented using an unrolled loop of dependent operations.
- Amount of remote accesses: percentage of remote accesses relative to the total amount of memory accesses. Threads use their identifiers to determine if they need to access data locally or remotely.
- Remote access pattern: whether accesses are performed by a small set of thread blocks that are scheduled for execution together (*batch*), or they are spread among a bigger set of thread blocks that are executed along with thread blocks that do not perform remote accesses (*spread*).
- Topology: whether remote accesses are performed between GPUs connected to the same or different PCIe switches, or to host memory.
- Occupancy: percentage of warps that execute concurrently on each SM relative to the maximum (i.e., 64). Theoretically, the bigger the occupancy, the more remote accesses can be overlapped with the execution of other threads. We control the occupancy by setting artificial scratchpad memory requirements in the kernel.
- Caching: remote accesses are not cached in the local cache hierarchy. However, remote accesses to read-only data structures can be cached in the private L1 R/ O cache in the SM by using the `__ldg` CUDA intrinsic.

3.3 Applications

We use two applications in order to test the performance of a GPU-based shared memory implementation: Finite Difference and Image Filtering. The original and modified implementations are discussed in detail in Section 6.

3.3.1 Finite Difference

A finite difference method is an iterative process on volumetric data that represents a physical space for simulating a phenomenon described by differential equations, often involving large data sets that can benefit from GPU acceleration [15, 10]. Domain decomposition assigns a portion of the input and output data (i.e., domain) to each GPU in the system.

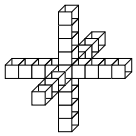


Figure 2: Data dependences in a 4-point 3D stencil computation.

The main loop of finite difference iterates over the steps of a simulation. For each step, the value of each point in the output volume is calculated by a stencil computation, illustrated in Figure 2, that takes as input the value at the point and its neighbors. The output volume of the current step becomes the input for the next step. The stencil computation for the points at the boundaries of a partition (i.e.,

Table 1: Analyzed finite difference configurations.

Halo	FLOPs	Occupancy	Volume Size	Grid size	% Remote
2	18	75%	128^3	(4, 32)	6.25%
			512^3	(16, 128)	0.78%
			768^3	(64, 512)	0.52%
4	36	62.5%	128^3	(4, 32)	12.5%
			512^3	(16, 128)	1.56%
			768^3	(64, 512)	1.04%
8	72	56.2%	128^3	(4, 32)	25%
			512^3	(16, 128)	3.12%
			768^3	(64, 512)	2.08%
16	144	31.2%	128^3	(4, 32)	50%
			512^3	(16, 128)	6.25%
			768^3	(64, 512)	4.17%

boundary data) requires input values from neighboring partitions (i.e., halo data).

We run different configurations of the finite difference computation, that are summarized in Table 1. Different volume size and halo sizes are tested. Increasing the halo size increases the amount of floating point operations (FLOPs) performed per output point, too. The amount of remote memory accesses depends on the volume and halo sizes and range from 0.52% to 50%. Besides, we analyze computation decompositions on dimensions X, Y and Z.

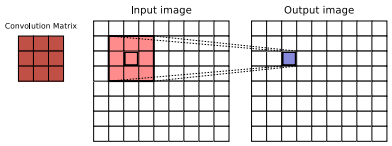


Figure 3: Data needed to compute an output pixel in a convolution.

3.3.2 *Image Filtering*

Image filtering allows to apply different effects that emphasize or remove features from images. It is typically performed in the spatial domain by using a convolution computation, or in the frequency domain by using fast Fourier transform (i.e., FFT). We use the convolution as it is the most efficient method on GPUS [16]. The convolution combines a small convolution matrix (e.g., 3×3 or 5×5) with each input pixel and its neighbors, by multiplying the value of the pixel and its corresponding element of the convolution matrix (Figure 3). The value of the output pixel is the sum of all the individual products. Domain decomposition assigns a portion of the image to each GPU. Like in the stencil, the computation pattern creates a halo of data that is required to compute the output value for the pixels in the boundaries. Moreover, the convolution matrix is completely accessed by each GPU in the system. The data sets and filter sizes used in the evaluation are summarized in Table 2.

4. ANALYSIS OF THE REMOTE ACCESS MECHANISM

In this section we study the characteristics of remote accesses and how they impact on the performance of applications.

Table 2: Analyzed image filtering configurations.

Filter	FLOPs	Occupancy	Image Size	Grid size	% Remote
3×3	18	100%	128^2	(4, 32)	81.27%
			4096^2	(128, 1024)	80.91%
			24576^2	(768, 3072)	80.90%
5×5	36	75%	128^2	(4, 32)	89.58%
			4096^2	(128, 1024)	89.30%
			24576^2	(768, 3072)	89.29%

Hypothesis 1 *Remote accesses in GPUs can achieve full PCIe bandwidth. Since PCIe is full-duplex, it can sustain the bandwidth for R/W concurrent remote accesses or accesses of the same type from two different GPUs.*

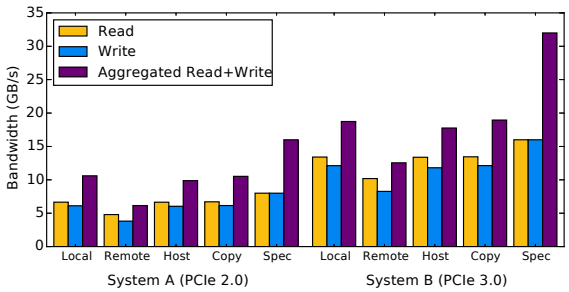


Figure 4: Memory bandwidth achieved by remote accesses for different PCIe generations. *spec* is the theoretical peak.

We use a microbenchmark that only performs remote memory accesses to obtain the maximum effective bandwidth. Two different variables are explored:

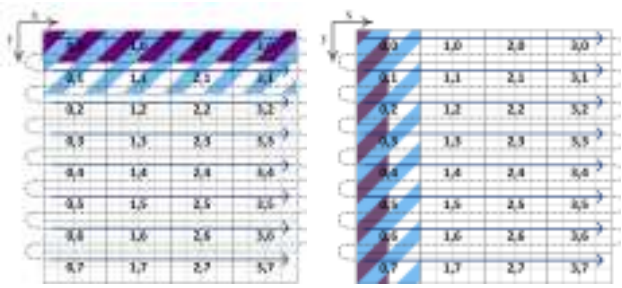
1. Number of concurrent transfers: running a kernel that reads or writes a remote memory we can obtain the effective peak remote memory bandwidth. If the kernel performs both read and write accesses we obtain the aggregated remote memory bandwidth.

2. Topology: crossing PCIe switches to reach the destination GPU may increase the access latency and impact on the achieved bandwidth. *local* label indicates that no PCIe switches are crossed, while *remote* indicates that one PCIe switch is crossed. *host* indicates that remote data is stored in host memory.

Results in Figure 4 show the measured bandwidth on our two test systems. *copy* bars show the measured bandwidth using `cudaMemcpy` instead of remote memory accesses and *spec* show the maximum theoretical bandwidth offered by the interconnect. A single GPU can achieve the same memory bandwidth as bulk memory transfers by using remote memory accesses (>6 GBps for PCIe 2.0 and >12 GBps for PCIe 3.0). Write accesses exhibit 8-10% lower bandwidth than read accesses. When one GPU performs read and write remote accesses concurrently or two different GPUs concurrently perform the same type of remote accesses (read or write), the measured aggregated memory bandwidth (over 10 GBps for PCIe 2.0 and over 18 GBps for PCIe 3.0) is higher than the peak memory bandwidth of a single GPU, although not twice. Crossing PCIe switches imposes a noticeable overhead, especially for write accesses (>20% for

reads, >30% for writes). The bandwidth of remote accesses to host memory is similar to that of accesses to a GPU connected to the same PCIe switch. The rest of experiments in this section are run on System B (PCIe 3.0).

Hypothesis 2 *Temporal distribution of remote memory accesses determine their effect on the kernel's performance. If all remote accesses are concentrated in the same phase of the kernel, it is more difficult to hide their costs as there is no other work that can be executed.*



(a) Rows (b) Columns

Figure 5: 2D kernel computation grids in which 20% (light blue) of the input and 10% (dark purple) of the output elements of the matrices are accessed remotely. In (a), the first rows are remote. In (b), the first columns are remote. Labels indicate the x, y indices of the thread blocks within the computation grid. Arrows indicate the order in which thread blocks are executed in our GPUs.

The distribution of remote accesses is determined by the chosen data decomposition and the thread block scheduling policy. In current NVIDIA GPUs, thread blocks are scheduled for execution linearly following their identifier within the computation grid, from the lowest-order to the highest-order dimensions.

We execute a microbenchmark that uses a 2D computation grid in which each thread reads an element from an input matrix, performs 10 FLOPS on the element and writes the result to an output matrix. 20% of reads, and 10% of writes are remote. We use the indices in the two different dimensions of the element being accessed by a thread to determine which accesses are remote, as shown in Figure 5. Two different data decompositions are emulated: (1) rows of the matrix are remote (Figure 5a), (2) columns of the matrix are remote (Figure 5b).

Figure 6 shows the distribution of remote accesses across the kernel execution time for the two different configurations. When rows are remote (top), we see that all read and write remote accesses are performed at the beginning of the kernel. This is because remote accesses are concentrated in a set of contiguous thread blocks that are executed at the same time. The achieved bandwidth is close to the peak bandwidth of the system measured in the previous experiment. When columns are remote (bottom), we see that remote accesses are spread through the whole kernel execution. In this case, thread blocks that perform remote accesses are executed concurrently with other thread blocks that do not. Thus, the requested remote bandwidth is lower than the peak.

Lines with circles show the average IPC (i.e., instructions per cycle) per SM in the kernel. They show that requesting

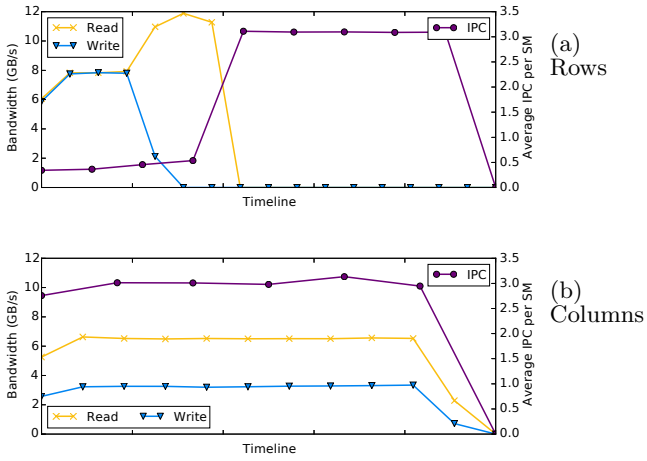


Figure 6: Mesured remote read and write access bandwidth during the execution of the configurations depticted in Figure 5. Lines with circles indicate the average IPC per SM (right axis).

a remote memory bandwidth close to the peak (rows) hugely impacts on the performance of the kernel, while lower bandwidth requirements (columns) can be sustained through kernel execution with almost no impact on the performance.

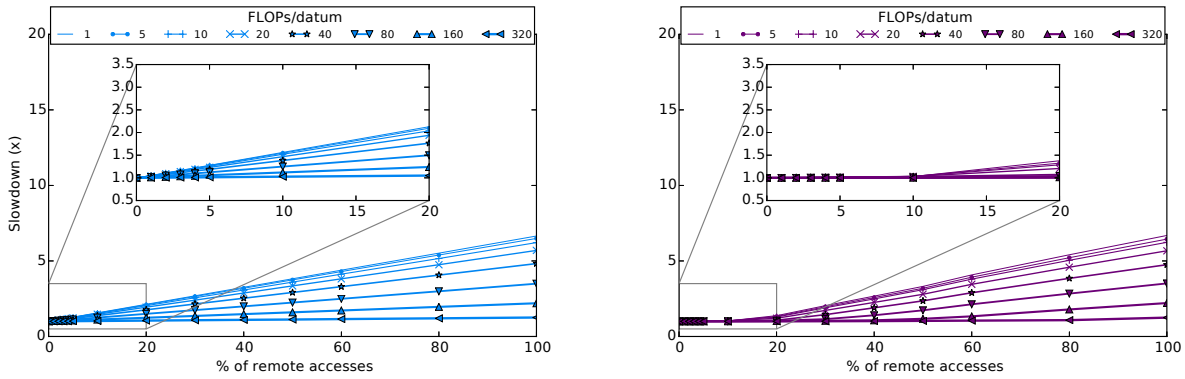
These results confirm our hypothesis and indicate that the thread block scheduling policy must be taken into account when choosing the thread blocks to perform remote memory accesses.

Hypothesis 3 *Compute-bound computations suffer from less performance degradation than memory-bound computations due to remote memory accesses.*

We execute a number of GPU kernels that explore the following variables: (1) Amount of remote memory accesses relative to the total of memory accesses, (2) amount of operations performed per input datum to encompass from completely compute-bound to memory-bound computation patterns, and (3) remote memory accesses distribution: *batch* or *spread*.

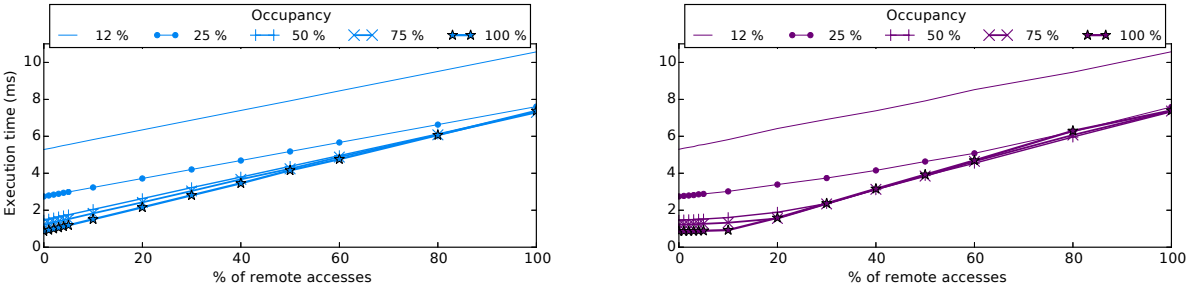
Figure 7 shows the overhead suffered by each of the GPU kernel configurations due to remote memory accesses. The overhead increases with the number of remote memory accesses (up to 7x slowdown), as expected. Compute-bound computations (large FLOPS/datum values) are less affected than memory-bound computations. The distribution of remote memory accesses is key in order to hide their overhead. *batch* configuration (Figure 7a) shows a degradation of the performance that grows linearly with the amount of remote memory accesses. On the other hand, *spread* configuration (Figure 7b) shows in the zoomed section that values up to 10% of remote memory accesses result in less than 10% overhead for all computation intensities (including completely memory-bound computations). The overhead increases to 45% for the configuration with 20% of remote accesses and, for greater values, the overhead increases linearly, showing that the cost of remote accesses cannot be hidden any longer.

We can say that current GPUs and interconnects are able to hide the costs of 10% of remote accesses almost completely. Future interconnects such as NVLink will likely increase this number.



(a) Batch (b) Spread

Figure 7: Performance overhead imposed by remote accesses for different computation intensities. Figure on the left shows the overhead when all remote accesses are concentrated in time (thread blocks are scheduled together). Figure on the right shows the overhead when remote accesses are evenly distributed in time.



(a) Batch (b) Spread

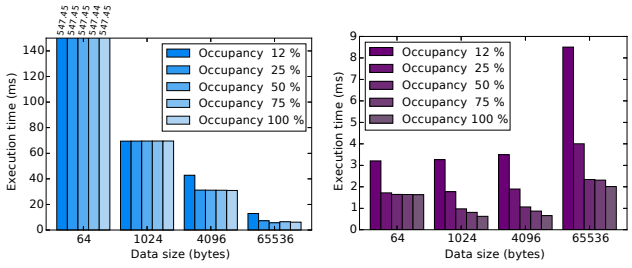
Figure 8: Execution time for different remote accesses and SM occupancies. In (a) all remote accesses are concentrated in time (thread blocks are scheduled together). In (b) remote accesses are evenly distributed in time.

Hypothesis 4 *The GPU execution model helps to hide the costs of remote memory accesses by executing work from other warps.*

While Figure 7b already shows that the GPU execution model can hide the costs of a certain amount of remote memory accesses, we perform a more detailed analysis. Results in that figure assume maximum thread block concurrency (i.e., occupancy) in the SM. However, the amount of thread blocks that can be concurrently executed on an SM is limited by the amount of resources used by each thread block. Many algorithms ported to GPU cannot reach the maximum occupancy, although most of them keep it high enough to be able to hide memory latency [25].

We run the same GPU kernels used in the previous experiments, but we artificially modify the amount of scratchpad memory used per thread block, thus lowering the occupancy in the SMs. Results in Figure 8 show the execution time of the GPU kernel configuration that performs 10 FLOPs/datum for a different amount of remote memory accesses, and different SM occupancies and remote memory access distributions. Figure 8a (using *batch* distribution) shows that the execution time of the kernels increases linearly, and no occupancy is able to hide the costs of remote memory accesses. Figure 8b (using *spread* distribution) shows that occupancies starting at 25% are able to hide the costs of remote accesses and the execution time is lower than the *batch* distribution even for 60% of remote memory accesses, thus confirming our hypothesis.

Hypothesis 5 *The overhead of remote accesses to small read/only data structures can be minimized using caching.*



(a) Uncached (b) Cached

Figure 9: Execution time for different sizes of the remotely accessed read-only data structure and different occupancies. Note the different Y scales.

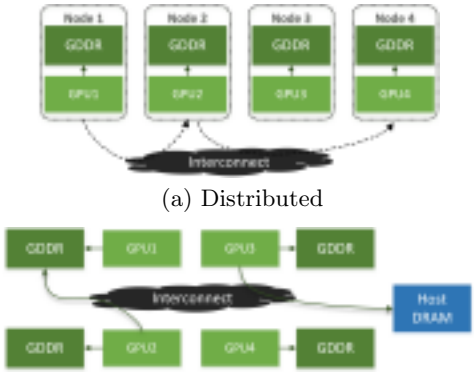
In the previous experiments, each thread reads different elements from either the remote or local allocations. However, oftentimes, all threads access the same elements of

input data structures. For example, in the matrix-vector multiplication, every thread (or warp, depending on the implementation) accesses the whole input vector to compute the output element. Another example is the 2D convolution, in which each thread accesses all the elements in the small convolution matrix and combines them with the input point and its neighbors to compute the output point.

We run again the microbenchmarks for the configuration in which 100% of input data accesses are remote and the code performs 10 FLOPS/datum. But this time we limit the size of the input data so that there are elements that are read by several threads. Since the size of the computation grid is constant across kernel configurations, the smaller the input data structure, the more threads read each of its elements. Figure 9 shows the execution time of the benchmarks for different SM occupancies. Results show that when caching is not enabled, remote accesses to a small data structure impose a huge overhead and high occupancy values do not help hiding the latency. Interestingly, the smaller the data structure, the higher the overhead.

Caching is effective for really small data structures as performance starts to degrade when the whole structure does not fit in the cache (the size of the R/O cache is 48 KB). The 64-byte configuration is also slower because it is smaller than the cache line size (required to achieve full bandwidth).

5. GPU-SM



(a) Distributed

(b) Shared (GPU-SM)

Figure 10: Multi-GPU system models. Solid lines indicate access through load/store instructions; dashed lines, bulk DMA transfers.

Given the results obtained in the previous section, we propose a set of programming practices to program multi-GPU systems as a shared memory machine.

5.1 Distributed Memory vs Shared Memory

The current approach to multi-GPU programming is to use GPUs as nodes of a distributed system (Figure 10a). Thus, programmers decompose computation and data and distribute them across GPUs. Data shared among different computation partitions is replicated, and outputs need to be gathered and, if they overlap, they need to be merged. In computations with data dependencies across computation partitions, some data generated in one GPU needs to be explicitly transferred to other GPU memories before next kernels can proceed. These copies perform bulk DMA transfers across the interconnect. However, these communications can introduce a large overhead, and techniques to overlap computation with communication are commonly used to reduce it, at the cost of increased code complexity.

We propose changing the perspective of how a multi-GPU system is programmed by looking at the GPUs and their memories as if they were a shared memory NUMA system (Figure 10b), which we call GPU-SM. Any GPU can access all the memories using regular load/store operations, thanks to the UVAS and remote memory access capabilities introduced in Section 2. Thus, programmers can freely allocate data in any of the memories. Nevertheless, the overhead imposed by remote memory accesses can be large, and data should be placed in such a way that they are minimized.

5.2 Writing Code for GPU-SM

When a GPU kernel is launched, a grid of thread blocks is instantiated and a hardware scheduler distributes them for execution among the SMs in the GPU. Since GPUs can now access all memories in the system, executing a GPU kernel across all the GPUs in the system could be as simple as aggregating all their SMs and memories. Thus, the scheduler would just need to distribute the thread blocks among a larger amount of SMs. However, while a GPU can contain several SMs, it is exposed as a single compute unit, and current schedulers cannot issue thread blocks to a different GPU. Hence, programmers have to decompose the computation grid into partitions and launch each partition on a different GPU.

The computation grid is a multidimensional space of thread blocks of up to 3 dimensions ($grid_x \times grid_y \times grid_z$). Programmers can decompose the grid on any of their dimensions (or a combination of them). The dimensions being decomposed determine how data structures must be partitioned and distributed. This is because programmers usually apply affine transformations to the block and thread indices to compute the indices that are used to access data structures. As a result, threads that belong to blocks with contiguous identifiers in one dimension tend to access elements that are contiguous in the same or a different dimension of the referenced data structure.

We now discuss important trade-offs when using GPU-SM.

Computation grid decomposition.

When thread blocks access mostly non-overlapping regions of a data structure, it can be decomposed in such a way that a data partition is predominantly accessed by a single partition of the computation grid. If there is overlap between accessed data regions, the chosen dimensions for the computation grid decomposition determine the pattern of the remote memory accesses. This is caused by the thread block scheduling policy in the GPU. For example, consider a 2D computation grid in which each thread uses its linear indices in X and Y dimensions to access a matrix, reading an input element and the k neighboring elements in the X and Y dimensions. This computation pattern creates a *halo* of elements that are needed by the threads in the edges of the thread block. Thus, neighboring thread blocks access the elements in this halo of points and, therefore, the thread blocks in the boundaries of the computation partitions will need to access data that is located in a different memory. Depending on the dimension being decomposed, accesses to this data arrive with different distributions. When the X di-

mension is decomposed, some columns of the input matrix are accessed remotely. Therefore, accesses will be performed by thread blocks that are evenly distributed in the kernel execution time. On the other hand, when the Y dimension is decomposed, some rows of are accessed remotely, which creates a single big batch of remote access and is more likely to harm the performance of the kernel. As we have seen in Section 4, decomposing the X dimension would be better in this case.

Replication versus caching.

Another common memory access pattern is produced when every thread traverses a whole input data structure. The distributed approach would use replication to eliminate the need for remote accesses, at the cost of increased memory usage and code complexity. But, as we have seen, small input data structures can be efficiently cached using the `__ldg` intrinsic. Therefore, replication can be avoided in kernels in which input data structures fit in the L1 R/O cache (48 KB in Kepler GPUs). If other data structures in the kernel also use `__ldg`, a lower upper limit should be used.

Output data structures.

Replicating output data structures implies that copies need to be merged after kernel execution in order to have a consistent version in all GPUs. This step imposes an overhead that, in most of the cases, is larger than the costs of using remote updates. Moreover, code complexity greatly increases in order to implement merging efficiently. Furthermore, GPUs perform better with gather memory access patterns [27], which minimize the number of write operations. Therefore, replication of output data should only be used if remote writes limit kernel performance.

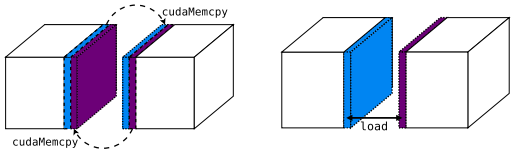
Memory fences.

GPUs implement several memory fence instructions that work with different granularities: block-level, GPU-level, and system-level. In the first case, the calling thread waits until all its writes to memory are visible to all threads in the thread block. The other two extend the visibility of the writes to the threads of all the thread blocks in a GPU or in the system (all GPUs), respectively. GPU-level and system-level fences are provided to enable synchronization across thread blocks in a kernel or across kernels launched on different GPUs. In the typical distributed system model, a GPU only accesses data stored in its memory. In the shared memory model, computations decomposed to be executed across several GPUs may access data stored remotely. Thus, if the kernel contained GPU-level memory fences, they need to be upgraded to system-level memory fences since all the kernel partitions executed on each GPU work on the same data.

5.2.1 Current Limitations

Virtual memory.

CPU-based shared memory systems use VM mechanisms such as page-fault handling to transparently detect the processors that request data. Thus, allocation [11, 12], replication [14] and migration [24] policies can be transparently implemented to minimize the amount of remote memory accesses. Unfortunately, GPUs do not implement such VM mechanisms yet and programmers perform data placement manually. However, vendors have announced true VM capabilities such as page-fault for future GPUs, which will allow for automatic memory placement policies.



(a) Distributed (b) GPU-SM

Figure 11: Inter-domain data dependences for finite difference.

The CUDA API does not provide the functionality to allow programmers manage the virtual addresses in the UVAS. This feature would enable transparent mapping of contiguous pages to alternate physical memories which, in turn, would make data distribution completely transparent to the kernel code. Currently, different allocations must be used to distribute data across different GPU memories, and the code must be aware of the different allocations and choose the appropriate pointer. Nevertheless, we consider that this limitation will be fixed in the future.

Atomics.

GPUs support atomic operations, which are implemented in the L2 cache. Since accesses to remote data cannot be cached in the common case, data resides in the GPU cache whose memory contains the accessed data, only. This limitation simplifies the support of atomic instructions across GPUs since they only need to be forwarded to the GPU that owns the data. PCIe 3.0 does support atomic operations but current NVIDIA GPUs do not forward atomic operations to remote GPUs. Thus, we cannot port kernels that use atomic operations to GPU-SM using current GPUs.

6. IMPLEMENTATION AND EVALUATION OF GPU-SM APPLICATIONS

We port two applications from its original distributed model implementation to GPU-SM.

6.1 Finite Difference

Distributed memory implementation.

In this implementation, the halo data that is shared between domains is replicated in each of the GPU memories (Figure 11a). This implies that GPUs exchange data between simulation steps, so exchange efficiency is critical to the scalability of such applications.

The GPU kernel uses a 2D computation grid that is mapped on the front XY plane of the volume. Each thread iterates through all the planes, computing a single output element for each plane. The implementation used in our tests is based on [22], which uses register tiling in the Z dimension of the array, and shared memory to reuse the elements read by neighboring threads in the X and Y dimensions. Due to the length of the optimized version of the kernel, we show a simplified version in Listing 1. The same GPU kernel can be used in both single- and multi- GPU versions, as each computation partition accesses data stored in its own GPU memory.

Listing 2 shows an optimized version of the host code that decomposes the simulated domain on its X dimension. It

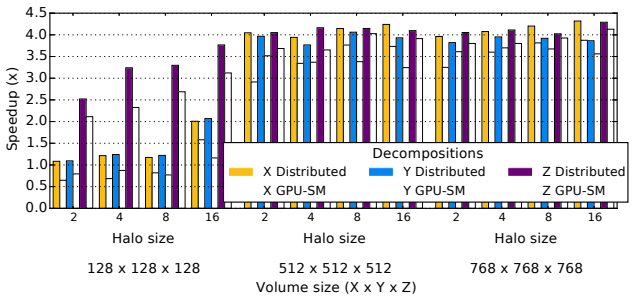


Figure 12: Finite difference: speedups of the multi-GPU implementations for 4 GPUs compared to the original implementation on a single GPU.

is much more complex than the single-GPU version and it heavily relies on CUDA abstractions such as streams and events in order to overlap computation and data transfers. Each iteration of the simulation is divided in three main phases.

1. Compute boundaries (lines 16–33): kernels are launched to compute the points in the boundaries, so that they can be communicated as soon as possible. Kernels are queued into different streams so they can be concurrently executed on the GPU.

2. Compute center (lines 35–41): a kernel is launched on a third stream to compute the rest of the points in the subdomain.

3. Exchange boundaries (lines 43–58): then, memory transfers are queued in the two first streams so that they start as soon as the kernels that compute the points finish.

These steps are repeated for each GPU in the system. Each operation is tracked using an event. Event barriers are also pushed to streams before each of the listed steps thus preventing operations of the current time step to start before the operations launched in the previous time step have consumed or produced the required data.

GPU-SM implementation.

Using GPU-SM, halo data does not need to be replicated in GPU memories and it is accessed through remote accesses (Figure 11b). Since CUDA currently does not allow us to transparently distribute the volumes by mapping pages on alternate GPUs, we use one allocation per GPU and the kernel is modified to be aware of the different allocations. Thus, threads that compute the elements in the boundaries use the pointers to the remote allocations (lines 13 and 16 in Listing 3). Without this restriction the kernel code would be the same as in Listing 1. On the other hand, the host code is much simpler than the distributed version since there are no data transfers, a single kernel launch computes the whole domain (lines 27–30 in Listing 4), and only one stream per GPU is used.

6.1.1 Performance Analysis

Figure 12 shows the speedup achieved by running the different implementations of the finite difference benchmark on 4 GPUs, compared to the original implementation on a single GPU. In the configuration with the smallest volume, the Z decomposition is the best one for both distributed and GPU-SM implementations. This is because the computation grid is very small (32×4) and it is further reduced (32 blocks per GPU) when the computation is decomposed on the X or Y dimensions, which leads to the underutilization of the SMs. When decomposed on the Z dimension, threads iterate over a smaller number of planes, but the computation grid it is not decomposed, and more thread blocks can run on each GPU concurrently. Bigger volume sizes produce higher speedups for all the configurations due to a larger number of thread blocks and a lower relative amount of remote accesses. Z remains as the best decomposition for GPU-SM implementations, and provides $> 3.60\times$ speedups for the $512\times512\times512$ volume and $> 3.80\times$ for the $768 \times 768 \times 768$. Compared to the distributed implementation, the average overhead due to remote accesses is 12% for the $512 \times 512 \times 512$ volume, and 8% for the $768 \times 768 \times 768$ one.

Figure 13 shows the execution timelines of the benchmark configuration with $768 \times 768 \times 768$ volume size, 16 halo size, for the X, Y and Z decompositions. The timelines show the remote memory access throughput and the average IPC per SM of the kernel for both the distributed and the GPU-SM versions. Results in Figure 12 indicate that the best decomposition for this configuration is Z, followed by X and Y. Note that the X and Z decompositions exhibit higher IPC than Y even for the distributed implementation, due to better spatial locality. The Y decomposition (rows) suffers a sharp drop in IPC due to the high remote access throughput (up to 8 GBps) of the thread blocks that execute at the beginning and the end of the kernel. The rate of remote accesses is constant across the execution of the X decomposition, while the Z decomposition shows a jagged pattern. The IPC of X degrades over time (because thread blocks performing remote accesses accumulate), while in Z all thread blocks perform remote accesses but only for a small period of time (the few first/last planes in the domain) that can be hidden with the execution of other thread blocks.

6.2 Image Filtering

Distributed memory implementation.

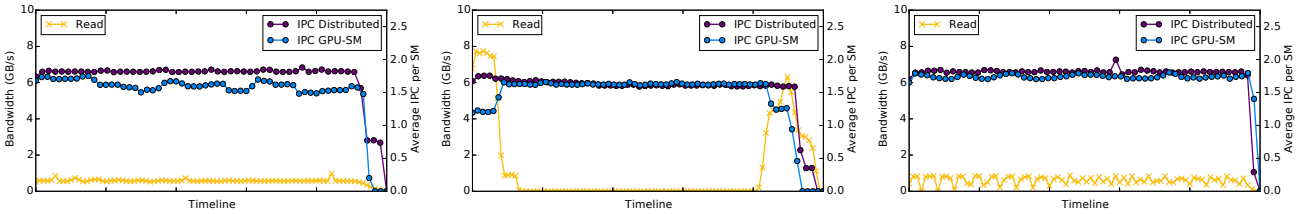
The traditional implementation of convolution for multiple GPUs replicates the halos of the partitions of the input image in each GPU. The convolution matrix is replicated in all GPU memories. The output image does not require replication because computation partitions write in non-overlapping regions.

GPU-SM implementation.

Thanks to shared memory, halo data for the input image partitions can be remotely accessed. With respect to the convolution matrix, it cannot be decomposed because it is fully accessed by every thread in each computation partition. We study two possibilities: (1) replicating it like in the distributed implementation, or (2) storing it in a single GPU or in host memory and accessing it remotely. We also evaluate the impact of caching it, as it is small enough to fit in the L1 R/O cache.

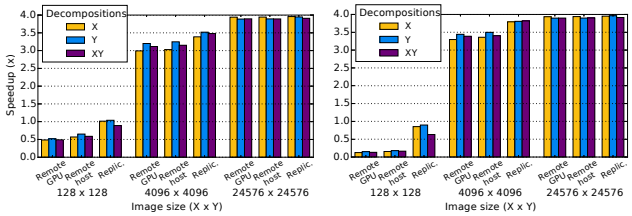
6.2.1 Performance Analysis

Figure 14 shows the speedup achieved by running the GPU-SM implementation of the convolution kernel on 4 GPUs. Like in the finite difference benchmark, the small-



(a) X (columns) (b) Y (rows) (c) Z (planes)

Figure 13: Finite difference: execution timeline for different decompostion configurations. Line with crosses indicate the achieved remote bandwidth. Lines with circles indicate the average IPC per SM.



(a) 3×3 (b) 5×5

Figure 14: Image filtering: speedups of the GPU-SM implementation for 4 GPUs compared to the original implementation on a single GPU (for two convolution matrix sizes).

est input data set is too small to benefit from multi-GPU execution. The bars show the results for the configurations that use the L1 R/O cache. Not using this cache introduced up to 1000× slowdowns when accessed remotely as each thread performs reads every element in the matrix (9 or 25 in our benchmarks), and each access triggers a remote memory access. Further, using the R/O cache is also a 35% faster than the regular cache hierarchy when the convolution matrix is replicated. Results for the medium-size image indicate that there is around a 10% performance loss when not using replication and resorting to caching to access the convolution matrix for both the 3×3 and 5×5 convolution matrix sizes (storing it in host memory instead of a GPU memory improves the results by 2%). This is because all the thread blocks in all SMs (> 100) are stalled at the beginning of the kernel until each of the SMs brings the matrix to its L1 R/O cache and, therefore, this cost cannot be hidden. Having a L2 shared R/O cache would help minimizing the performance overhead as only the first SM would need to bring the matrix and the rest of SMs could access cached data. For larger matrix sizes these initial costs are negligible and we achieve linear speedups.

7. RELATED WORK

Schaa et al. [26] analyze the performance and programmability of multi-GPU systems and Wong et al. [29] analyze the microarchitecture of NVIDIA GPUs through microbenchmarking. However, they use GPUs that do not provide the remote memory access mechanism that enables the shared memory programming model. Tanasic et al. [28] exploit remote memory accesses to compute the size of the partitions to be communicated in multi-GPU sorting.

Shared memory machines have been extensively studied in the past. They can be Symmetric Multi-Processing (SMP), such as Sun Enterprise series or Intel Pentium Pro powered machines [13]. A more scalable approach, and the one dominant in small scale parallel machines today is Non-Uniform Memory Access (NUMA) machines. Some well known examples of the cache coherent NUMA (ccNUMA) type machines are the MIT Alewife [7], Stanford DASH [21], and modern multi-socket machines built with Intel CPUs [23]. Alternatively, The Cray T3E machine is a non-coherent shared memory multiprocessor [9], more similar to the multi-GPU systems that we target in this paper. Shared memory machines implement various latency tolerance mechanisms, such as data prefetching or multithreading [20], in order to hide the extra cost of accessing remote memory. For example, the APRIL processor [8] from the Alewife machine [7] switches between threads on each remote memory request or failed synchronization attempt, similarly to GPUs.

8. CONCLUSIONS

In this paper we show that the remote memory access mechanism enables easier multi-GPU programming. While PCIe offers limited bandwidth compared to GPU memories, GPU's ability to hide long latency accesses allows it to tolerate a moderate amount of remote accesses. Other features like the read-only cache in latest NVIDIA GPUs can also be used to minimize the amount of remote memory requests. We also show that shared memory implementations of finite difference and image filtering computations deliver a performance comparable to their highly-optimized distributed counterparts, while being simpler and much more concise. Future interconnects promise higher bandwidths that will open the GPU-SM model to a broader range of applications.

9. ACKNOWLEDGEMENTS

This work is supported by NVIDIA through the UPC/BSC CUDA Center of Excellence, Spanish Government through Programa Severo Ochoa (SEV-2011-0067) and Spanish Ministry of Science and Technology through TIN2012-34557 project.

10. REFERENCES

[1] Green500 list - June 2014. http://www.green500.org/lists/green201406/.
[2] TOP500 list - November 2014. http://top500.org/list/2014/11/.
[3] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2012.

[4] APU 101: All about AMD Fusion Accelerated Processing Units, 2013.

[5] NVIDIA NVLINK high-speed interconnect. http://www.nvidia.com/object/nvlink.html, 2014.

[6] Tegra K1 Next-Gen Mobile Processor, 2014.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995.*, pages 2–13, June 1995.

[8] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.*, pages 104–114, May 1990.

[9] E. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the cray t3e multiprocessor. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 39–39, Nov 1997.

[10] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguadé, J.M. Cela, and M. Valero. Assessing Accelerator-Based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):147–162, 2011.

[11] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 19–31. ACM, 1989.

[12] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 212–221. ACM, 1991.

[13] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[14] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394. ACM, 2013.

[15] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[16] O. Fialka and M. Cadik. Fft and convolution performance in image filtering on gpu. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 609–614, July 2006.

[17] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture, 2009.

[18] Intel Corporation. *Ivy Bridge Archictecture*, 2011.

[19] The Khronos Group Inc. *The OpenCL Specification*, 2013.

[20] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91–101. IPS Press, 1991.

[21] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, Monica S. Lam, and Dash The Ease of use. The Stanford DASH multiprocessor. *IEEE Computer*, 25:63–79, 1992.

[22] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84. ACM, 2009.

[23] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques.*, pages 261–270, Sept 2009.

[24] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, and E. Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Proceedings of 2000 International Conference on Parallel Processing*, ICPP 2000, pages 95–103, 2000.

[25] NVIDIA Corporation. *CUDA C Programming Guide*, 2013.

[26] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *IEEE International Symposium on Parallel Distributed Processing.*, pages 1–12, May 2009.

[27] J.A. Stratton, C. Rodrigues, I-Jui Sung, Li-Wen Chang, N. Anssari, Geng Liu, W.-M.W. Hwu, and N. Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer*, 45(8):26–32, August 2012.

[28] Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, and Wen-mei W. Hwu. Comparison based sorting for systems with multiple GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 1–11. ACM, 2013.

[29] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS 2010, pages 235–246, 2010.

APPENDIX

A. FINITE DIFFERENCE CODE LISTINGS

Listing 1: Distributed implementation (simplified version of the kernel).

```
template <typename T, unsigned Halo>
__global__ void
stencil(T* out, const T* in,
        unsigned cols, unsigned rows, unsigned planes)
{
  int k = threadIdx.x + blockIdx.x * blockDim.x + Halo;
  int j = threadIdx.y + blockIdx.y * blockDim.y + Halo;

  if ((k < Halo + cols) && j < (Halo + rows))
    for (int p = Halo; p < planes + Halo; ++p) {
      T c = IN(k, j, p);
      for (int s = 1; s <= Halo; ++s) {
        T left   = in[IDX_3D(k-s, j, p)];
        T right  = in[IDX_3D(k+s, j, p)];
        T top    = in[IDX_3D(k, j-s, p)];
        T bottom = in[IDX_3D(k, j+s, p)];
        T back   = in[IDX_3D(k, j, p-s)];
        T front  = in[IDX_3D(k, j, p+s)];
        c += 3.f * (left + right) +
             2.f * (top + bottom) +
             1.f * (back + front);
      }
      out[IDX_3D(k, j, p)] = c;
    }
}
```

Listing 2: Distributed implementation (host).

```
struct work_descriptor {
  float *in, *out;
  cudaStream_t stream[NUM_STREAMS];
  cudaEvent_t events_A[NUM_EVENTS], cudaEvent_t events_B[NUM_EVENTS];
  cudaEvent_t *events_prev = events_A;
  cudaEvent_t *events_cur = events_B;
  bool has_left_neigh, has_right_neigh;
  unsigned planes;
};
void do_stencil(work_descriptor wd[NUM_GPUS])
{
  for (int t = 0; t < TIME_STEPS; ++t) {
    for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
      // 1a. Compute right boundary
      if (wd[gpu].has_left_neigh)
        cudaStreamWaitEvent(wd[gpu].stream[EXE_R],
                            wd[gpu-1].events_prev[COMM_R]);
      cudaStreamWaitEvent(wd[gpu].stream[EXE_R], wd[gpu].events_prev[EXE_M]);
      launch_stencil(wd[gpu].in, wd[gpu].out,
                     halo + wd[gpu].planes - halo*2, halo*3, // offset, size
                     wd[gpu].stream[EXE_R]);
      cudaEventRecord(wd[gpu].events_cur[EXE_R], wd[gpu].stream[EXE_R]);
      // 1b. Compute left boundary
      if (wd[gpu].has_right_neigh)
        cudaStreamWaitEvent(wd[gpu].stream[EXE_L],
                            wd[gpu+1].events_prev[COMM_L]);
      cudaStreamWaitEvent(wd[gpu].stream[EXE_L], wd[gpu].events_prev[EXE_M]);
      launch_stencil(wd[gpu].in, wd[gpu].out,
                     0, halo * 3, // offset, size
                     wd[gpu].stream[EXE_L]);
      cudaEventRecord(wd[gpu].events_cur[EXE_L], wd[gpu].stream[EXE_L]);

      // 2. Compute center
      cudaStreamWaitEvent(wd[gpu].stream[EXE_M], wd[gpu].events_prev[EXE_L]);
      cudaStreamWaitEvent(wd[gpu].stream[EXE_M], wd[gpu].events_prev[EXE_R]);
      launch_stencil(wd[gpu].in, wd[gpu].out,
                     halo, wd[gpu].planes, // offset, size
                     wd[gpu].stream[EXE_M]);
      cudaEventRecord(wd[gpu].events_cur[EXE_M], wd[gpu].stream[EXE_M]);

      // 3a. Exchange right boundary
      if (wd[gpu].has_right_neigh) {
        cudaStreamWaitEvent(wd[gpu].stream[COMM_R],
                            wd[gpu].events_cur[EXE_R]);
        copyAsync(wd[gpu+1].out, wd[gpu].out,
                  halo + wd[gpu].planes - halo, halo, // offset, size
                  wd[gpu].stream[COMM_R]);
        cudaEventRecord(wd[gpu].events_cur[COMM_R], wd[gpu].stream[COMM_R]);
      }
      // 3b. Exchange left boundary
      if (wd[gpu].has_left_neigh) {
        cudaStreamWaitEvent(wd[gpu].stream[COMM_L],
                            wd[gpu].events_cur[EXE_L]);
        copyAsync(wd[gpu-1].out, wd[gpu].out,
                  halo, halo, // offset, size
                  wd[gpu].stream[COMM_L]);
        cudaEventRecord(wd[gpu].events_cur[COMM_L], wd[gpu].stream[COMM_L]);
      }
    }
    for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
      swap(wd[gpu].in, wd[gpu].out);
      swap(wd[gpu].events_prev, wd[gpu].events_cur);
    }
  }
}
```

Listing 3: GPU-SM implementation (simplified version of the kernel).

```
template <typename T, unsigned Halo>
__global__ void
stencil(T* out, const T* in, const T* in_left, const T* in_right,
        unsigned cols, unsigned rows, unsigned planes)
{
  int k = threadIdx.x + blockIdx.x * blockDim.x + Halo;
  int j = threadIdx.y + blockIdx.y * blockDim.y + Halo;

  if (k < cols && j < rows)
    for (int p = Halo; p < planes + Halo; ++p) {
      T c = in[IDX_3D(k, j, p)];
      for (int s = 1; s <= Halo; ++s) {
        T left = (in_left && k-s < 0) ?
                 in_left[IDX_3D(cols + (k-s), j, p)] :
                 in[IDX_3D(k-s, j, p)];
        T right = (in_right && k+s >= cols) ?
                  in_right[IDX_3D((k+s) - cols, j, p)] :
                  in[IDX_3D(k+s, j, p)];
        T top    = in[IDX_3D(k, j-s, p)];
        T bottom = in[IDX_3D(k, j+s, p)];
        T back   = in[IDX_3D(k, j, p-s)];
        T front  = in[IDX_3D(k, j, p+s)];
        c += 3.f * (left + right) +
             2.f * (top + bottom) +
             1.f * (back + front);
      }
      out[IDX_3D(k, j, p)] = c;
    }
}
```

Listing 4: GPU-SM implementation (host).

```
struct work_descriptor {
  float *in, *out;
  cudaStream_t stream;
  cudaEvent_t event_prev;
  bool has_left_neigh, has_right_neigh;
  unsigned planes;
};

void do_stencil(work_descriptor wd[NUM_GPUS])
{
  for (int t = 0; t < TIME_STEPS; ++t) {
    for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
      float *in_left = nullptr;
      float *in_right = nullptr;
      if (wd[gpu].has_left_neigh) {
        in_left = wd[gpu-1].in;
        cudaStreamWaitEvent(wd[gpu].stream, wd[gpu-1].event_prev);
      }
      if (wd[gpu].has_right_neigh) {
        in_right = wd[gpu+1].in;
        cudaStreamWaitEvent(wd[gpu].stream, wd[gpu+1].event_prev);
      }
      cudaStreamWaitEvent(wd[gpu].stream, wd[gpu].event_prev);
      launch_stencil(wd[gpu].in, in_left, in_right, wd[gpu].out,
                     wd[gpu].has_left_neigh ?
                        0 : halo, // offset
                     wd[gpu].planes + (wd[gpu].has_right_neigh ?
                                       0 : halo), // size
                     wd[gpu].stream);
    }

    for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
      cudaEventRecord(wd[gpu].event_prev, wd[gpu].stream);
      swap(wd[gpu].in, wd[gpu].out);
    }
  }
}
```