IRis

Northeastern University

Electrical and Computer Engineering Master's Theses

Department of Electrical and Computer Engineering

January 01, 2009

Modeling execution and predicting performance in multi-GPU environments

Dana Schaa
Northeastern University

Recommended Citation

Schaa, Dana, "Modeling execution and predicting performance in multi-GPU environments" (2009). *Electrical and Computer Engineering Master's Theses.* Paper 32. http://hdl.handle.net/2047/d20000059

This work is available open access, hosted by Northeastern University.

MODELING EXECUTION AND PREDICTING PERFORMANCE IN MULTI-GPU ENVIRONMENTS

A Thesis Presented

by

Dana Schaa

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University

Boston, Massachusetts

August 2009

© Copyright 2009 by Dana Schaa

All Rights Reserved

Abstract

Graphics processing units (GPUs) have become widely accepted as the computing platform of choice in many high performance computing domains, due to the potential for approaching or exceeding the performance of a large cluster of CPUs with a single GPU for many parallel applications. Obtaining high performance on a single GPU has been widely researched, and researchers typically present speedups on the order of 10-100X for applications that map well to the GPU programming model and architecture. Progressing further, we now wish to utilize multiple GPUs to continue to obtain larger speedups, or allow applications to work with more or finer-grained data.

Although existing work has been presented that utilizes multiple GPUs as parallel accelerators, a study of the overhead and benefits of using multiple GPUs has been lacking. Since the overhead affecting GPU execution are not as obvious or well-known as with CPUs, developers may be cautious to invest the time to create a multiple-GPU implementation, or to invest in additional hardware without knowing whether execution will benefit. This thesis investigates the major factors of multi-GPU execution and creates models which allow them to be analyzed. The ultimate goal of our analysis is to allow developers to easily determine how a given application will scale across multiple GPUs.

Using the scalability (including communication) models presented in this thesis, a

developer is able to predict the performance of an application with a high degree of accuracy. For the applications evaluated in this work, we saw an 11% average difference and 40% maximum difference between predicted and actual execution times. The models allow for the modeling of both various numbers and configurations of GPUs, and for various data sizes—all of which can be done without having to purchase hardware or fully implement a multiple-GPU version of the application. The performance predictions can then be used to select the optimal cost-performance point, allowing the appropriate hardware to be purchased for the given applications needs.

Acknowledgements

I first want to thank Jenny Mankin for all of her *infinitely valuable* input and feedback regarding this work and all of my endeavors.

I also need to acknowledge the unquantifiable support from my parents, Scott and Vickie, my brother, Josh, and the rest of my extended family who have always been and still are there to help me take the next step.

Finally, I'd like to thank my advisor, Professor David Kaeli, for all of his time and effort.

This work was supported in part by Gordon-CenSSIS, the Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (Award Number EEC-9986821). The GPUs used in this work were generously donated by NVIDIA.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Introduction 1

1.1.1 Utilizing Multiple GPUs 2

1.2 Contributions 3

1.3 Organization of the Thesis 4

2 Related Work **5**

2.1 Optimizing Execution on a Single GPU 5

2.2 Execution on Multiple GPUs 7

2.3 CUDA Alternatives 9

3 CUDA and Multiple GPU Execution **12**

3.1 The CUDA Programming Model 13

3.1.1 Grids, Blocks, and Threads: Adapting Algorithms to the CUDA Model 13

3.1.2 Memory Model 17

3.2 GPU-Parallel Execution 19
3.2.1 Shared-System GPUs 19
3.2.2 Distributed GPUs 19
3.2.3 GPU-Parallel Algorithms 20

4 Modeling Scalability in Parallel Environments 22
4.1 Modeling GPUs with Traditional Parallel Computing 22
4.2 Modeling GPU Execution 23
4.3 Modeling PCI-Express 25
4.3.1 Pinned Memory 25
4.3.2 Data Transfers 26
4.4 Modeling RAM and Disk 28
4.4.1 Determining Disk Latency 28
4.4.2 Empirical Disk Throughput 30

5 Applications and Environment 33
5.1 Applications 33
5.2 Characterizing the Application Space 35
5.3 Hardware Setup 35

6 Predicting Execution and Results 37
6.1 Predicting Execution Time 37
6.2 Prediction Results 39
6.3 Zero-Communication Applications 40
6.4 Data Sync Each Iteration 41
6.5 Multi-read Data 42
6.6 General Performance Considerations 43

6.6.1 Applications Whose Data Sets Fit Inside RAM 44
6.6.2 Applications Whose Data Sets Do Not Fit Inside RAM 46

7 Discussion **51**
7.1 Modeling Scalability in Traditional Environments 51
7.2 Limitations of Scalability Equations 54
7.3 Obtaining Repeatable Results 55

8 Conclusion and Future Work **57**
8.1 Summary of Contributions 58
8.2 Future Work 59

Bibliography **60**

List of Figures

1.1 Theoretical GFLOPS for NVIDIA GPGPUs 2

3.1 The configurations of systems and GPUs used in this work. 13

3.2 GeForce 8800 GTX High-Level Architecture 14

3.3 Grids, Blocks, and Threads. 17

3.4 Memory hierarchy of an NVIDIA G8 or G9 series GPU. 18

4.1 PCI-Express configuration for a two-GPU system. 28

4.2 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. . . 31

6.1 Predicting performance for distributed ray tracing. Results are shown in Figure 6.3. 39

6.2 Convolution results plotted on a logarithmic scale. 47

6.3 Results for Ray Tracing across four data sets. 48

6.4 Results for Image Reconstruction for a single data size. 49

6.5 Distributed Matrix Multiplication using Fox's Algorithm. 49

6.6 Results for Image Reconstruction using a 10Gb/s network. 50

6.7 Results for Ray Tracing on a 1024x768 image using a 10Gb/s network. 50

List of Tables

4.1 Transfer throughput between CPU and GPU. *The throughput of 4 shared-system GPUs is estimated and is a best-case scenario. 27

4.2 Design space and requirements for predicting execution 31

Chapter 1

Introduction

1.1 Introduction

General purpose graphics processing units (GPGPUs) are now ubiquitous in the field of high performance computing (HPC) due to their impressive processing potential for certain classes of parallel applications. The current generation of GPGPUs has surpassed a teraflop in terms of theoretical computations per second. Due to this processing power, a single GPU has the potential to replace a large number of superscalar CPUs while requiring far less overhead in terms of cost, power/cooling, energy, and administration.

The benefits of executing general purpose applications on graphic processing units (GPUs) has been recognized for some time. Initially, algorithms had to be mapped into the graphics pipeline, but over the last decade APIs were created to abstract the graphics hardware from the programmer [6, 26, 33]. However GPUs were taken to the mainstream only with the the availability of standard C libraries using NVIDIA's CUDA programming interface—which was built for and runs on NVIDIA GTX GPUs.

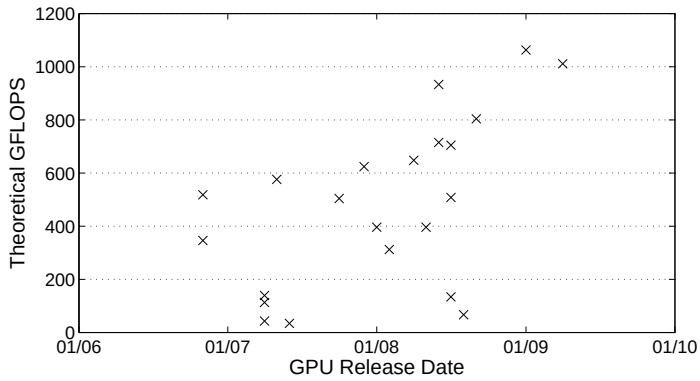


Figure 1.1: Theoretical GFLOPS for NVIDIA GPGPUs

Since its first release, a number of efforts have explored how to reap large performance gains on a CUDA-enabled GPUs [9, 13, 20, 28, 34, 36, 38, 39, 42].

1.1.1 Utilizing Multiple GPUs

The current trend in GPU research is to focus on low-level program tuning (see Chapter 2) to obtain maximum performance. However, there is always the need to perform faster, or to work with larger or finer-grained data sets. The logical next step is to target on multiple GPUs.

As the factors affecting performance on multiple GPUs are not as well known as with traditional CPUs, the benefit that can be gained from utilizing multiple GPUs is harder to predict. We have noticed a reluctance from developers to invest the time, effort, and money to purchase additional hardware and implement multi-GPU versions of applications.

To help identify when 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 execution for a given application (based on a single-GPU reference implementation) while varying the number of GPUs, their configuration, and the data set size of the application.

Execution on parallel GPUs is promising because applications that are best suited to run on GPUs inherently have large amounts of segmentable parallelism. By showing that multiple GPU execution is a feasible scenario, we help programmers alleviate many of the limitations of GPUs (such as memory resources, shared buses, availability of processing elements, etc.) and thus provide even more than the obvious speedup from execution across a larger number of cores. Of course, inter-GPU communication becomes an 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, 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.

This thesis focuses specifically on CUDA-enabled GPUs from NVIDIA, however the models and methodology that are presented can easily be extended to any GPGPU platform.

1.2 Contributions

The contributions of our work are as follows:

- Identification and classification of the major factors affecting execution in multiple-GPU environments.
- Models representing each of the major factors affecting multiple-GPU execution.
- A methodology for utilizing these models to predict the scalability of an application across multiple GPUs, GPU configurations, and data set sizes.
- An evaluation of six applications to show the accuracy of the performance prediction methodology and models.

1.3 Organization of the Thesis

The following chapter presents works related to topics covered in this thesis. These works are grouped into one of the following categories: high-performance computing with CUDA on single GPUs, computing on parallel GPUs, and alternative hardware and programming models. In Chapter 3, we provide an introduction to CUDA as relevant to this work. The topics specifically cover the NVIDIA GeForce series hardware and the ramifications of the threading and memory models. Considerations for execution on multiple NVIDIA GPUs using CUDA are also discussed. Chapter 4 then goes into detail about the models that we created that allow the prediction of execution times on multiple GPUs. Chapter 5 introduces the applications that we used to verify our predictions and also introduces our hardware testing environment. Chapter 6 presents the results from our study and Chapter 7 draws some final conclusions.

Chapter 2

Related Work

We divide the prior relevant work in the area of GPUs into three categories. Each is discussed below.

2.1 Optimizing Execution on a Single GPU

Ryoo et. al explore areas of the CUDA programming model and of GeForce hardware (such as the configuration of memory banks) that algorithms must consider to achieve optimal execution [39]. They conclude that overall performance is largely dependent on application characteristics—especially the amount of interaction with the CPU and the number of accesses to GPU global memory. Specific optimizations that they investigate are the word granularities of memory accesses, the use of on-chip cache, and loop unrolling.

In other work Ryoo et. al provide an investigation of the search space for tuning applications on an NVIDIA GeForce 8800 GTX GPU [38]. Among their findings was that the size and dimensions of threads blocks (detailed in Chapter 3) had a large impact in the utilization of the GPU functional units and therefore had a large impact

on performance. However, they note that different versions of CUDA did not receive the same performance benefit that they had achieved. They conclude that "even small changes in application or runtime software likely need a new search for optimal configurations". The implications of this finding is that code will not only have to be tuned for new generations of hardware, but also for software updates as well. This finding lends credibility to our approach that focuses on avoidance of fine-tuning code in exchange for portability.

Similar to Ryoo [38], Hensley et. al provide a detailed methodology for determining peak theoretical execution for a certain GPGPU application [16]. However, their work requires extensive insight into the underlying microarchitecture, including determining the number of fetches performed by a shader, checking the memory alignment of pixels, and sometimes "forcing" raster patterns to improve transfers. This is a useful exercise for programmers trying to squeeze performance from applications in a static environment, but in general may not be useful to scientific programmers and researchers who are not highly familiar with graphics programming and who want their algorithms to port across different hardware and software versions.

Work done by Jang et. al explores the optimization space of AMD RVxx GPUs, though their methodology is applicable to vector-based GPUs in general [20]. Their work focuses on the utilization of ALUs, fetch bandwidth, and thread usage. Their findings include the importance of using intrinsic functions and vector operations on resource utilization. Using the techniques presented they were able to improve the performance of their original GPU implementation between 1.3-6.7X for the applications they evaluated.

Mistry et. al present a phase unwrapping algorithm that uses CUDA as an accelerator for MATLAB [28]. Using the MEX interface, they offloaded an affine transform to

the GPU resulting in a 6.25X speedup over the optimized MATLAB implementation. They also presented an evaluation of overhead using C/CUDA-only approach and a MATLAB/MEX/CUDA approach and found that I/O efficiency increased enough from using the MEX interface to amortize the extra interaction requirements.

2.2 Execution on Multiple GPUs

As opposed to the work in Section 2.1 which targets optimized execution on a single GPU, the following 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 that they have implemented on the cluster include medical reconstruction, particle flow, and dispersion algorithms [12, 35]. Their work targets effective usage of distributed GPUs.

Moerschell and Owens describe the implementation of a distributed shared memory system to simplify execution on multiple distributed GPUs [29]. 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 [11]. 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.

Expanding on the two previous works, Stuart and Owens created a message passing interface for autonomous communication between data parallel processors [44]. Their interface avoids interaction with the CPU by creating communication threads that dynamically determine when communication is desired by polling requests from the GPUs. They use the abstraction of *slots* to avoid defining the unit of communication specifically as a thread, block, or grid (defined in Section 3.1.1). The ability to communicate between GPUs without direct CPU interaction is very desirable, and perhaps will be supported by hardware in the future. However other factors, such as algorithm complexity, will definitely increase, especially with communication between large numbers of blocks and threads that are physically confined to blocks, and abstractly confined to warps. Also, it is not clear whether allowing arbitrary communication is useful in GPGPU environments, as non-deterministic communication will likely devastate performance. Although an interface allowing arbitrary communication is a more robust solution, practically-speaking only fixed, very normal communication will fit with the current GPGPU model.

Strengert et. al created an extension to the CUDA programming API called CUDASA that includes the concepts of *jobs* and *tasks*, and automatically handles the distribution of CUDA programs over a network and the multiple threads on each system [43]. They chose to use a distributed shared memory paradigm for managing distributed processes. While they were able to obtain good speedups for multiple GPUs connected to a single system, their distributed performance was not as strong. Our investigation of the communication factors of multiple-GPU interaction helps to

explain why they saw the results for the algorithms they chose.

Caravela is a stream-based computing model that incorporates GPUs into GRID computing and uses them to replace CPUs as computation devices [46]. While the GRID is not an ideal environment for general purpose scientific computing, this model may have a niche for long-executing algorithms with large memory requirements, as well as for researchers who wish to run many batch GPU-based jobs.

Given this growing interest in exploiting multiple GPUs and demonstrations 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 a model for determining 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.

2.3 CUDA Alternatives

CUDA has become the defacto language for high performance computing on GPUs because it allows programmers to completely abstract away the graphics—simply supporting the C standard library and adding some new data types and functions that are specific to tasks such as allocating memory on the GPU, transferring data from CPU to GPU, etc. It's success has had a large impact on industry, perhaps most importantly, effort has been made to create an open standard for programming on many-core GPUs (called OpenCL). Reviewing the OpenCL standard, the influence of CUDA is easily recognized. The standard itself is quite similar to the CUDA programming model, though some terminology and concepts are more generic. Creating

a standard for execution on many-core GPUs is a bit trickier than other computing standards, because GPU hardware varies greatly by manufacturer, and knowing and understanding the hardware model has a large impact on performance. For example, since GeForce hardware requires manual caching of data (limited to 16KB), programmers will structure their data into blocks that fit nicely in cache. Since GPGPU is relatively new, and programming models and compilers are still evolving, it isn't entirely clear how well the OpenCL standard will be received, and how performance will be affected due to the abstraction.

In terms of raw processing power and target market, AMD is NVIDIA's most direct competitor. Currently their Radion HD 4890 GPU can execute a theoretical 1.36TFLOPs, edging out NVIDIA's top single-GPU, and they were the first manufacturer to create GPUs with hardware that can handle double-precision operations–something very significant in the HPC world. However, AMD has trailed in their programming model. They have acquired Brook [6], a stream-based programming language, and also have a strong assembly language interface (which NVIDIA is lacking), yet they are still having trouble competing with NVIDIA due to the simplicity of CUDA's C language support.

Brook+ is AMD's implementation of Brook with enhancements for their GPUs. The language is an extension of C/C++, with a few conceptual differences from CUDA. A notable exception is the idea of *streams* which are defined as data elements of the same type that can be operated on in parallel. Instead of explicitly threaded programming as with CUDA, the data itself defines the parallelism here. Brook+ code is also initially compiled to an intermediate language, where it can receive another round of optimization targeting the GPU [21].

Sony, Toshiba, and IBM collaborated to create the Cell processor (the chip used

to power the Playstation 3), which fits into the GPU market as well [17, 45]. The cell is comprised of a fully functional Power processor, and 8 Synergistic Processing Elements (SPEs). Each SPE contains multiple pipelines and operates using SIMD instructions. Also, a fast ring interconnect facilitates fast transfers of data between SPEs. The largest problem with the Cell is its complex programming model that requires, among other things, the programmer to explicitly program using low-level DMA intrinsics for moving data. The Cell trades programmability for efficiency and requires the programmer to work with low level instructions to obtain high performance results.

Finally, Intel's attempt to enter the GPU market is a many-core co-processor called Larrabee. Larrabee uses a group of *in-order* Pentium-based CPUs (which require considerably less area and power than the latest superscalar processors) to execute many tasks in parallel. The advantages of Larrabee are that it can execute x86 binaries, it has access to main memory and disk, and the cores share a fully coherent L2 cache (many of these issues are discussed in Chapter 4). Since Larrabee supports the full x86 instruction set, it is even possible to run an operating system on each Larrabee core. Larrabee's Vector Processing Unit (VPU) has a width of 16 units that can execute integer, single-precision, and double-precision instructions [40].

Chapter 3

CUDA and Multiple GPU Execution

This chapter provides a brief overview of the CUDA programming model and architecture with emphasis on factors that effect execution on multiple GPUs. Those seeking more details on CUDA programming should refer to the CUDA tutorials provided by Luebke, et. al [24]. Also, Ryoo et. al provide a nice overview of the GeForce 8800 GTX architecture, and also present strategies for optimizing performance [39]. 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** - We use this term to define a networked group of distributed systems each containing a single GPU. In reality, there is no reason that each system has to contain a single GPU, and this is something that will be investigated in future work.

- **Shared-system GPUs** - A single system containing multiple GPUs which communicate through a shared CPU RAM (such as the NVIDIA Tesla S870

server [31]).

- **GPU-Parallel Execution** - Execution which 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.

(a) Distributed GPUs. (b) Shared-System GPUs.

Figure 3.1: The configurations of systems and GPUs used in this work.

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 Grids, Blocks, and Threads: Adapting Algorithms to the CUDA Model

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

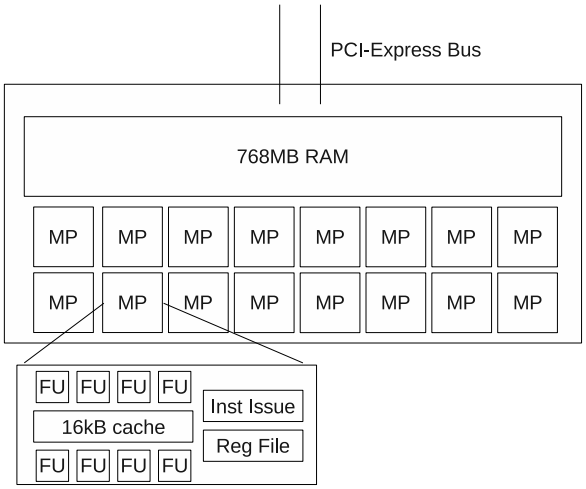


Figure 3.2: GeForce 8800 GTX High-Level Architecture

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 multiple GPUs because of their high degree of data-level parallelism. Further, since applications have to be partitioned into (quasi-independent-)blocks, this model lends itself well to execution on multiple GPUs since the execution of one block should not affect another (though this is not always the case).

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.

Threads

CUDA programming uses the SPMD model as the basis for concurrency, though they refer to each data stream as a *thread*, and call their paradigm Single Program Multiple Thread (SPMT). This means that each CUDA thread is responsible for an independent flow of program control. NVIDIA's decision to use the SPMT model allows parallel programmers who are experienced with writing multithreaded programs to feel very comfortable with CUDA. However, while it is true that CUDA threads are technically independent, in reality performance of a CUDA program is heavily reliant on groups of threads executing identical instructions in lock-step.

Despite CUDA's SPMT model, NVIDIA's GPU hardware is build using SIMD multiprocessing units. Threads are grouped into units of 32 called *warps*. A warp is the basic schedulable unit on a multiprocessor, and all 32 threads of a warp must execute the same instruction, although their data is different. Multiprocessors have 8 functional units (exaggeratedly called *cores*) which perform most operations in 4 cycles. In the first cycle, the first 8 threads (threads 0-7) enter their data in the pipeline. This is followed by a context switch which activates the next 8 threads (8-15). In the second cycle, these threads enter their data into the pipeline. The third and fourth groups then follow in suit. On the fifth cycle, the first threads have their results, and are ready to execute the next instruction. If there are not enough threads to fill all 32 places in the warp, these cycles are wasted.

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

Sometimes it will happen that threads in a warp will reach a conditional statement (e.g. an if statement) and take different paths. In this case, the flow of instructions will diverge and some threads will need to execute instructions inside the conditional while other threads will not. To deal with this, the instructions inside the conditional are executed on the multiprocessor, but threads that shouldn't execute are masked off and simply sit idle until the control flows converge again.

Blocks and Grids

In CUDA, a block is the unit of schedulability that can be assigned to a multiprocessor. A block is comprised of an integer number of warps, and at any given time may only be assigned to at most one multiprocessor.

Although warps are significant in terms of throughput, they are effectively transparent to the programming model. Instead, CUDA models threads as either 1-D, 2-D, or 3-D structures (blocks) which are the schedulable units of execution of a multiprocessor. If all threads of a block are waiting for a long-latency memory read or write to complete, then CUDA may schedule another block to execute on the same multiprocessor. However, CUDA does not swap the register file or shared cache when blocks change, so if a new block runs while another is waiting, it must be able to work with the resources that are still available.

When a CUDA program (called a *kernel*) is executed on the GPU, each thread has certain intrinsics that are automatically populated. These values include its block's coordinates within the grid, and its thread's coordinates within the block. A common practice for mapping threads to problem sets is to have one thread responsible for each element in the output data. To do this, the dimensions of the blocks and threads are usually structured to mirror the dimensions of the output data set (commonly a

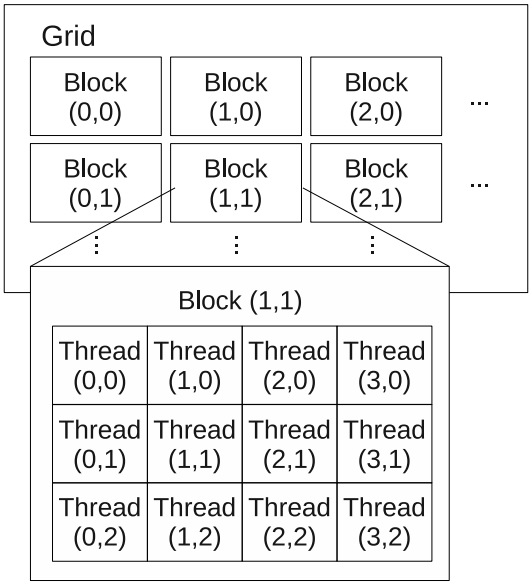


Figure 3.3: Grids, Blocks, and Threads.

matrix).

3.1.2 Memory Model

Main memory on the G80 Series GPUs is a large RAM (0.5-1.5GB) 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 its shared caches.

GPUs can not directly access host memory during execution. Instead, data is explicitly transferred between device memory and host memory prior to and following GPU execution. Since manual memory management is required for GPU execution

(there is no paging mechanism), and because GPUs cannot transfer data between GPUs and CPUs 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 introduces stalls in execution.

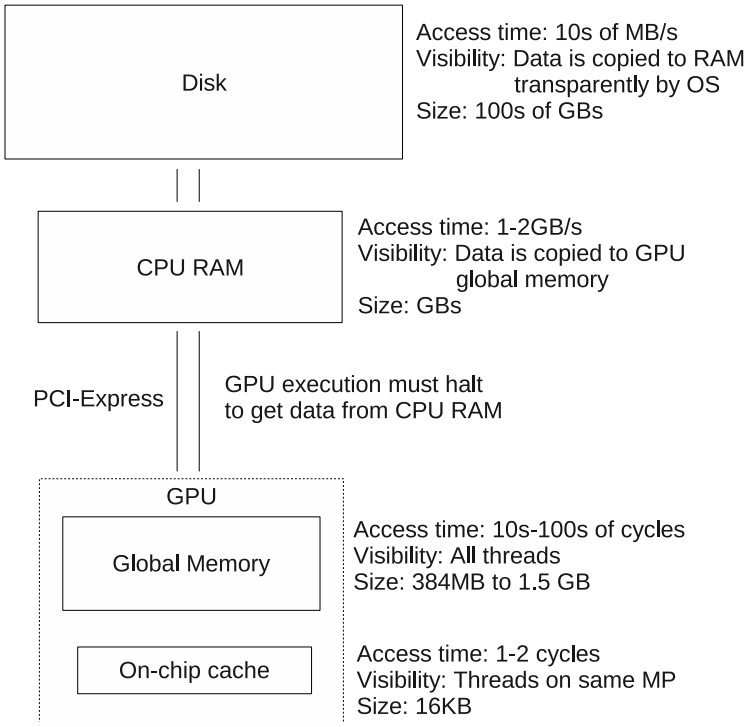


Figure 3.4: Memory hierarchy of an NVIDIA G8 or G9 series GPU.

As with traditional parallel computing, using multiple GPUs provides additional resources, potentially requiring fewer GPU calls and allowing the simplification of algorithms. However, compounding data transfers and execution breaks with traditional parallel computing communication costs may squander any benefits reaped

from parallel execution. These issues (and others related to parallel GPU communication) are discussed in detail in Chapter 4.

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 distributed process interacts with the GPU in the same manner as a single-GPU application.

Just as in traditional parallel computing, distributed GPUs scale better than their shared-system counterparts because they 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. This restriction has inspired researchers to implement software mechanisms that allow inter-GPU communication without having to explicitly involve the CPU thread (though none are widely used) [29, 11, 44].

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 device memory, the algorithm needs to be modified so that the data can be exchanged with main memory of the CPU. 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.

As an example, consider a matrix multiplication algorithm. If the two input matrices and one output matrix are too large to fit in global memory on the GPU, then the data will need to be partitioned and multiple GPU calls will be required. To partition the data, we divide the output matrix into blocks. For a given call to the GPU we then need to transfer only the input data needed to compute the elements contained in the current block of output data. In doing so, we have essentially created

a multi-threaded program that runs on a single-threaded processor. With multiple GPUs connected to the same system, almost no modification is needed to have these threads run in parallel on different GPUs as opposed to serially (one after the other) on the same GPU. Similarly, all that would be needed to have the algorithm run on distributed GPUs is the MPI communication code. Therefore, this GPU code can be easily modified to allow it to run on multiple GPUs.

The memory (and other resource) limitations of GPU therefore make the transition to GPU-parallel execution a very natural next-step for further performance gains.

Chapter 4

Modeling Scalability in Parallel Environments

4.1 Modeling GPUs with Traditional Parallel Computing

We initially take a simple view of the traditional parallel computing model in which speedup is obtained by dividing program execution across multiple processors, and some overhead is obtained in the form of communication. In general, 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 4.1. In this equation, 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,

which t_{cpu_comm} is the inter-CPU communication requirement. Equation 4.2 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 4.2, 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.

$$t_{total} = t_{cpu} + t_{cpu_comm} + t_{gpu} + t_{gpu_comm} \tag{4.1}$$

$$t_{cpu_comm} = \begin{cases} t_{memcpy} & \text{for shared systems} \\ t_{network} & \text{for distributed systems} \end{cases} \tag{4.2}$$

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 4.2, and t_{gpu_comm} represents additional communication overhead and is discussed in Sections 4.3 and 4.4.

Using these factors, we provide a methodology which 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 or fully implementing the parallelized application.

4.2 Modeling GPU Execution

Our methodology requires that a CUDA program exists which 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 model this 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 (and therefore inter-GPU communication) is highly non-deterministic. 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 multiple GPUs all of the same model (we will allow for heterogeneous GPU modeling in our future work).

Using our approach, we first need to determine how GPU execution scales on N 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 grain and dimensions of the datasets 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 diving 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 4.3. The total execution time across N GPUs can then be represented by Equation 4.4. 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.

$$t_{element} = \frac{t_{ref_gpu}}{N_{elements}} \tag{4.3}$$

$$t_{gpu} = t_{element} * \left\lceil \frac{N_{elements}}{M_{gpus}} \right\rceil \tag{4.4}$$

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 which might suffer from precision due to their length). Equation 4.5 is then the execution time of N subsets across M GPUs.

$$t_{gpu} = t_{subset} * \left\lceil \frac{N_{subsets}}{M_{gpus}} \right\rceil \tag{4.5}$$

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.

4.3 Modeling PCI-Express

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

4.3.1 Pinned Memory

The CUDA driver supports allocation of memory that is pinned to RAM (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 4.7) is much more expensive than the traditional method of requesting pageable memory from the kernel. Measured pinned requests take 0.1s on average.

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. As such, allocating pinned memory for use with the GPU should only be done when the entire data set can fit in RAM.

4.3.2 Data Transfers

In order to increase the problem set, and because of the impact of multiple GPUs, our shared-systems are equipped with GeForce 8800 GTX Ultras. The Ultra GPUs are clocked higher 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. However, since our main goal is to predict execution correctly for any system, the choice of using Ultra GPUs is arbitrary.

The transfer rates from both pinned and pageable memory in CPU RAM to a GeForce 8800 GTX and an GeForce 8800 GTX Ultra across an 16x PCI-e bus are

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	*348MB/s
8800 Ultra	1	pinned	3182MB/s
8800 Ultra	2	pinned	1389MB/s
8800 Ultra	4	pinned	*695MB/s

Table 4.1: Transfer throughput between CPU and GPU. *The throughput of 4 shared-system GPUs is estimated and is a best-case scenario.

shown in Table 4.1. Pinned memory is faster because the CUDA driver knows the data's location in CPU RAM and does not have to locate it, 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 4.1 shows measured transfer rates for one and two GPUs, and extrapolates the per-GPU throughput to four Ultra GPUs in a shared-bus scenario.

This communication overhead must be carefully considered, especially for algorithms with large data sets which 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 4.1 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).

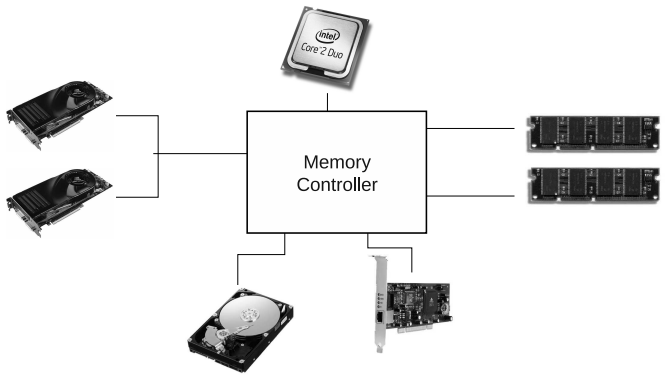


Figure 4.1: PCI-Express configuration for a two-GPU system.

4.4 Modeling RAM and Disk

4.4.1 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 4.6 shows that the time to transfer data from disk to memory varies based 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 4.6(a) shows that when data is smaller than RAM, no paging is necessary. In Equation 4.6(b), a fraction of the data resides on disk and must be paged in, and in Equation 4.6(c) all of the data from disk 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.

$$t_{disk'} = \begin{cases} 0 & x < B_{RAM} & (a) \\ \frac{x - B_{RAM}}{T_{disk}} & B_{RAM} < x < B_{RAM} + B_{transfer} & (b) \\ \frac{B_{transfer}}{T_{disk}} & B_{RAM} + B_{transfer} < x & (c) \end{cases} \tag{4.6}$$

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

The following provides the flow of the model which is used in this work to accurately predict disk access times. 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 valid upper bound on transfers.

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 determined by Equation 4.6.

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 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 of these 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 presented in Equation 4.7, and are algorithm-specific.

4.4.2 Empirical Disk Throughput

In order to predict execution, we measure disk throughput for a given system. To determine throughput (and to verify Equation 4.6), we run a test which varies the total amount of data allocated, while transferring a fixed amount of data from disk to GPU.

Figure 4.2 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 (∼3.8GB on our 4GB system due to 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 determining PCI-e and disk throughput. These factors combine to

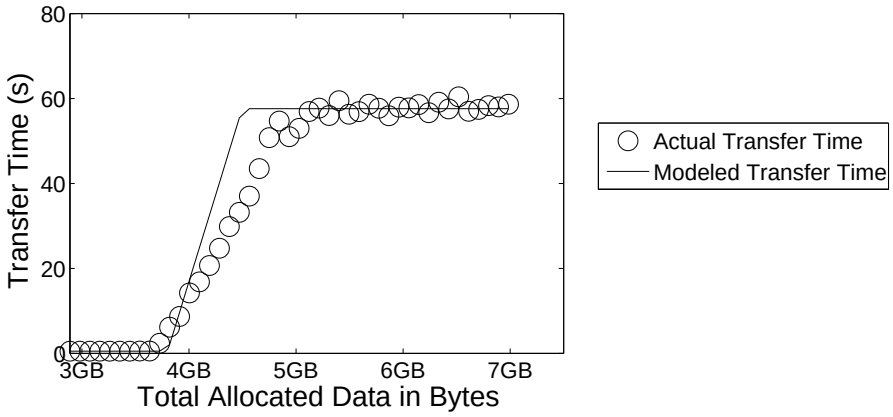


Figure 4.2: Time to transfer 720MB of paged data to a GeForce 8800 GTX GPU, based on the total data allocation on a system with 4GB of RAM.

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 4.2: Design space and requirements for predicting execution

make up Equation 4.7, 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 4.1.

$$t_{gpu_comm} = \begin{cases} t_{pinned_alloc} + t_{pci-e} & \text{pinned memory} \\ t_{disk} + t_{pci-e} & \text{pageable memory} \end{cases} \tag{4.7}$$

It should be noted that we do not need to model RAM transfers related to transfers, 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 4.2 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.

Chapter 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 multi-GPU implementation of the algorithm. We then compare the results to actual execution of multiple-GPU implementations 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 which uses tomography to reconstruct a three-dimensional volume from multiple two-dimensional x-ray views. 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 [37]. In his single GPU implementation, each pixel value is computed by a independent thread which contributes to a global state that is written to a frame buffer. In the multiple-GPU version, each pixel is still independent, but after each iteration the location of 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 utilizes 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 [10]. 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 Characterizing the Application Space

When we present our results in following chapter, we do so by grouping the applications based on their communication characteristics. We do this in order to draw some meaningful conclusions about the applications with similar execution characteristics when running in a multi-GPU environment. However, environment specific variables prevent us from drawing absolute conclusions about the execution of a specific application and data set. For example, both the Image Reconstruction and Ray Tracing applications show better performance with shared-system GPUs than with distributed GPUs. Still, if we decreased the size of our system RAM, increased the data set size to greater than RAM, or increased the network speed, this will cease to be the case. A brief investigation of these factors is presented at the end of Section 6.

5.3 Hardware Setup

For the multiple-GPU 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 system used in multithreaded experiments has an 2.4GHz Intel Core2 processor with 4GB of RAM. This system is equipped with a 612 MHz GeForce 8800 Ultra with 768GB of on-board RAM. All systems are running Fedora 7 and use a separate graphics card to run the system display. Running with a separate display

card is critical for GPU performance and repeatability because without it part of the GPUs resources and execution cycles are required to run the system display. If a separate GPU is not available on the system to run the display, then the X server should be stopped and the program should be invoked from a terminal.

Chapter 6

Predicting Execution and Results

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 multiple-GPU version (which also includes the communication scheme). We utilize the equations and methodology described in Chapter 4. This provide us with the GPU and CPU execution costs, the PCI-e transfer cost, and the network communication cost. Given the number of processors and particular data sizes, we are able to create predict the execution time for any number of GPUs for a particular configuration, even when we vary the data set size. 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 increases at a rate of $log_2 N$, where N is the number of GPUs rounded up to the nearest power of two. Factors such as this are implementation specific and must be considered.

Figure 6.1 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 for 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 $\frac{1}{P^{th}}$ 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 which 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 is 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 6.3).

Figures 6.2 through 6.7 were created using the same technique described here. These figures allow us to visualize the results from our predictions, and easily choose

```
# ------------ Distributed Ray Tracing ------------
P = 16                      # Number of GPUs
XDIM = 1024                 # Original image width (pixels)
YDIM = 768                  # Original image height (pixels)
SCALE = 4                   # Scale the image to 4X the original size
GLOBAL_STATE_SIZE = 13548   # In Bytes

for i = 1 to SCALE          # Loop over scale of image
 for j = 1 to P             # Loop over number of GPUs
  ROWS = (i * XDIM / j)     # Distribute the rows
  COLS = i * YDIM
  DATA_PER_GPU = ROWS * COLS * PIXEL_SIZE

  GPU_TIME = (i^2 * 0.314 / j)
  CPU_TIME = 0
  NET_TIME = step(j - 1) * (DATA_PER_GPU * (j - 1) / NET_BAND +
     GLOBAL_STATE_SIZE * (j - 1) / NET_BAND)
  DISK_TIME = 0
  PCIE_TIME = (DATA_PER_GPU + GLOBAL_STATE_SIZE) / PCIE_BAND

  FPS[j,i] = 1 / (GPU_TIME + CPU_TIME + NET_TIME + DISK_TIME +
     PCIE_TIME)
```

Figure 6.1: Predicting performance for distributed ray tracing. Results are shown in Figure 6.3.

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 worst-case prediction error 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 4.6 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 others (which involve disk paging) 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.

6.3 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 which 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 6.2 shows the predicted and empirical results for the convolution application.

Figure 6.2(a) shows that when running on multiple systems, distributed GPUs prevent paging which 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 6.2(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 6.2 shows that it is very important for application data sets to fit entirely in RAM if we want to effectively exploit GPU resources. 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.

6.4 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 above, these threads must update the global state, and this update must be synchronized so that it is completed before the next iteration begins.

For applications that possess this pattern, shared-system implementations have the advantage that as soon as data is transferred across the PCIe bus back to main memory, the data it is available to all other GPUs. In applications such as ray

tracing (Figure 6.3) and image reconstruction (Figure 6.4), where a large number of iterations need to be computed, the shared-system approach shows more potential for performance gains than distributed GPUs. Note that in our example, 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 6.2, and the distributed GPUs would likely outperform the shared-system GPUs since network latency is much shorter than disk access latency.

The step-like behavior illustrated by the distributed image reconstruction algorithm in Figure 6.4 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 which are 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 can be used without adding any communication costs.

Since the amount of data that must be transferred to the GPU in ray tracing application is relatively small, the shared-system algorithm tends to scale nicely (i.e., almost linearly) as more GPUs are added. The expected PCI-e bottleneck does not occur because the amount of data transferred is small in comparison to the GPU execution time.

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

When using the distributed FFT algorithm, t_{CPU} (from Equation 4.1) plays a large role due to the cost of computing matrix transposes, which we chose to leave on the CPU. Since transposing algorithms are 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.

6.6 General Performance Considerations

Since we are considering such a large design space, it is impossible to conclude that a certain class of application will always run better with a certain configuration of GPUs. For instance, assume that we had an application with data that will not fit entirely in CPU RAM. Our instinct is to say that we should probably use enough distributed systems so that data does not have to be paged in from disk. In systems where disk access is fast, the network speed is slow, or where the amount of network communication required becomes outrageously large, our instinct would be wrong.

If we could generalize the best configuration for an application based on some

high-level characteristics, then the models and methodology presented in this work would not be required. However, using some information gained from the models, and limiting the problem space a bit, we can begin to provide so even simpler guidelines for determining the best GPU configuration (although we will not be able to predict the exact performance or even the optimal number of GPUs). Since we have three main parameters that affect the choice of configuration (RAM size, network speed, and PCI-e bandwidth), we will hold RAM size constant and provide guidelines for choosing a proper configuration based on the ratio between network speed and PCI-e bandwidth.

6.6.1 Applications Whose Data Sets Fit Inside RAM

When an applications data set fits entirely in the RAM of a CPU, we do not need to worry about overhead from paging data to and from disk. For these applications, the choice between distributed and shared-system GPUs will then usually depend on the amount of data that must be transferred over the network (and at what speed), and the amount of data that must be transferred over the PCI-e bus. Recall that in shared systems, transfer time across the PCI-e bus scales approximately with the number of GPUs.

$$t_{pcie} = \frac{B_{pcie}}{T_{pcie}} N_{gpus} \tag{6.1}$$

$$t_{network} = \frac{B_{pcie}}{T_{pcie}} + \frac{B_{network}}{T_{network}} \tag{6.2}$$

In Equations 6.1 and 6.2, B and T represent number of bytes and throughput respectively. If t_{pcie} is smaller, then a shared-memory configuration is probably a better choice. Alternatively, if $t_{network}$ is smaller, then a distributed configuration is

probably best. Notice that the equations above can represent each iteration of an iterative application by adding I to both sides. However these will obviously cancel out, resulting in the equation shown. The assumption we make here is that there is not significant execution on the CPU that overlaps with GPU execution. If CPU execution is present, its characteristics may also play a role in determining which configuration is best.

Based on the results from our "Data Sync Each Iteration" applications, we would be tempted to conclude that applications which fit entirely in RAM are better suited for shared-system GPU configurations. However, Figure 6.6 shows our predictions for image reconstruction using a 10Gb/s network. The 10Gb/s network is nearly identical to the speed of the PCI-e bus and the figure shows that there are only a few seconds difference in predictions for shared and distributed GPUs. Accounting for the current limitation of at most 4 GPUs connected to the same shared-system, the distributed configuration may indeed be more attractive.

The results from ray-tracing given the faster network are shown as well in Figure 6.7. From the figure, we can see that for the given data set size, the PCI-e bus begins to saturate at about six GPUs, or about 18 frames per second (FPS). As mentioned previously, the current maximum for the number of GPUs connected to a single system is 4. So despite our prediction, the maximum performance that can be obtained is really closer to 13 FPS.

On the other hand, there is no limit on the number of systems that can be connected together (with the right network hardware), and we show that 16 systems each with a single GPU and a 10Gb/s network can obtain above 40 FPS with no sign of scalability issues.

6.6.2 Applications Whose Data Sets Do Not Fit Inside RAM

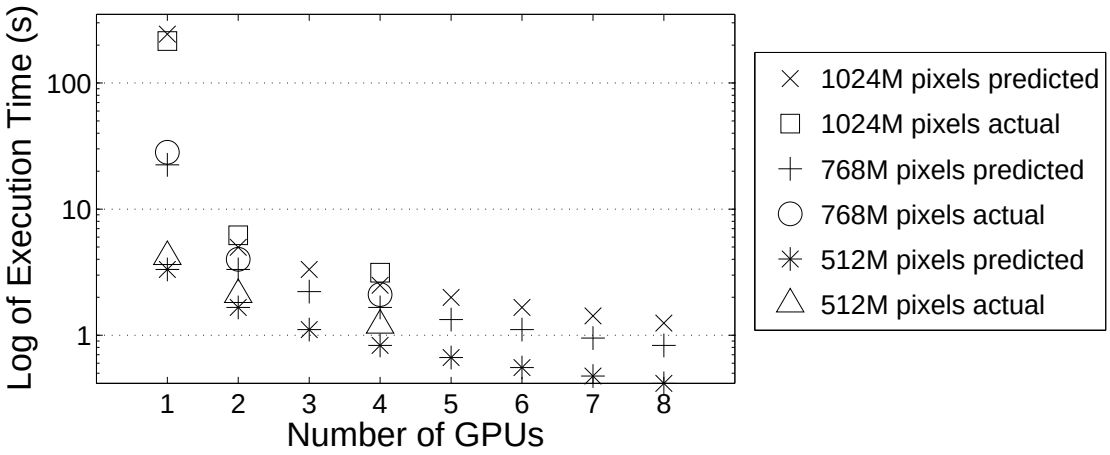
When an application's data set does not fit entirely in RAM, paging in data from disk will be required. Based on the rules for paging described in Chapter 4, this will drastically affect performance each time a transfer is made between GPU and CPU memories. Using more distributed CPUs will often be able to alleviate the need for paging, but will add overhead in the form of network communication. The amount of data that must be transferred, as well as the bandwidths of the disk and the network must be considered in order to determine the best configuration for an application and data size.

Using the aforementioned factors, Equations 6.3 and 6.4 describe the relationship that must be considered when choosing a configuration.

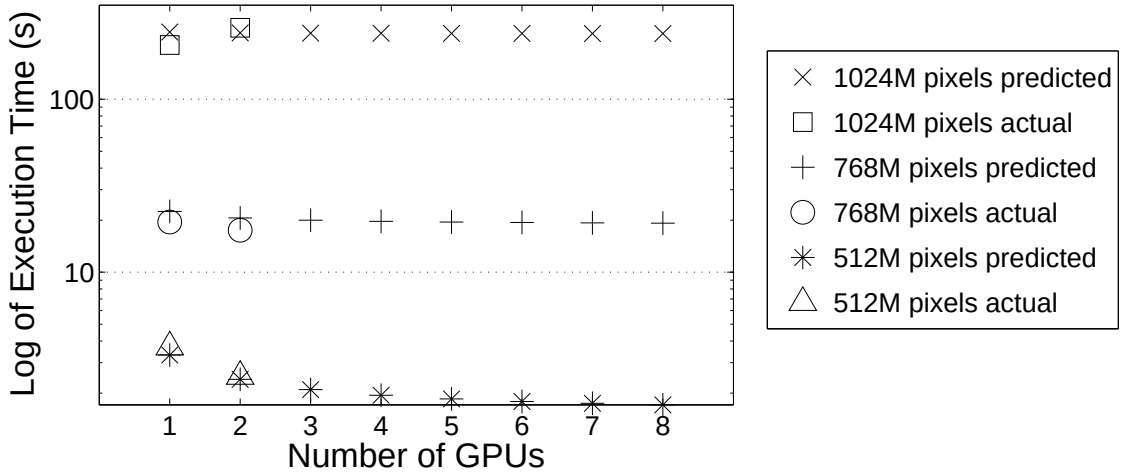
$$t_{network} = \frac{B_{network}}{T_{network}} \tag{6.3}$$

$$t_{disk} = \frac{B_{disk}}{T_{disk}} \tag{6.4}$$

In Equation 6.3 and 6.4, B and T again represent the total number of bytes of to be transferred, and the throughput (bandwidth), respectively. If $t_{network}$ is less, then a distributed approach is probably the best choice, otherwise a shared-system approach will likely be better. Only if the two numbers are similar should we need to also consider the PCI-e overhead as well.

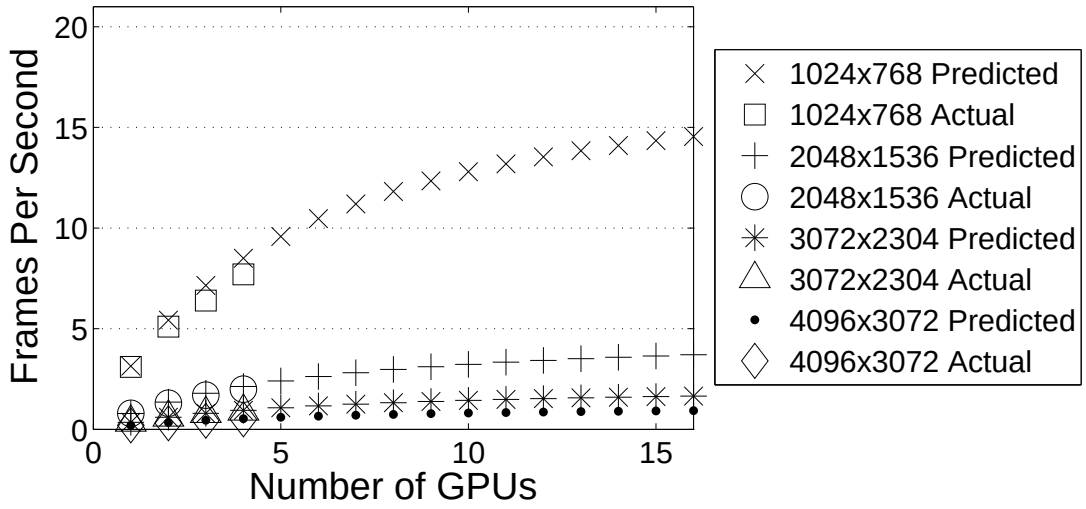


(a) Convolution with Distributed GPUs.

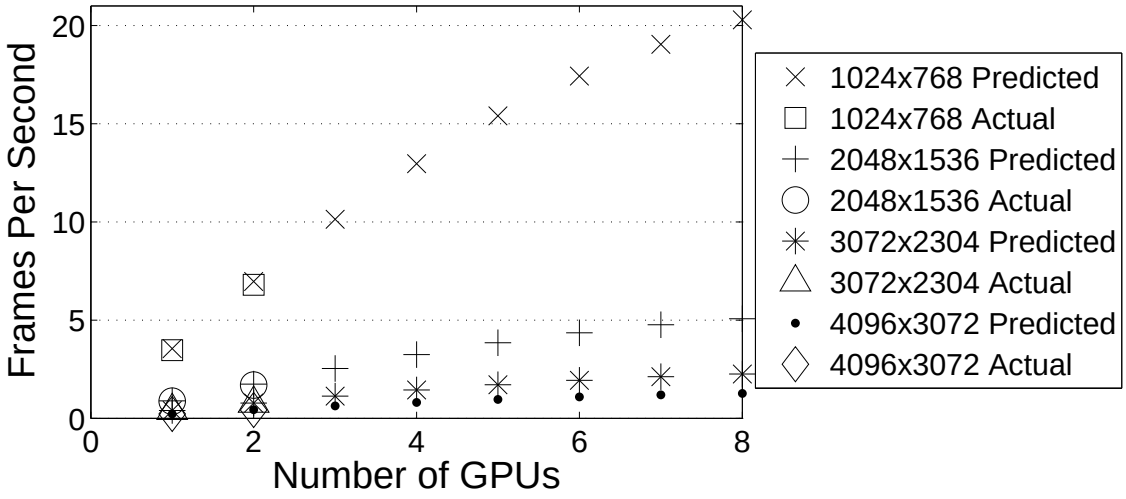


(b) Convolution with Shared-System GPUs

Figure 6.2: Convolution results plotted on a logarithmic scale.



(a) Distributed GPUs



(b) Shared-System GPUs

Figure 6.3: Results for Ray Tracing across four data sets.

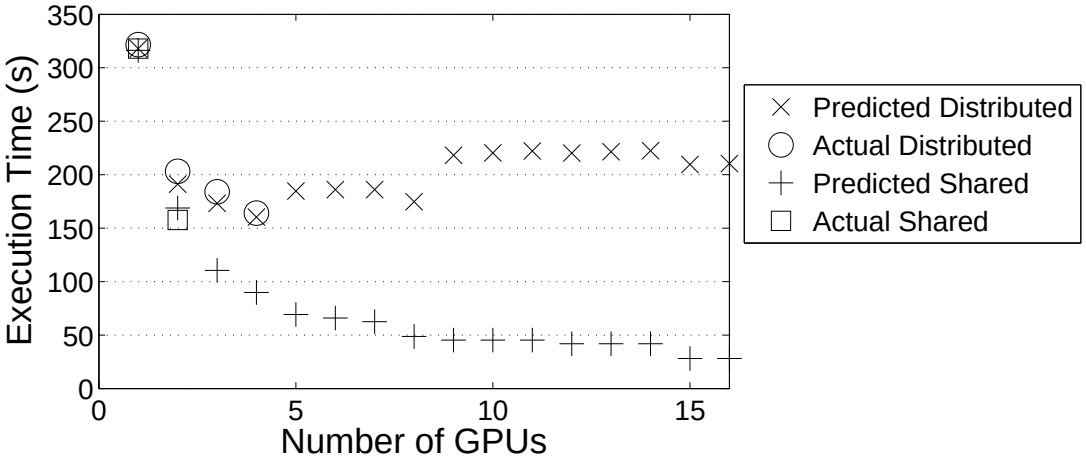


Figure 6.4: Results for Image Reconstruction for a single data size.

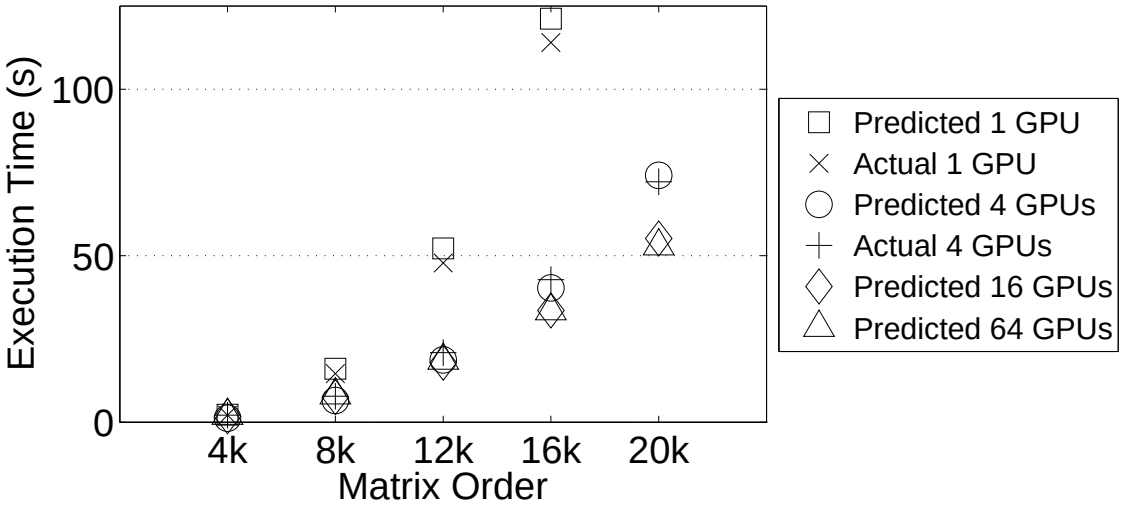


Figure 6.5: Distributed Matrix Multiplication using Fox's Algorithm.

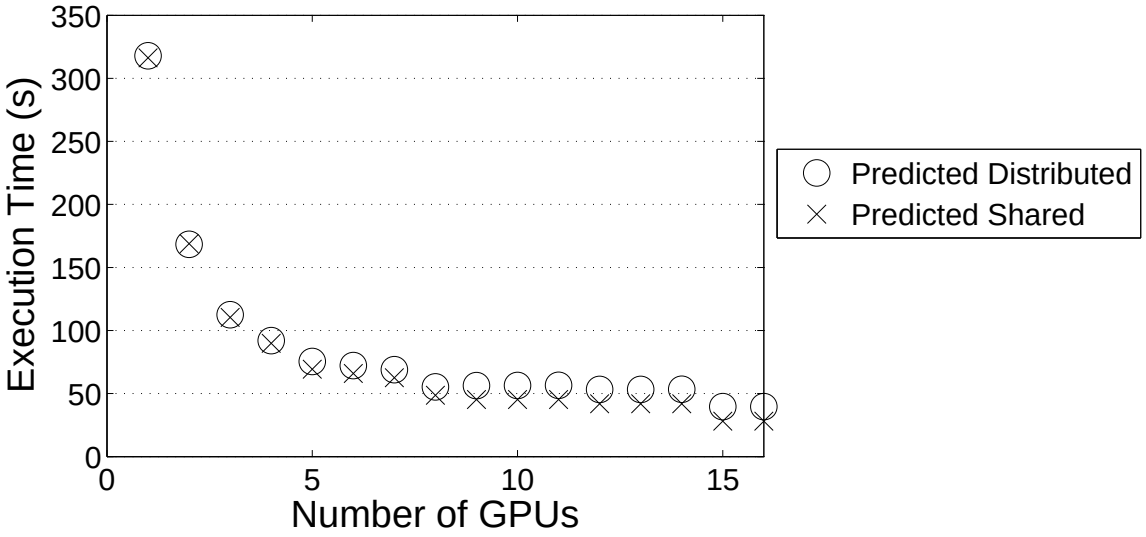


Figure 6.6: Results for Image Reconstruction using a 10Gb/s network.

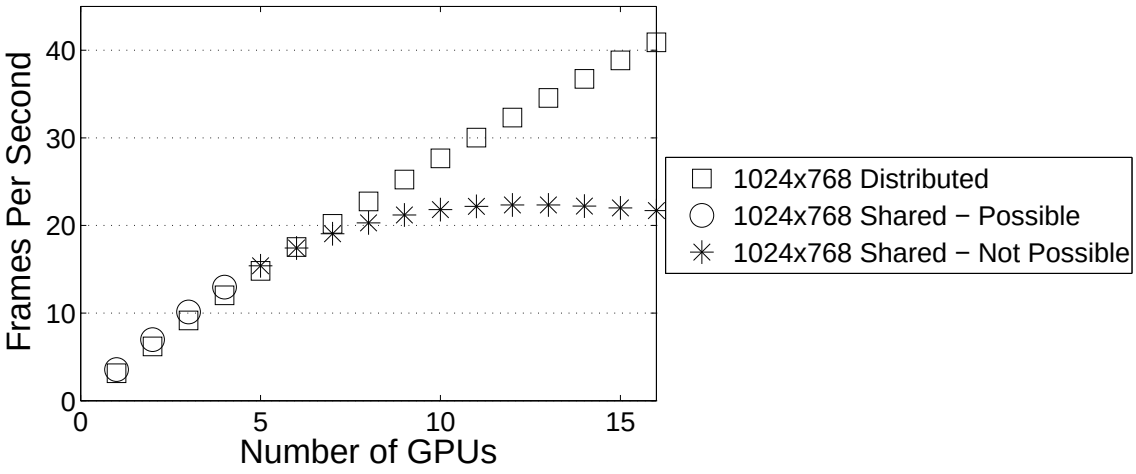


Figure 6.7: Results for Ray Tracing on a 1024x768 image using a 10Gb/s network.

Chapter 7

Discussion

7.1 Modeling Scalability in Traditional Environments

Performance prediction for parallel programs involves creating models to represent the computation and communication of applications for massively parallel systems, or within a network of workstations [7, 4]. However, the large number of factors that significantly affect performance as problem size or number of processing elements are increased makes modeling parallel systems very difficult.

The tradeoff when selecting a performance prediction model is that more accurate models usually require a substantial amount of effort, and are often specific to only a single machine. The remainder of this section discusses the stengths and weaknesses of tradtional models broken-down to a system-level, and identifies how our approach deals with each in turn.

Computation. Predicting the time spent executing instructions is usually based around a source code analysis of the program; dividing execution into basic blocks

that can then be modeled. In order to get an actual execution time prediction, some portion of the code also needs to be run on the target hardware or simulated using a system-specific simulator. Expecting accurate prediction times is very optimistic because of the effects of many factors (such as cache misses or paging) that cannot easily be determined as processing elements and data set sizes increase. Some models use statistics to provide a confidence interval for execution, though these models require the user to provide system and application specific probabilities for the timing of many of the computational or memory components [7].

When working in a GPU environment, the architecture dictates that each block of parallel threads is scheduled to a dedicated multiprocessor and remains there until execution completes. Also, since each block must execute in a SIMD fashion, the thread block will likely follow a very regular and repeatable execution pattern. This simplified view of execution makes it easier to model the scalability of parallel threads as we add additional multiprocessors or modify the data set size. An analysis of GPU execution scalability is provided in Section 4.2.

Memory Hierarchy. The memory hierarchy is extremely sensitive to hard-to-determine (and even harder to model) factors that change significantly as the problem size or number of processing elements is increased. These factors include the sensitivity of the memory hierarchy to contention (in shared caches, main memory, and disks), and the effects observed after the capacity of a specific memory component is reached.

In fact, we have found no realistic attempt to incorporate the actual effects of paging with disk when predicting the performance of algorithms on traditional computing systems. This is likely due to the potential for egregious performance differences caused by the nature of the access to disk, the variable latency time due to its

mechanical properties, the effects of contention, the potential need for paging before reading, etc.

As with modeling computation in GPU environments, the effects on the memory hierarchy are much more deterministic on a GPU as well. First, each multiprocessor has its own dedicated shared memory (cache) that is manually managed by the executing thread block. By itself, this greatly simplifies the requirements of our model. Accesses to GPU RAM are still variable based on contention, however due to the static scheduling and regular execution patterns of the parallel threads, the access times are much more repeatable across runs of the application. These factors are also accounted for in Section 4.2.

A major contribution of our work is an accurate model that determines access time to disk, though we are again aided by the GPU environment. In order for an application to run on a GPU, all of its data must be transferred to the GPU prior to execution, and from the GPU after execution completes. Since we must package our data into continuous (streaming) blocks for execution on the GPU, paging to disk is much more regular and predictable. The details of this model are described in Section 4.4.

Network Communication. It is common for network models to include cycle accurate counts for the preparation of data by the sender, the actual transmission, and the processing of data by the receiver. However, they do not usually consider contention in the network, irregular overhead by the operating system, work done by the process during communication, and a myriad of other factors [8].

As with our disk model, we have the advantage here that network requests coming from the GPU must packaged together, and are transmitted as a group before or after kernel execution. Again the GPU environment presents us with a very regular

communication pattern in which we can easily determine the points in the algorithm that the various nodes of the system will communicate and the total amount of data. For this work, we assume that the network topology is known, and we can thus accurately predict network transmission time without having cycle-specific estimates.

7.2 Limitations of Scalability Equations

The models presented in this work rely on applications in which each thread has a regular, predictable execution flow. Applications where threads may exit early (such as in iterative algorithms) or where communication patterns are unpredictable (such as particle interactions) cannot be successfully modeled because of the unknown effects on the system. Memory access patterns and associated latency, and communication requirements are just some of the manifestations of unpredictability.

Also, when attempting to model the scalability of GPU execution in this work, we make the assumption that GPU occupancy remains high; that is, the number of blocks and threads available to each GPU is large enough so that performance degradation is not seen as successive GPUs are added. This is not always the case, and some of the applications presented would show degradation if we were able to experiment with larger numbers of GPUs.

Ongoing work is focused on taking into account the minimal thread configuration for a given application and data set size that will allow scaling to occur without a corresponding degradation in execution (described by the Isoefficiency metric [14]). We are also focusing on accurately predicting the degradation that will occur when occupancy drops beyond this threshold.

7.3 Obtaining Repeatable Results

The predictions presented here were tested on systems running an unmodified version of Fedora 7, but a lot of effort was put into quelling operating system "jitter". Jitter refers to the irregularities in measurements or execution times caused by the operating system, user or system processes, or otherwise uncontrollable events that may skew the accuracy of timing [25].

Some examples of uncontrollable factors that affect the accuracy of our models are listed below:

- All processes affect the amount of system memory available, which can cause our applications to begin to page data to disk earlier than expected.
- Daemon or other processes that run during testing require process switches and may access I/O devices (such as the network) which can cause contention for resources.
- If our applications have completely filled system RAM, a context switch (to system or other user processes) requires data to be paged, potentially having drastic effects on the measured execution time.

The following steps were taken in order to reduce jitter when running tests on our system:

- The X server was disabled.
- Unnecessary user-space and system processes were terminated.
- All workstations were connected to a private network switch, with no connection to outside networks.

For our purposes, jitter has the most noticeable effects on applications that either have very short or very long execution time. Very short executions are more susceptible to inaccurate measurements due to scheduling or time-sharing on the CPU. The longest running processes are usually those that completely fill system RAM and require paging to disk. During these executions, there is a large probability that other processes (such as daemons) will run and require their memory pages to be paged in from disk—affecting the duration of their execution time and increasing the amount of data that our application must page in as well (exceeding the worst-case scenario that our model predicts).

Chapter 8

Conclusion and Future Work

In this work, we presented a novel methodology for accurately predicting the execution time of GPU applications as the number of GPUs in the system and/or the data set size is scaled. In order to achieve accurate results, we created models for representing all aspects of GPU computing, including the scaling of computation on the GPU, GPU-CPU interaction, and CPU-disk interaction. The CPU-disk model is extremely important because it is the least deterministic and an inaccurate model will have a huge impact on performance prediction due to the limited disk-to-CPU bandwidth.

When we applied our prediction methodology to real-world applications, we found that we had average difference of only 11% between predicted and actual execution time. Equally important is that our worst prediction difference was off by only 40%, and occurred because of optimizations to the Linux memory manager that represent a corner-case in our simplified disk paging model. The specific case where our largest difference occurred was when the data allocated was slightly larger than RAM. Our model predicted that some percentage of the data would need to be paged in from disk for each of many iterations of the algorithm. However, Linux was likely able to

predict that the entire block of data would be needed and did not page out the amount that we predicted it would in a strict LRU scheme. Regardless, due to the regular execution profile of most GPU applications, the static scheduling of the applications on the GPU, and the consistency of communication transfers, we were quite accurate in the vast majority of cases.

8.1 Summary of Contributions

The practical importance of this work is that it shows that researchers, programmers, or administrators can predict performance on a vast number of systems and data sets and choose a GPU-based hardware solution that is in line with their cost-performance needs.

The contributions of this work are thus as follows:

- At a system level, we have identified the significant factors related to computation and communication of applications in multi-GPU environments.

- For each significant factor, we have created models that allow a developer to predict the actual amount of time that will be spent by the factor as the number of GPUs and data sets are scaled and as the GPU configuration changes.

- Using the system-level models, we then show how to predict execution and generate plots that allow a developer to observe how the application will scale across various numbers and configurations of GPUs, and with any input data size.

- Finally, we validated our models and methodology on six scientific-based high performance computing applications and showed that the actual execution time

is modeled accurately by our predictions.

8.2 Future Work

As the scope of modeling execution is very broad, there are many opportunities to improve or expand upon this work. The most obvious place for improvement is highlighted in Section 7.2. While our methodology allows a developer to scale their program to an unbounded number of GPUs, we currently do not consider the effects of underutilization on performance.

In addition to improving the models used for performance prediction, our validation environment currently only covers a small domain space. To verify our predictions in distributed environments, we validate on up to 4 workstations connected via a network, and validate on only 2 GPUs in a shared system environment. In order to conclusively show the accuracy of our models, we need to increase the size of our coverage in both areas.

Finally, though interaction with NVIDIA may be required, creating a more sound model of the overhead of the CUDA driver for increasing numbers of GPUs would greatly improve the correctness of the PCI-e model. The current problem we have is that the CUDA driver is closed source, so there is no way to gain insight into the overhead of sharing the driver or bus with multiple GPUs. Since the goal of this work is to create accurate models without requiring a developer to purchase multiple GPUs ahead of time, a model that can predict PCI-e scalability accurately is key.

Bibliography

[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] Ron Babich. Riding the GPU bandwagon with CUDA. `http://super.bu.edu/~brower/scidacFNAL2008/slides/scidac08_babich.pdf`.

[3] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[4] Robert J. Block, Sekhar Sarukkai, and Pankaj Mehra. Automated performance prediction of message-passing parallel programs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, New York, NY, USA, 1995. ACM.

[5] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[6] 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.

[7] Mark J. Clement and Michael J. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23:1405–1420, 1997.

[8] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

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

[10] Mike Fagan. Fox matrix multiply algorithm. `http://www.caam.rice.edu/~caam520/Topics/ParallelAlgorithms/LinearAlgebra/fox.html`.

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

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

[13] 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.

[14] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, 1993.

[15] Mark J. Harris. GPGPU: Beyond graphics. *Eurographics Tutorial*, 2004.

[16] Justin Hensley and Jason Yang. Tuning GPGPU applications for performance. In *SIGGRAPH '07: ACM SIGGRAPH 2007*, 2007.

[17] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Mike Houston. General purpose computing on graphics processors (GPGPU). `http://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf`.

[19] Mike Houston. Understanding GPUs through benchmarking. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 8. ACM, 2007.

[20] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-aware optimization targeting multithreaded stream computing. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 62–70, New York, NY, USA, 2009. ACM.

[21] Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli. Architecture-aware optimization targeting multithreaded stream computing. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 62–70, New York, NY, USA, 2009. ACM.

[22] Richard Johnson. GPU computing with CUDA. `http://cscads.rice.edu/workshops/july2007/autotune-slides-07/RichardJohson.pdf`.

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

[24] Dave Luebke, Michael Garland, John Owens, and Kevin Skadron. GPGPU: ASPLOS 2008 CUDA TUTORIAL. http://www.gpgpu.org/asplos2008/.

[25] Pradipta De Vijay Mann and Umang Mittaly. Handling os jitter on multicore multithreaded systems. *Parallel and Distributed Processing Symposium, International*, 0:1–12, 2009.

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

[27] B. Scott Michel. GPGPU computing and multicore processors: Exploring the spectrum. `http://sunset.usc.edu/gsaw/gsaw2007/evening/michel.pdf`.

[28] Perhaad Mistry, Sherman Braganza, David Kaeli, and Miriam Leeser. Accelerating phase unwrapping and affine transformations for optical quadrature microscopy using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 28–37, New York, NY, USA, 2009. ACM.

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

[30] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008*, 2008.

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

[32] 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.

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

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

[35] 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.

[36] 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.

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

[38] 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.

[39] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen 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.

[40] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

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

[42] 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.

[43] M. Strengert, C Mller, C. Dachsbacher, and T. Ertl. CUDASA: Compute unified device and systems architecture. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV08)*, pages 49–56. Eurographics Association, 2008.

[44] Jeff A Stuart and John D. Owens. Message passing on data-parallel architectures. *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[45] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.

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

[47] 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.

[48] Bixia Zheng, Derek Gladding, and Micah Villmow. Building a high level language compiler for GPGPU. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, 2008.