Exploring the Multiple-GPU Design Space

Dana Schaa and David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
{dschaa, kaeli}@ece.neu.edu

Abstract

Graphics Processing Units (GPUs) have been growing in popularity due to their impressive processing capabilities, and with general purpose programming languages such as NVIDIA's CUDA interface, are becoming the platform of choice in the scientific computing community. Previous studies that used GPUs focused on obtaining significant performance gains from execution on a single GPU. These studies employed low-level, architecture-specific tuning in order to achieve sizeable benefits over multicore CPU execution.

In this paper, we consider the benefits of running on multiple (parallel) GPUs to provide further orders of performance speedup. Our methodology allows developers to accurately predict execution time for GPU applications while varying the number and configuration of the GPUs, and the size of the input data set. This is a natural next step in GPU computing because it allows researchers to determine the most appropriate GPU configuration for an application without having to purchase hardware, or write the code for a multiple-GPU implementation. When used to predict performance on six scientific applications, our framework produces accurate performance estimates (11% difference on average and 40% maximum difference in a single case) for a range of short and long running scientific programs.

1. Introduction

The benefits of using Graphics Processing Units (GPUs) for general purpose programming has been recognized for some time, and many general purpose APIs have been created to abstract the graphics hardware from the programmer [2], [10], [15]. General purpose GPU (GPGPU) programming has become the scientific computing platform of choice mainly due to the availability of standard C libraries using NVIDIA's CUDA programming interface running on NVIDIA GTX GPUs. Since its first release, a number of efforts have explored how to reap large performance gains on CUDA-enabled GPUs [3], [7], [16], [18], [20], [21], [23].

One reason for this rapid adoption is that current generation GPUs (theoretically approaching a TFLOP of computational power) have the potential to replace a large number of superscalar CPUs for certain classes of parallel applications. However, obtaining peak GPU performance requires significant programming effort. For example, when running the matrix multiplication function in CUDA's BLAS library on a GeForce 8800 GTX (theoretical 368 peak GFLOPS) we measured a peak execution of around 40 GFLOPS. To begin to obtain better performance, researchers would hand-tune their code to match the characteristics of the underlying GPU hardware. Although tuning may be required in some cases to achieve high performance (such as aligning data for efficient reads in memory-bound applications), these types of optimizations require hardware-specific knowledge and make the code less portable between generations of devices. Similarly, previous work has shown that the optimal data layout varies across different CUDA software releases [20].

Instead of focusing on fine tuning applications for hardware, our approach is to utilize multiple GPUs to exploit even larger degrees of parallelism. To show that execution on multiple GPUs is beneficial, we introduce models for the various components of GPU execution and provide a methodology for predicting execution of GPU applications. Our methodology is designed to accurately predict the execution time of a given CUDA application (based on a single-GPU implementation), while varying the number of GPUs, their configuration, and the data set size of the application. Using this technique, we can generate accurate performance estimates that allow users to determine the system and GPU configuration that best fits their cost-performance needs.

Execution on parallel GPUs is promising because applications that are best suited to run on GPUs inherently have large amounts of parallelism. Utilizing multiple GPUs avoids dealing with many of the limitations of GPUs (e.g., on-chip memory resources) by exploiting the combined resources of multiple boards. Of course, inter-GPU communication becomes a new problem that we need to address, and involves considering the efficiency of the current communication fabric provided on GPUs. Our resulting framework is both effective and accurate in capturing the dynamics present as we move from a single GPU to multiple GPUs. With our work, GPU developers can determine potential speedups gained from execution of their applications on any number of GPUs without having to purchase expensive hardware or even write code to simulate a parallel implementation.

The remainder of this paper is organized as follows.

Section 2 presents previous GPGPU performance studies, including some works that utilize multiple GPUs. Section 3 is a brief overview of the CUDA programming model. Since the factors affecting multiple-GPU execution have not been discussed in previous work, in Section 4 we provide background on the multiple-GPU design space, and also describe our methodology. Section 5 discusses the applications used to illustrate the power of our methodology. Results are presented in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

Using GPUs for general purpose scientific computing has allowed a range of challenging problems to be solved faster and has enabled researchers to study larger (e.g., finer-grained) data sets [3], [7], [16], [18], [23]. In [21], Ryoo et. al provide a nice overview of the GeForce 8800 GTX architecture, and also present strategies for optimizing performance. We build upon some of the ideas presented in these prior papers by exploiting additional degrees of available parallelism with the mapping of applications to multiple GPUs. Our work shows that there can be great benefits when pursuing this path.

2.1. Parallel GPUs

Even though most prior work has focused on algorithm mapping techniques specific to performance on a single GPU, there are a number of efforts studying how to exploit larger numbers of GPUs to accelerate specific problems.

The Visualization Lab at Stony Brook University has a 66-node cluster that contains GeForce FX5800 Ultra and Quadro FX4500 graphics cards that are used for both visualization and computation. Parallel algorithms implemented on the cluster include medical reconstruction, particle flow, and dispersion algorithms [6], [17]. This prior work targets effective usage of a networked groups of distributed systems, each containing a single GPU.

Moerschell and Owens describe the implementation of a distributed shared memory system to simplify execution on multiple distributed GPUs [11]. In their work, they formulate a memory consistency model to handle inter-GPU memory requests. By their own admission, the shortcoming of their approach is that memory requests have a large impact on performance, and any abstraction where the programmer does not know where data is stored (i.e., on or off GPU) is impractical for current GPGPU implementations. Still, as GPUs begin to incorporate Scalable Link Interface (SLI) technology for boards connected to the same system, this technique may prove promising.

In a related paper, Fan et. al explore how to utilize distributed GPU memories using object oriented libraries [5]. For one particular application, they were able to decrease the code size for a Lattice-Boltzman model from 2800 lines to 100 lines, while maintaining identical performance.

Caravela [24] is a stream-based computing model that incorporates GPUs into GRID computing and uses them to replace CPUs as computation devices. While the GRID is not an ideal environment for coordinated GPGPU execution (due to the communication overheads we will discuss in Section 4), this model may have a niche for long-executing applications with large memory requirements, as well as for researchers who wish to run many batch GPU-based jobs.

Given the growing interest in exploiting multiple GPUs, and previous work that has demonstrated that performance gains are possible for specific applications, an in-depth analysis of the factors affecting multiple-GPU execution is lacking. The main contribution of our work is the creation of models representing the significant aspects of parallel-GPU execution (especially communication). These models can be used to determine the best system configuration to deliver the required amount of performance for any application, while taking into account factors such as hardware specifications and input data size. Our work should accelerate the move to utilizing multiple GPUs in GPGPU computing.

3. Parallel Computing with CUDA

Next, we provide a brief overview of the CUDA programming model as relevant to this work. We also discuss issues with CUDA related to distributed and shared-system GPU computing. Those seeking more details on CUDA programming should refer to the CUDA tutorials provided by Luebke, et. al [9]. To facilitate a discussion of the issues involved in utilizing multiple GPUs, we begin by formalizing some of the terminology in this work:

- **Distributed GPUs** - A networked group of distributed systems each containing a single GPU.
- **Shared-system GPUs** - A single system containing multiple GPUs that communicate through a shared CPU RAM (such as the NVIDIA Tesla S870 server [13]).
- **GPU-Parallel Execution** - Execution that takes place across multiple GPUs in parallel (as opposed to parallel execution on a single GPU). This term is inclusive of both distributed and shared-system GPU execution.

3.1. The CUDA Programming Model

CUDA terminology refers to a GPU as the *device*, and a CPU as the *host*. These terms are used in the same manner for the remainder of this paper. Next, we summarize CUDA's threading and memory models.

3.1.1. Highly Threaded Environment. CUDA supports a large number of active threads and uses single-cycle context switches to hide datapath and memory-access latencies. When running on NVIDIA's G80 Series GPUs, threads

are managed across 16 multiprocessors, each consisting of 8 single-instruction-multiple-data (SIMD) cores. CUDA's method of managing execution is to divide groups of threads into *blocks*, where a single block is active on a multiprocessor at a time. All of the blocks combine to make up a *grid*. Threads can determine their location within a block and their block's location within the grid from intrinsic data elements initialized by CUDA. Threads within a block can synchronize with each other using a barrier function provided by CUDA, but it is not possible for threads in different blocks to directly communicate or synchronize. Applications that map well to this model have the potential for success with utilizing multiple GPUs because of their high degree of data-level parallelism.

The CUDA 1.1 architecture does support a number of atomic operations, but these operations are only available on a subset of GPUs[1], and frequent use of atomic operations limits the parallelism afforded by a GPU. These atomic operations are the only mechanism for synchronization between threads in different blocks.

3.1.2. Memory Model. Main memory on the G80 Series GPUs is a large RAM (384MB-768MB) that is accessible from every multiprocessor. This memory is referred to as *device memory* or *global memory*. Additionally, each multiprocessor contains 16KB of cache that is shared between all threads in a block. This cache is referred to as *shared memory* or *shared cache*. Unlike most CPU memory models, there is no mechanism for automated caching between GPU RAM and shared memory.

GPUs can not directly access CPU RAM during execution. Instead, data is explicitly transferred between GPU and CPU RAM prior to and following GPU execution. Since manual memory management is required for GPU execution (there is no paging mechanism), and because GPUs cannot exchange data with the CPU during execution, programmers need to modify and potentially segment their applications such that all relevant data is located in the GPU when needed. Data sets that are too large to fit in a single GPU require multiple transfers between CPU and GPU memories, and this can introduce stalls in execution.

As with traditional parallel computing, using multiple GPUs provides additional resources, potentially requiring fewer GPU calls and allowing for the simplification of algorithms. However, compounding data transfers and execution breaks with traditional parallel computing communication costs may cancel out any benefits reaped from parallel execution. These issues (and others related to parallel GPU communication) are discussed in detail in Section 4.

---

1. Atomic operations are not available on the GeForce 8800 GTX and Ultra GPUs used in this work.

3.2. GPU-Parallel Execution

3.2.1. Shared-System GPUs. In the CUDA environment, GPUs cannot yet interact with each other directly, but it is likely that SLI will soon be supported for inter-GPU communication on devices connected to the same system. Until then, shared-system GPU execution requires that different CPU threads invoke execution on each GPU. The rules for interaction between CPU threads and CUDA-supported GPUs are as follows:

1) A CPU thread can only execute programs on a single GPU (working with two GPUs requires two CPU threads, etc.).
2) Any CUDA resources created by one CPU thread cannot be accessed by another thread.
3) Multiple CPU threads can invoke execution on a single GPU, but may not be run simultaneously.

These rules help to ensure isolation between different GPU applications.

3.2.2. Distributed GPUs. Distributed execution does not face the same program restructuring issues as found in shared-system GPU execution. In a distributed application, if each system contains only a single GPU, all of the threading rules described in the previous section will not apply since each CPU thread interacts with the GPU in the same manner as a single-GPU application.

Just as in traditional parallel computing, distributed GPU processing scales better than processing on shared-systems because it will not overwhelm shared system resources. However, unlike the forthcoming SLI support for multiple-GPU systems, distributed execution will continue to require programmers to utilize a communication middleware such as MPI.

3.2.3. GPU-Parallel Algorithms. An obvious disadvantage of a GPU being located across the PCI-e bus is that it does not have direct access to the CPU memory bus, nor does it have the ability to swap data to disk. Because of these limitations, when an application's data set is too large to fit entirely into GPU RAM, the algorithm needs to be modified so that the data can be exchanged with CPU RAM. The modifications required to split an algorithm's data set essentially creates a GPU-parallel version of the algorithm already, and so the transition to multiple GPUs is natural and only involves coding the appropriate shared memory or network-based communication mechanism.

4. Methodology for Predicting Multiple-GPU Performance

Traditional parallel computing obtains speedup by dividing up program execution across multiple processors while

attempting to limit the overhead associated with communication. Distributed systems are limited by network throughput, but have the advantage that they otherwise scale easily. Shared memory systems have a much lower communication penalty, but do not scale as well because of the finite system resources that must be shared (RAM, buses, etc.).

The traditional parallel computing model can be adapted to GPU computing as expressed in Equation 1.

$$t_{total} = t_{cpu} + t_{cpu\_comm} + t_{gpu} + t_{gpu\_comm} \qquad (1)$$

$$t_{cpu\_comm} = \begin{cases} t_{memcpy} & \text{for shared systems} \\ t_{network} & \text{for distributed systems} \end{cases} \qquad (2)$$

In Equation 1, t_{cpu} and t_{cpu_comm} represent the factors of traditional parallel computer: t_{cpu} is the amount of time spent executing on a CPU, and t_{cpu_comm} is the inter-CPU communication requirement. Both t_{cpu} and t_{cpu_comm} should only be considered for time that does not overlap with GPU execution or communication. Since one of our requirements is that deterministic applications must be used (Section 4.1), it should be trivial to factor out the overlap. Equation 2 acknowledges that the time for CPU communication varies based on the GPU configuration. Since GPUs can theoretically be managed as CPU co-processors, we can employ a traditional parallel computing communication model. In Equation 2, t_{memcpy} is the time spent transferring data within RAM for shared memory systems, and $t_{network}$ is the time spent transferring data across a network for distributed systems.

In addition to the typical overhead costs associated with parallel computing, we now add t_{gpu} and t_{gpu_comm}, where t_{gpu} represents the execution time on the GPU and is discussed in Section 4.1, and t_{gpu_comm} represents additional communication overhead and is discussed in Sections 4.2 and 4.3.

Using these factors, we provide a methodology that can be used to extrapolate actual execution time across multiple GPUs and data sets. The ultimate goal is to allow developers to determine the benefits of multiple-GPU execution without needing to purchase hardware and with using only the single-GPU application.

4.1. Modeling GPU Execution

Our methodology requires that a CUDA program exists that executes on a single GPU. This application is used as the basis for extrapolating the amount of time that multiple GPUs will spend on computation. In order to extrapolate GPU execution accurately, we introduce the requirement that the application running on the GPU must be deterministic. However, this requirement does not limit us severely since most applications that will benefit from GPUs are already highly parallel and possess a stable execution profile. Still, applications such as those that model particle interaction may require reworking if exchanging information with neighbors is non-deterministic (thus requiring inter-GPU communication). Lastly, since the execution time will change based on the GPU hardware, we assume in this paper that the multiple-GPU application will run on GPUs all of the same model (we allow for heterogeneous GPU modeling in our future work).

Using our approach, we first need to determine how GPU execution scales on M GPUs. The two metrics that we use to predict application scalability as a function of the number of GPUs are *per-element* averages and *per-subset* averages. Elements refer to the smallest unit of computation involved with the problem being considered, as measured on an element-by-element basis. Subsets refer to working with multiple elements, and are specific to the granularity and dimensions of the data sets involved in the application being parallelized.

To calculate the per-element average, we determine the time it takes to compute a single element of a problem by dividing the total execution time of the reference problem (t_{ref_gpu}) by the number of elements ($N_{elements}$) that are calculated. This is the average execution time of a single element and is shown in Equation 3. The total execution time across M GPUs can then be represented by Equation 4. As long as a large number of elements are present, this has proven to be a highly accurate method. However, the programmer should still maintain certain basic CUDA performance practices, such as ensuring that warps remain as filled as possible when dividing the application between processors to avoid performance degradation. Also, when finding the reference execution time, the PCI-Express transfer time should be factored out.

$$t_{element} = \frac{t_{ref\_gpu}}{N_{elements}} \qquad (3)$$

$$t_{gpu} = t_{element} * \left\lceil \frac{N_{elements}}{M_{gpus}} \right\rceil \qquad (4)$$

An alternative to using per element averages is to work at a coarser granularity. Applications sometimes lend themselves to splitting data into larger *subsets* (e.g., 2D slices of a 3D matrix). Using the reference GPU implementation, the execution time of a single subset (t_{subset}) is known, and the subsets are divided between the multiple GPUs. We assume that t_{subset} can be obtained empirically, because the execution is likely long enough to obtain an accurate reference time (as opposed to per-element execution times that might suffer from imprecision due to their length). Equation 5 is then the execution time of N subsets across M GPUs.

$$t_{gpu} = t_{subset} * \left\lceil \frac{N_{subsets}}{M_{gpus}} \right\rceil \qquad (5)$$

In either case, if the number of execution units cannot be divided evenly by the number of processing units, the GPU execution time is based on the longest running execution.

4.2. Modeling PCI-Express

In this section, we discuss the impact of shared-system GPUs on the PCI-e bus.

4.2.1. Pinned Memory. The CUDA driver supports allocation of memory that is *pinned* to RAM (i.e., non-pageable). Pinned memory increases the device bandwidth and helps reduce data transfer overhead, because transfers can occur without having to first move the data to known locations within RAM. However, because interaction with the CUDA driver is required, each request for pinned allocation (t_{pinned_alloc} in Equation 7) is much more expensive than the traditional method of requesting pageable memory from the kernel. Measured pinned requests take 0.1s on average (CUDA version 1.0).

In multiple-GPU systems, or in general when the data set size approaches the capacity of RAM, the programmer must be careful that the amount of pinned memory allocated does not exceed what is available, or else system performance will degrade significantly. The use of pinned memory also makes code less portable, because systems with less RAM will suffer when applications allocate large amounts of pinned data. As expected, our tests show that creating pinned buffers to serve as *staging-areas* for GPU data transfers is not a good choice, because copying data from pageable to pinned RAM is the exact operation that the CUDA driver performs before normal, pageable transfers to the GPU occur. As such, allocating pinned memory for use with the GPU should only be done when the entire data set can fit in RAM.

4.2.2. Data Transfers. In order to increase our design space, and because of the impact of multiple GPUs, our shared-systems are equipped with GeForce 8800 GTX Ultras. The Ultra GPUs are clocked faster than the standard 8800 GTX GPUs, which gives them the ability to transfer and receive data at a faster rate and can potentially help alleviate some of the PCI-e bottlenecks.

The transfer rates from both pinned and pageable memory in CPU RAM to a GeForce 8800 GTX and a GeForce 8800 GTX Ultra across a 16x PCI-e bus are shown in Table 1. Pinned memory is faster because the CUDA driver knows the data's location in CPU RAM and does not have to locate it, nor potentially swap it in from disk, nor copy it to a non-pageable buffer before transferring it to the GPU.

As expected, as more GPUs are connected to the same shared PCI-e bus, the increased pressure impacts transfer latencies. Table 1 shows measured transfer rates for one and two GPUs, and extrapolates the per-GPU throughput to four Ultra GPUs in a shared-bus scenario.

Device	GPUs	Memory Type	Throughput
8800 GTX	1	pageable	1350MB/s
8800 GTX	1	pinned	1390MB/s
8800 Ultra	1	pageable	1638MB/s
8800 Ultra	2	pageable	695MB/s
8800 Ultra	4	pageable	*347MB/s
8800 Ultra	1	pinned	3182MB/s
8800 Ultra	2	pinned	1389MB/s
8800 Ultra	4	pinned	*695MB/s

Table 1. Transfer throughput between CPU and GPU. *The throughput of 4 shared-system GPUs is an upperbound.

This communication overhead must be carefully considered, especially for algorithms with large data sets that execute quickly on the GPU. Although transfer rates vary based on direction, they are similar enough that we use the CPU to GPU rate as a reasonable estimate for transfers in both directions.

Figure 1 shows how GPUs that are connected to the same system will share the PCI-e bus. The bus switch allows either of the two GPUs to utilize all 16 PCI-e channels, or the switch can divide the channels between the two GPUs. Regardless of the algorithm used for switching, one or both GPUs will incur delays before they receive all of their data. As such, delays will occur before execution begins on the GPU (CUDA requires that all data is received before execution begins).
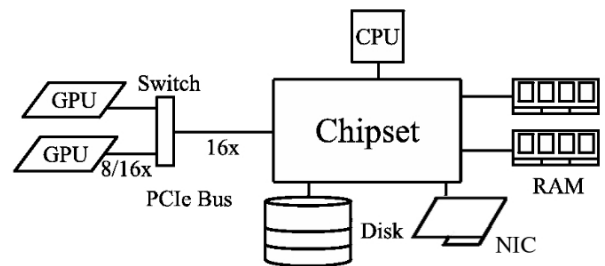

Figure 1. PCI-Express configuration for a two-GPU system.

4.3. Modeling RAM and Disk

4.3.1. Determining Disk Latency. The data that is transferred to the GPU must first be present in system RAM. Applications with working sets larger than RAM will therefore incur extra delays if data has been paged and must be retrieved from disk prior to transfer. Equation 6 shows that the time to transfer data from disk to memory varies based

$$t_{disk'} = \begin{cases} 0 & x < B_{RAM} & (a) \\ \frac{x - B_{RAM}}{T_{disk}} & B_{RAM} < x < B_{RAM} + B_{transfer} & (b) \\ \frac{B_{transfer}}{T_{disk}} & B_{RAM} + B_{transfer} < x & (c) \end{cases} \tag{6}$$

Model for LRU paging, where B is bytes of data and x is the total amount of data allocated.

$$t_{gpu\_comm} = \begin{cases} t_{pinned\_alloc} + t_{pci-e} & \text{pinned memory} \\ t_{disk} + t_{pci-e} & \text{pageable memory} \end{cases} \tag{7}$$

on the relationship between the size of RAM (B_{RAM}), the size of the input data (x), and the amount of data being transferred to the GPU ($B_{transfer}$). Equation 6(a) shows that when data fits in RAM, no paging is necessary. In Equation 6(b), a fraction of the data resides on disk and must be paged in, and in Equation 6(c) all of the data from disk must be transferred in. These equations are used to represent one-way transfers between disk and RAM ($t_{disk'}$), which may occur multiple times during a single GPU execution.

We assume that our GPU applications process streaming data, which implies that the data being accessed is always the least recently used (LRU). Even if this is not the case, our model still provides a good upper bound on transfers. The following list details the model used in this work to accurately predict disk access times:

1) Whenever a data set is larger than RAM, all transfers to the GPU require input data that is not present in RAM. This requires both paging of old data out to disk, and paging desired data in from disk—which equates to twice $t_{disk'}$ as accounted for in Equation 6.
2) Copying output data from the GPU to the CPU does not require paging any data to disk. This is because the input data in CPU main memory is unmodified and the OS can invalidate it in RAM without consequence (a copy will still exist in swap on the disk). Therefore there is no additional disk access time required when transferring data back to RAM from the GPU. This holds true as long as the size of the output data is less than or equal to the size of the input data.
3) If a GPU algorithm runs multiple iterations, a given call may require a combination of input data and output data from previous iterations. In this case, both input and output data would have to be paged in from disk. Prior input data living in RAM could be discarded as in (2), but prior output data will need to be paged to disk.

These three assumptions, while straightforward, generally prove to accurately estimate the impact of disk paging in conjunction with the Linux memory manager, even though they ignore any bookkeeping overhead. The combination of $t_{disk'}$s make up t_{disk} which is used in Equation 7, and are algorithm-specific.
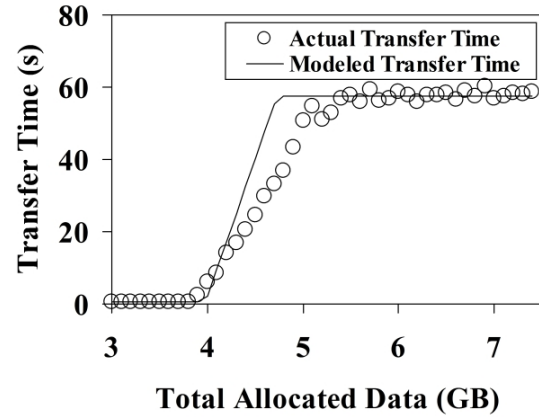

Figure 2. Time to transfer 720MB of paged data to a GeForce 8800 GTX GPU, based on the total data allocation on a system with 4GB of RAM.

4.3.2. Empirical Disk Throughput. In order to predict execution time, disk throughput is measured for a given system. To determine throughput (and to verify Equation 6), we ran a set of tests that vary the amount of data allocated, while transferring a fixed amount of data from disk to GPU.

Figure 2 presents the time to transfer 720MB of data from disk to a single GeForce 8800 GTX GPU. It shows that once the RAM limit is reached ($\sim$3.8GB on our 4GB system, with about 200MB reserved for the kernel footprint), the transfer time increases until all data resides on disk. In this scenario, space must be cleared in RAM by paging out the same amount of data that needs to be paged in, so the throughput in the figure is really only half of the actual speed. Based on the results, 26.2MB/s is assumed to be the data transfer rate for our disk subsystem, and is used in calculations to estimate execution time in this work. Since we assume that memory accesses always need LRU data, we can model a system with N GPUs by dividing the disk bandwidth by N to obtain the disk bandwidth for each GPU transfer.

To summarize, in this section we discussed how to estimate GPU computation time based on a reference implementation from a single GPU, and also introduced techniques for

System Specific Inputs	Algorithm Specific Inputs	Variables	Output
Disk Throughput Network Bandwidth PCI-e (GPU) Bandwidth RAM Size	Communication Requirements Reference Implementation	Number of GPUs Data Set Sizes GPU Configuration	Execution Times

Table 2. Design space and requirements for predicting execution

determining PCI-e and disk throughput. These factors are combined in Equation 7—in which t_{pinned_alloc} is the time required for memory allocation by the CUDA driver, t_{pci-e} is the time to transfer data across the PCI-e bus, and t_{disk} is the time to page in data from disk. These costs, represented as t_{gpu_comm} combine to make up the GPU communication requirements as presented in Equation 1.

It should be noted that we do not need to represent RAM in the communication model, because it is already taken into account in the empirical throughputs of both PCI-e and disk transfers.

The models that we have presented in this section provide all the information necessary to predict execution. Table 2 summarizes the inputs described in this section, as well as the factors that can be varied in order to obtain a complete picture of performance across the application design space.

5. Applications and Environment

Next, we discuss the six scientific applications used to evaluate our framework. For each application we predict the execution time while varying the number and configuration of GPUs. We also predict the execution time while varying the input data set sizes—all of which is done without requiring a GPU-parallel implementation of the algorithm. We then compare the results to actual execution of GPU-parallel implementations on multiple GPUs to verify the accuracy of our framework.

5.1. Applications

Convolution: A 7x7 convolution kernel is applied to an image of variable size. All images used were specifically designed to be too large to fit entirely in a single 8800 GTX GPU memory, and are therefore divided into segments with overlapping pixels at the boundaries. Each pixel in the image is independent, and no communication is required between threads.

Least-Squares Pseudo-Inverse: The least-squares pseudo-inverse application is based on a medical visualization algorithm where point-paths are compared to a reference path. Each point-to-reference comparison is calculated by a thread, and no communication between

threads is required. Each thread computes a series of small matrix multiplications and a 3x3 matrix inversion.

Image Reconstruction: This application is a medical imaging algorithm that uses tomography to reconstruct a three-dimensional volume from multiple two-dimensional X-ray views [18]. The X-ray views and volume slices are independent and are divided between the GPUs. The algorithm requires a fixed number of iterations, between which large amounts of data must be swapped between GPU and CPU. When using multiple GPUs, each must receive updated values from all other GPUs between iterations.

Ray Tracing: This application is a modified version of the Ray Tracing program created by Rollins [19]. In his single GPU implementation, each pixel value is computed by a independent thread that contributes to a global state that is written to a frame buffer. In the GPU-parallel version, each pixel is still independent, but after each iteration the state of the objects on the screen must be synchronized before a frame buffer update can be performed.

2D FFT: For the two-dimensional FFT, we divided the input data into blocks whose size is based on the number of available GPUs. The algorithm uses CUDA's CUFFT libraries for the two FFTs, performs a local transpose of each block, and requires intermediate communication. Since the transpose is done on the CPU, the execution time for each transpose is obtained empirically for each test.

Matrix Multiplication: Our matrix multiplication algorithm divides the input and output matrices into blocks, where each thread is responsible for a single value in the output matrix. For the distributed implementation, Fox's algorithm is used for the communication scheme [4]. For the matrix multiplication and 2D FFT, the choice of using a distributed communication scheme was arbitrary, as we are trying to show that we can predict performance for any algorithm, and may not be the fastest choice.

5.2. Hardware Setup

For the GPU-parallel implementations discussed in Section 6, two different configurations are used. For experiments where a cluster of nodes is used as a distributed system, each system had a 1.86GHz Intel Core2 processor with 4GB of RAM, along with a GeForce 8800 GTX GPU with 768MB of on-board memory connected via 16x PCI-e bus. The

```
# ------------ Distributed Ray Tracing ------------
0 P = 16                          # Number of GPUs
1 XDIM = 1024                     # Original image width (pixels)
2 YDIM = 768                      # Original image height (pixels)
3 SCALE = 4                       # Scale the image to 4X the original size
4 GLOBAL_STATE_SIZE = 13548  # In Bytes
5
6 for i = 1 to SCALE             # Loop over scale of image
7  for j = 1 to P                # Loop over number of GPUs
8    ROWS = (i * XDIM / j)       # Distribute the rows
9    COLS = i * YDIM
10   DATA_PER_GPU = ROWS * COLS * PIXEL_SIZE
11
12   GPU_TIME = (i² * 0.314 / j)
13   CPU_TIME = 0
14   NETWORK_TIME = step(j - 1) * (DATA_PER_GPU * (j - 1) / NET_BAND + ...
15       GLOBAL_STATE_SIZE * (j - 1) / NET_BAND)
16   DISK_TIME = 0
17   PCIE_TIME = (DATA_PER_GPU / PCIE_BAND) + (GLOBAL_STATE_SIZE / PCIE_BAND)
18
19   FPS[j,i] = 1 / (GPU_TIME + CPU_TIME + NETWORK_TIME + DISK_TIME + PCIE_TIME)
```

Figure 3. Predicting performance for distributed ray tracing. Results are shown in Figure 5(a).

system used in multithreaded experiments has a 2.4GHz Intel Core2 processor with 4GB of RAM. This system is equipped with a GeForce 8800 GTX Ultra with 768GB of on-board RAM. All systems are running Fedora 7 and use a separate graphics card to run the system display.

6. Performance Prediction and Verification

Next, we compare estimated and actual execution times for multiple GPUs for our six applications.

6.1. Predicting Execution Time

To predict execution time, we begin with a single-GPU implementation of an algorithm, and create a high-level specification for a GPU-parallel version (which also includes the communication scheme). Using the equations and methodology described in Section 4, we can compute the GPU and CPU execution costs, the PCI-e transfer cost, the disk paging cost, and the network communication cost. Using these calculations we are able to predict the execution time for variable data set sizes and numbers of GPUs.

While our methodology identifies methods for accurately accounting for individual hardware element latencies, it is up to the developer to accurately account for algorithm specific factors. For example, if data is broadcast using MPI, the communication is on the order of $\lceil log_2 N \rceil$. Factors such as this are algorithm specific and must be considered.

Figure 3 contains pseudo-code representing the equations we used to plot the predicted execution of distributed ray tracing for up to 16 GPUs (line 0). An initial frame of 1024x768 pixels is used, and scales up to 4 times the original size in each direction (lines 1-3). The single-GPU execution time is determined from an actual reference implementation (0.314 seconds of computation per frame), and since each pixel is independent, line 12 both accounts for scaling the frame size and computing the seconds of execution per GPU. The rows of pixels are divided evenly between the P processors (line 8), which means that each GPU is only responsible for $\frac{1}{P}$ of the pixels in each frame. The size of the global state (line 4) is based on the number of objects we used in the ray tracing program, and is just over 13KB in our experiments. Since the data size is small enough to fit entirely in RAM, no disk paging is required (line 16). Similarly, with no significant execution on the CPU, the CPU execution time can be disregarded (line 13). Line 14 is the time required to collect the output of each GPU (which is displayed on the screen) and the global state that must be transferred back across the network is accounted for in line 15. In addition to the network transfer, each GPU must also transfer its part of the frame across the PCI-e bus to CPU RAM, and then receive the updated global state (line 17). DISK_BAND, PCIE_BAND, and NET_BAND are all empirical measurements that are constants on our test systems. The result is the predicted number of frames per second (line 19) that this algorithm should be able to compute using 1 to 16 GPUs and for 4 different data sizes. Later in this section we discuss and plot the results for distributed ray tracing (Figure 5(a)).

Figures 4 through 7 were all generated using the same

(a) Convolution with Distributed GPUs.

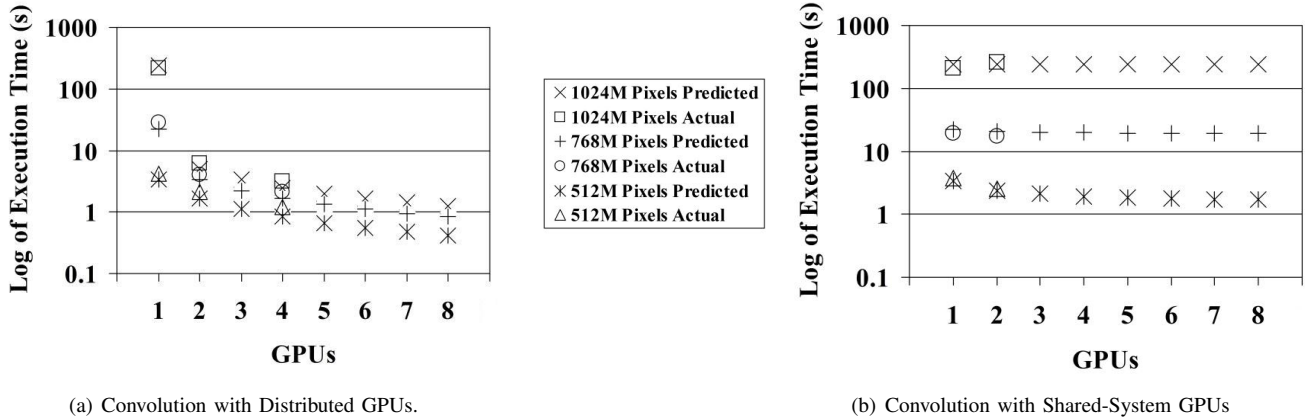(b) Convolution with Shared-System GPUs

Figure 4. Convolution results plotted on a logarithmic scale.

technique described here. These figures allow us to visualize the results from our predictions, and easily choose a system configuration that best matches the computational needs of a given application.

6.2. Prediction Results

To demonstrate the utility of our framework, we tested various configurations and data sets for each application using four distributed and two shared-system GPUs. Using our methodology, the average difference between predicted and actual execution is 11%. Our largest prediction difference was 40% and occurred when the data set was just slightly larger than RAM. For this case, our models assumed that data must always be brought in from disk, when in reality the Linux memory manager implements a more efficient policy. However, the piece-wise function presented in Equation 6 could easily be modified to accommodate this OS feature.

We feel that our framework provides guidance in the design optimization space, even considering that some execution times are incredibly short (errors may become more significant), while other execution times (involving disk paging or large computations) are very long (errors may have time to manifest themselves).

Next we present the results for all applications. While the main purpose of this work is to present our methodology and verify our models, we also highlight some trends based on communication and execution characteristics as well.

6.2.1. Zero-Communication Applications. Both the least-squares and convolution applications require no communication to take place during execution. Each pixel or point that is computed on the GPU is assigned to a thread and undergoes a series of operations that do not depend on any other threads. The operations on a single thread are usually fast, which means that large input matrices or images are required to amortize the extra cost associated with GPU

data transfers. Data that is too large to fit in a single GPU memory causes multiple transfers, incurring additional PCI-e (and perhaps disk) overhead.

Using multiple distributed GPUs when processing large data sets allows parallelization of memory transfers to and from the GPU, and also requires fewer calls per GPU. This means that each GPU spends less time transferring data. Shared-system GPUs have a common PCI-e bus, so the benefits of parallelization are not as large for these types of algorithms because transfer overhead does not improve. However, computation is still parallelized and therefore some speedup is seen as well. Figure 4 shows the predicted and empirical results for the convolution application.

Figure 4(a) shows that when running on multiple systems, distributed GPUs prevent paging that is caused when a single system runs out of RAM and must swap data in from disk before transferring it to the GPU. Alternatively, Figure 4(b) shows that since multiple shared-system GPUs have a common RAM, adding more GPUs does not prevent paging to disk.

Applications that have completely independent elements and do not require communication or synchronization can benefit greatly from GPU execution. However, Figure 4 shows that it is very important for application data sets to fit entirely in RAM if we want to effectively exploit GPU resources (note the log scale). We want to ensure that when these data sets grow, performance will scale. Distributed GPUs are likely the best match for these types of algorithms if large data sets are involved.

6.2.2. Data Sync Each Iteration. The threads in the ray tracing and image reconstruction applications work on independent elements of a matrix across many consecutive iterations of the algorithm. However, dissimilar to zero-communication applications described previously, these threads must update the global state, and this update must be synchronized so that it is completed before the next iteration

(a) Ray Tracing with Distributed GPUs
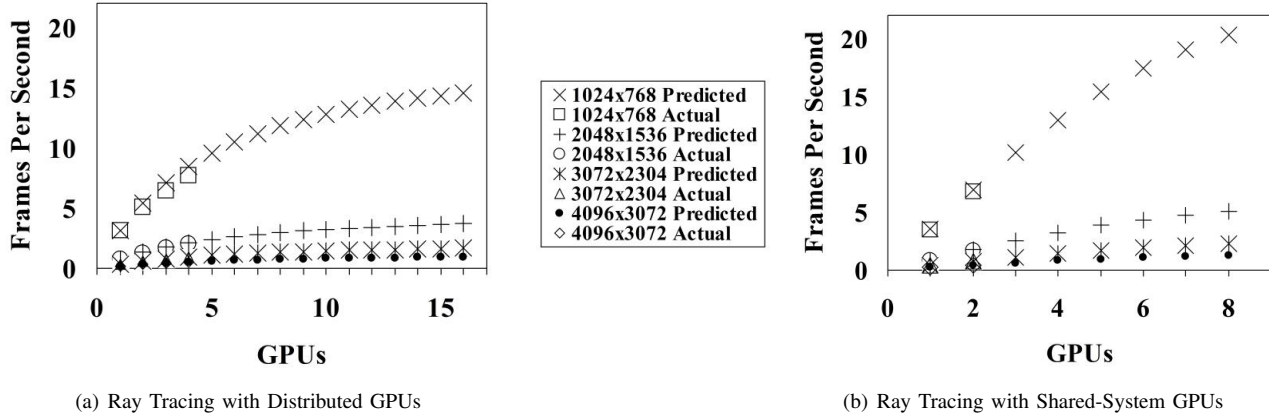

(b) Ray Tracing with Shared-System GPUs

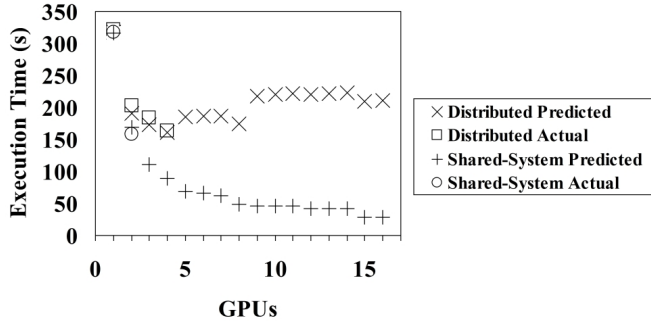Figure 5. Results for Ray Tracing across four data sets.

Figure 6. Results for Image Reconstruction for a single data size.

begins.

For applications that possess this pattern, shared-system implementations have the advantage that as soon as data is transferred across the PCI-e bus back to main memory, the data is available to all other GPUs. In applications such as ray tracing (Figure 5) and image reconstruction (Figure 6), where a large number of iterations need to be computed, the shared-system approach shows more potential for performance gains than distributed GPUs.

For the shared-system execution of ray tracing, the expected PCI-e bottleneck does not begin to occur until we get close to 8 GPUs even though the execution time is relatively short, because the data simply needs to be placed in CPU RAM to synchronize. On the other hand, the distributed execution hits a network bottleneck much sooner because each GPU must broadcast its updates to every other GPU. Similar conclusions can be drawn from the results of the image reconstruction application.

Note that the data sets of both of these applications fit entirely in RAM, so disk paging is not a concern. However, if these applications were scaled to the point where their data no longer fits in RAM, we would see similar results as shown in Figure 4, and the distributed GPUs would likely outperform the shared-system GPUs since network transfers are usually much faster than disk accesses. Also, observe that the step-like behavior illustrated by the distributed image reconstruction algorithm in Figure 6 is due to the $\lceil log_2 N \rceil$ cost of MPI's broadcasting. This means that we see an increase in the cost of communication at processor numbers that are one greater than a power of two. Execution time tends to decrease slightly as each processor is added before the next power of two, because of the availability of additional computational resources without any additional communication costs.

It should also be noted that these tests were performed on a 1Gb network. When we modeled execution on a 10Gb network, the results showed the distributed performance was on par with shared-system performance for image reconstruction, and that distributed ray tracing outperforms shared-system ray tracing due to the PCI-e contention as more GPUs are added. The variation of performance due to system parameters is yet another justification for using our methodology.

6.2.3. Multi-read Data. Applications such as the distributed 2D FFT and matrix multiplication involve transferring large amounts of data between processors. The reason for this communication is because each data element is read multiple times by different threads during execution. In distributed environments, this almost always implies that the data will be transferred between processors. Since these algorithms rely heavily on network speed, and because a single GPU can compute as much as many CPUs, a fast network is critical for these algorithms to be run effectively on multiple GPUs.

Since the block size of the matrix multiplication algorithm varies with the number of GPUs, we used our single-GPU implementation to measure the per-GPU execution time for
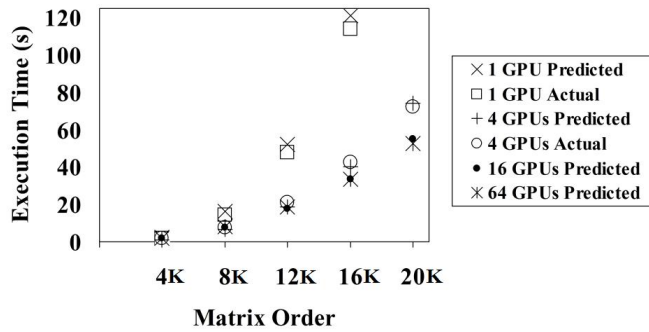
Figure 7. Distributed Matrix Multiplication using Fox's Algorithm.

each data-size-per-GPU combination. The communication time was predicted by modeling Fox's algorithm.

In the distributed FFT algorithm, t_{CPU} (from Equation 1) plays a large role due to the cost of computing matrix transposes, which we chose to leave on the CPU. Since transposing algorithms is incredibly sensitive to hardware-specific optimizations, and because it is easy to produce tests for different data sizes on the CPU, t_{CPU} was always obtained from actual measured data.

7. Conclusion

GPUs are already impacting scientific computing by utilizing massively parallel processing and providing the potential for orders of magnitude higher computation per second than multi-core CPUs. However, current research has focused on extracting the best performance from a single GPU platform by focusing on application-specific coding, which negatively affects the scalability and portability of algorithms. Specific applications that incorporate multiple GPUs have been presented in prior work, but an in-depth study that considers how best to construct multiple-GPU systems has not been pursued.

In this work we show that our modeling framework can produce accurate estimates when moving single-GPU applications to a multiple-GPU platform. Our approach develops a set of performance equations that capture many of the latencies and dependencies introduced in a multiple-GPU environment. Our methodology is generic enough to apply to any system or GPU model, as long as the application behaves in a deterministic manner. In the six applications we tested, we were able to estimate the execution time across multiple GPU applications with an average difference of 11% when compared to actual execution times. Our validation study included applications that have a wide range of execution durations. In future work we intend to refine our model and also consider the

impact of a wider range of workloads in our framework.

References

[1] AMD. AMD FireStream 9170: Industry's First GPU with Double-Precision Floating Point. http://ati.amd.com/products/streamprocessor/specs.html, 12 2007.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004.

[3] B. Deschizeaux and J. Blanc. *Imaging Earth's Subsurface Using CUDA, GPU Gems 3*, chapter 38. Addison-Wesley, 2007.

[4] M. Fagan. Fox Matrix Multiply Algorithm. http://www.caam.rice.edu/ caam520/Topics/ ParallelAlgorithms/LinearAlgebra/fox.html.

[5] Z. Fan, F. Qiu, and A. Kaufman. ZippyGPU: Programming Toolkit for General-Purpose Computation on GPU Clusters. In *GPGPU Workshop at Supercomputing*, 2006.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of ACM/IEEE Supercomputing Conference*, 2004.

[7] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[8] D. Luebke. NVIDIA's CUDA: Democratizing Parallel Computing. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[9] D. Luebke, M. Garland, J. Owens, and K. Skadron. GPGPU: ASPLOS 2008 CUDA TUTORIAL. http://www.gpgpu.org/asplos2008/.

[10] M. McCool, K. Wadleigh, B. Henderson, and H. Lin. Performance Evaluation of GPUs using the RapidMind development platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[11] A. Moerschell and J. Owens. Distributed Texture Memory in a Multi-GPU Environment. In *Graphics Hardware 2006*, 2006.

[12] NVIDIA. *NVIDIA CUDA Programming Guide v1.1*, 11 2007.

[13] NVIDIA. NVIDIA Tesla C870 - GPU Computing Processor. http://www.nvidia.com/object/tesla_gpu_processor.html, 12 2007.

[14] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Prucell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1), 2007.

[15] M. Papakipos. The PeakStream Platform. In *LACSI Workshop on Heterogeneous Computing*, 2006.

[16] S. Popov, J. Gunther, H. Seidel, and P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007.

[17] F. Qui, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller. Dispersion Simulation and Visualization for Urban Security. In *IEEE Visualization Proceedings*, 2004.

[18] D. Rivera, D. Schaa, M. Moffie, and D. Kaeli. Exploring Novel Paralellization Techniques for 3-D Imaging Applications. *Accepted to SBAC-PAD 2007: 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[19] E. Rollins. Real-Time Ray Tracing with NVIDIA CUDA GPGPU and Intel Quad-Core. http://eric_rollins.home.mindspring.com/ray/cuda.html.

[20] S. Ryoo, C. Rodrigues, S. Stone, S. Baghorkhi, S.-Z. Ueng, and W. Hwu. Program Optimization Study on a 128-Core GPU. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, 2007.

[23] S. S. Stone, H. Yi, W. W. Hwu, J. P. Haldar, B. P. Sutton, and Z.-P. Liang. How GPUs can improve the quality of magnetic resonance imaging. In *In The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[24] S. Yamagiwa and L. Sousa. Caravela: A Novel Stream-Based Distributed Computing Environment. In *Computer*, volume 40, pages 70–77, 2007.

[25] J. Zhang, W. Meleis, D. Kaeli, and T. Wu. Acceleration of Maximum Likelihood Estimation for Tomosynthesis Mammography. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing*, 2006.