Generalizing the Utility of Graphics Processing Units in Large-Scale Heterogeneous Computing Systems

Shucai Xiao

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Wu-chun Feng, Chair
Peter Athanas
Pavan Balaji
Yong Cao
Cameron Patterson
Sandeep K. Shukla

March 12th 2012
Blacksburg, Virginia

Keywords: Graphics Processing Unit (GPU), CUDA, OpenCL, BLAST, Smith-Waterman, Fine-Grained Parallelization, GPU Virtualization, GPU Synchronization, Task Migration

Generalizing the Utility of Graphics Processing Units in Large-Scale Heterogeneous Computing Systems

Shucai Xiao

ABSTRACT

Today, heterogeneous computing systems are widely used to meet the increasing demand for high-performance computing. These systems commonly use powerful and energy-efficient accelerators to augment general-purpose processors (i.e., CPUs). The graphic processing unit (GPU) is one such accelerator. Originally designed solely for graphics processing, GPUs have evolved into programmable processors that can deliver massive parallel processing power for general-purpose applications.

Using SIMD (Single Instruction Multiple Data) based components as building units, the current GPU architecture is well suited for data-parallel applications where the execution of each task is independent. With the delivery of programming models such as Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), programming GPUs has become much easier than before. However, developing and optimizing an application on a GPU is still a challenging task, even for well-trained computing experts. Such programming tasks will be even more challenging in large-scale heterogeneous systems, particularly in the context of *utility computing*, where GPU resources are used as a service. These challenges are largely due to the limitations in the current programming models: (1) there are no intra- and inter-GPU cooperative mechanisms that are natively supported; (2) current programming models only support the utilization of GPUs installed locally; and (3) to use GPUs on another node, application programs need to explicitly call application programming interface (API) functions for data communication.

To reduce the mapping efforts and to better utilize the GPU resources, we investigate

generalizing the utility of GPUs in large-scale heterogeneous systems with GPUs as accelerators. We generalize the utility of GPUs through the *transparent virtualization of GPUs*, which can enable applications to view all GPUs in the system as if they were installed locally. As a result, all GPUs in the system can be used as local GPUs. Moreover, GPU virtualization is a key capability to support the notion of "GPU as a service." Specifically, we propose the *virtual OpenCL* (or *VOCL*) framework for the transparent virtualization of GPUs. To achieve good performance, we optimize and extend the framework in three aspects: (1) optimize VOCL by reducing the data transfer overhead between the local node and remote node; (2) propose GPU synchronization to reduce the overhead of switching back and forth if multiple kernel launches are needed for data communication across different compute units on a GPU; and (3) extend VOCL to support live virtual GPU migration for quick system maintenance and load rebalancing across GPUs.

With the above optimizations and extensions, we thoroughly evaluate VOCL along three dimensions: (1) show the performance improvement for each of our optimization strategies; (2) evaluate the overhead of using remote GPUs via several microbenchmark suites as well as a few real-world applications; and (3) demonstrate the overhead as well as the benefit of live virtual GPU migration. Our experimental results indicate that VOCL can generalize the utility of GPUs in large-scale systems at a reasonable virtualization and migration cost.

Acknowledgments

I would like to acknowledge many people who helped me during my Ph.D. study at Virginia Tech. First and foremost, I would like to thank my Ph.D. advisor, Dr. Wu-chun Feng, for his generous support, excellent guidance, and rigorous scholarship. He not only taught me the research and presentation skills, but more importantly, inspired me to develop the independent thinking capability that will continuously benefit me in my career. I will never forget the times and endeavors that he has invested to make my graduate study productive. I would also like to thank Dr. Heshan Lin for his generous and patient guidance and suggestions in my PhD study. During my internship at NVIDIA Corporation and Argonne National Laboratory, Dr. Peng Wang and Dr. Pavan Balaji taught me a lot of hands-on skills and gave me very useful suggestions for my research, respectively. I really appreciate their effort and help.

I feel so fortunate that I have an excellent doctoral advisory committee, and I would like to thank Dr. Peter Athanas, Dr. Pavan Balaji, Dr. Yong Cao, Dr. Cameron Patterson, and Dr. Sandeep K. Shukla for their support, advice, and encouragement. During my graduate study at Virginia Tech, I have worked closely with Ashwin M. Aji, Jeremy Archuleta, Dr. James Dinan, Song Huang, Yang Jiao, Thomas Scogland, and Dr. Qian Zhu. Many of my publications would have been impossible without their help. The members of the Synergy Laboratory have contributed immensely to my research work and personal life at

Virginia Tech. I am especially grateful for the group members: Jacqueline Addesa, Ashwin M. Aji, Ryan Braithwaite, Mayank Daga, Michella Datoc, Marwa Elteir, Dr. Mark K. Gardner, Chris Goddard, Yang Jiao, Umar Kalim, Konstantinos Krommydas, Kenneth Lee, Dr. Heshan Lin, Gabriel Martinez, Ganesh Narayanaswamy, Lee Nau, Rishi K. Prasad, Paul Sathre, Thomas Scogland, Ajeet Singh, Balaji Subramaniam, Sahil Talwar. Benjamin Wang, and Jing Zhang.

I must thank all my friends at Blacksburg, because my life here would be less wonderful without them. Dr. Kaigui Bian, Dr. Ruiliang Chen, Dr. Chao Huang, Guanghong Pei, Dr. Ende Pan, Dr. Jiang Wang, Dr. Jingwei Zhang, Dr. Yexin Zheng, and Dr. Jian Zuo, have offered me a lot of help starting from the first year of my life at Blacksburg. Later, I found more and more lovely friends here: Xuetao Chen, Zhimin Chen, Yipan Deng, Xu Guo, Chuan Han, Canming Jiang, Dr. Dong Li, Min Li (CS), Min Li (ECE), Hua Lin, Chun-yi Su, Xiaokui Su, Guanying Wang, Ting Wang, Liguang Xie, Huijun Xiong, Kui Xu, Bin Xue, Dr. Yanzhu Ye, and Hao Zhang. It is too long a list to cover all of them.

Lastly, but most importantly, my family has always been a source of support and encouragement for me. I am deeply indebted to my parents for instilling me into being honest, persevering, and hardworking. I would also like to express my deepest gratitude to my wife Xiaoman. Her support and love smooth the way to my Ph.D. Finally, I am thankful for the arrival of my daughter Meiheng. Her smile gives me additional encouragement in the final stage of my study here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, the growth rate of computational power required for scientific computations has been outstripping that of the computational capabilities of traditional processors in some areas. One example is searching biological sequence databases. In early 2000, researchers found that searching a popular public sequence database slowed by 64% each year [19]. With the appearance of the next-generation sequence technologies and emergent search areas in the life sciences, such as meta-genomics, the generation of sequence data is increasing at an unprecedented rate, thus further slowing down genomic database search.

Another example is found in cosmology. One of the common computations is to compute the two-point angular correlation function (TPACF), which can be very time-consuming as it requires computing auto- and cross-correlations between hundreds of thousands to millions of galaxies [68]. With the information of more galaxies observed, such computation will take even more time than ever before.

A third example is the hydrological sciences, which have taken advantage of high-performance computing in the areas of atmospheric and ocean sciences due to their dependence on fine spatial grids and small time steps for integration [43]. With the spatial grids and the temporal resolutions becoming even finer, computers with far greater computational horsepower are necessary.

On the other hand, gains in computational horsepower by increasing the clock speed have reached a wall due to the fabrication process and power/leakage constraints, as shown in Figure 1.1. As such, parallelisms have been widely utilized to achieve computational power gains. For parallelism, in the low level, performance improvement is driven by increasing the number of cores in both the traditional X86 multi-core processors and the newly appeared many-core processors. In a high level, compute nodes are installed with multiple processors and supercomputers are composed of hundreds or thousands of compute nodes, in which accelerators such as graphics processing units (GPUs) are used. In fact, GPUs are used as accelerators in three of the top five supercomputers in a recent Top500 list [4].

Originally designed for accelerating the rendering of images in a frame buffer for output to a display, which is essentially a data-parallel and compute-intensive task, GPUs have evolved into programmable processors useful for general-purpose tasks (referred to as general-purpose computation on the GPU (GPGPU)). Following the streaming architecture of the J-Machine [10, 22] and Merrimac [23], which has an order of magnitude more performance per unit cost than contemporary cluster-based scientific computers, GPUs deliver much higher peak performance than traditional processors, and the performance

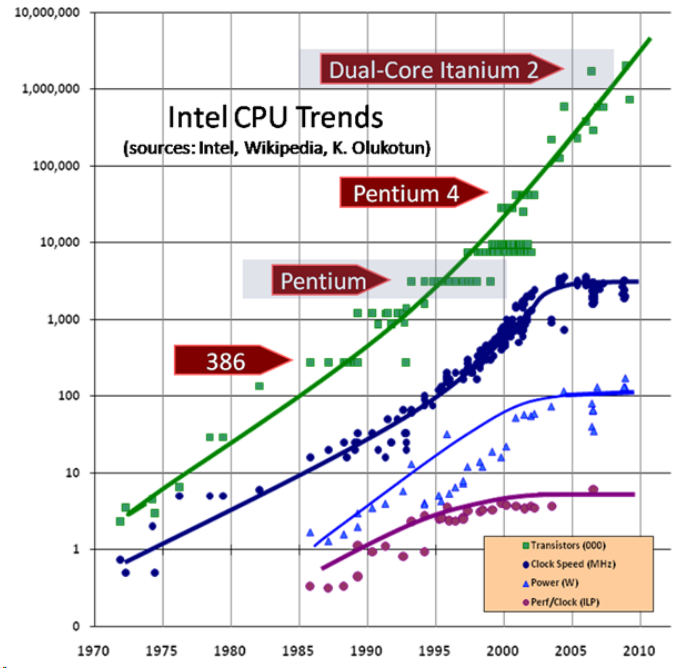


Figure 1.1: Intel CPU Introductions [77]

gap between them becomes even larger as time goes on [63]. With emerging parallel programming models (e.g., NVIDIA's Compute Unified Device Architecture (CUDA) [63], AMD/ATI's Brook+ [9], and OpenCL [42]), programming GPUs has become much easier than before though it is still a low-level and laborious process. As such, many applications, including dense linear algebra [80, 82], fast Fourier transformation (FFT) [32, 83], sequence alignment [54, 57, 75, 87], sort [31, 33, 38], and shortest-path search [41] have been parallelized on GPUs, and performance improvement has been reported [64].

However, parallelizing applications to achieve good performance on GPUs is *not* straightforward and sometimes requires significant programming effort, even for well-trained

computing experts. This problem becomes even worse when using computational resources in large-scale heterogeneous systems. For instance, on a single GPU card, programmers need to carefully design the access pattern to device memory to improve memory access throughput. Moreover, there is no explicit support for data communication across different single-instruction, multiple-data (SIMD) compute units on the GPU. When the execution of an application needs such data communication, we need to use an implicit approach, which requires the program execution to switch back and forth between the host processor (i.e., CPU) and the device (i.e., GPU) and can cause a large overhead to program execution.

In large-scale systems with GPUs as accelerators, besides the challenges on a single GPU card as mentioned above, the use of GPUs across the whole system is another challenge. Specifically, current GPU programming models only support the utilization of GPUs installed locally. When using GPUs in other compute nodes, application programs need to call TCP sockets and/or Message Passing Interface (MPI) [56] API functions explicitly for data communication across different compute nodes. With the delivery of more supercomputers using GPUs as accelerators, this problem will become a common scenario. Thus generalizing the utilization of GPUs, i.e., using all GPUs in the same way as local GPUs in large-scale systems, becomes more important than ever before.

To resolve this problem, the work in this dissertation investigates the transparent virtualization of GPUs. Transparent virtualization of GPUs is very important in the utility of GPUs in many aspects. First, it enables all GPUs in the system to be used in the same way as if they were installed locally. As such, programmers can focus on the parallelization of GPUs with no need to consider data communication across different machines. Second,

virtualization has the potential of increasing the usability and reliability of GPUs across the system to maximize system flexibility. As such, any GPU in the system can be used by any program. To this end, even if GPUs are installed in only part of the compute nodes, programs running in the system can still take advantage of GPUs no matter whether GPUs are configured on a node. With only a portion of the compute nodes installed with GPUs, we can save the system hardware investment and power consumption. For example, the supercomputer in the Blue Waters project [59] has only 10% of its compute nodes installed with GPUs. Third, virtualization is a key capability to support the notion of *utility computing* [11]. With the ability to support GPU virtualization, end users can use GPU resources in large-scale systems in the context of "GPU as a service." In this way, end users need to consider neither the initial hardware investment nor the system maintenance, or even how their tasks are actually computed. They just submit their jobs, wait for the final results, and pay for the resources that they have used. From the perspective of the service provider, they can coordinate the jobs of many customers and use the computational resources more efficiently. In addition, virtualization eases the system maintenance by allowing more nodes to be added to the system without stopping the system. As such, system maintenance and upgrading can be performed smoothly.

The VOCL framework provides user-level virtualization of GPU devices in large-scale heterogeneous computing systems. Though it does not handle security issues, it does provide similar usage and management benefits such as transparent utilization of remote GPUs and task migration. In addition, various optimization techniques are proposed to reduce the overhead in the virtualization and task migration. Overall, this dissertation addresses the following areas: (1) GPU virtualization, (2) GPU synchronization, (3) task

migration, and (4) application case studies.

1.1 GPU Virtualization

GPUs are widely used in today's supercomputers as accelerators for general-purpose computation. In the meantime, the development of virtualization technologies has been advocating the virtualization of GPUs to view them in the same way no matter whether they are installed locally or remotely. However, the current GPU programming models can only support the utilization of GPUs installed locally. For GPUs on remote machines, a program needs to explicitly call API functions for data communication between different compute nodes. As a result, additional programming effort is needed for the application parallelization. To resolve this problem, we propose the *Virtual OpenCL* (or *VOCL*) for the virtualization of GPUs based on the OpenCL programming model. VOCL enables all accelerators installed in the system to be used in the same way as if they were installed locally. With the VOCL framework, an application can use all the GPUs in the system to accelerate its execution if they are available. Conversely, a GPU can be used by any program running in the system. Overall, VOCL manages computational resources in a more efficient and flexible way.

Since current GPUs use a separate memory address space from the host processor, inputs should be transferred to the GPU memory before GPU computations can be performed. After the computation is completed, results should be copied back to the host memory for program output. In GPU computing, overhead caused by the data transfer between host memory and device memory can slow down the program execution by up to 50-fold in some applications [34]. When remote GPUs are used, it is expected that even

larger overhead will be caused by the data transfer. As such, reducing the data transfer overhead is of great importance in the utilization of remote GPUs. To reduce the data transfer overhead in VOCL when remote GPUs are used, we propose two optimization strategies—kernel-argument caching and data-transfer pipelining. With kernel-argument caching, our experimental results show that the overhead caused by setting kernel arguments can be reduced by about 100 times. With data-transfer pipelining and using multiple threads to handle transfer in the remote node, we can achieve 80% - 90% of the bandwidth in the native OpenCL environment. We also evaluate the virtualization overhead via several real-world application kernels—matrix multiplication, matrix transpose, n-body, and Smith-Waterman. Our experimental results indicate that VOCL can support the transparent virtualization of GPUs at a reasonable virtualization cost, particularly for compute-intensive applications.

1.2 GPU Synchronization

When parallelized on the GPU, some applications need data communication across different SIMD compute units of the GPU during their execution. On the GPU card, data communication occurs via the global memory and then requires barrier synchronization across compute units to ensure correct data communication. Currently, there is no explicit support for the synchronization, and such synchronization is only implicitly available via the CPU, which requires the program execution to switch back and forth between the host (i.e., CPU) and the device (i.e., GPU), causing significant overhead for program execution. The overhead can become even larger in the utilization of remote GPUs. When remote GPUs are used, the switching back and forth between the host and the device becomes a

two-phase procedure. Specifically, each kernel launch needs to be sent from the local node to the remote node, and then the kernel is launched on the remote node. To reduce the synchronization overhead, we propose two *GPU synchronization* approaches [29, 88] for the data communication across different compute units. These approaches can synchronize the execution on different compute units without the CPU involved, thus eliminating the overhead of switching back and forth between the host and the device, particularly in the scenario of using remote GPUs. From our experimental results, the synchronization time can be reduced to 1/4 of that of the implicit barrier synchronization approach when our proposed GPU synchronization is used.

1.3 Task Migration

Task migration is the act of transferring tasks across different nodes. It is to provide for an enhanced degree of dynamic load distribution, fault resilience, eased system administration, and data access locality [81].

With the VOCL framework, an application program can view and use all the OpenCL-enabled accelerators (e.g., GPUs) in a system. At the same time, each device can be used by multiple programs concurrently. When multiple programs are running in a system, load imbalance can exist across different devices, and application performance will be affected. Since today's supercomputers consist of large numbers of compute nodes, if task migration is available across different nodes, we can balance loads by migrating some tasks from GPUs with heavy load to those with light load.

Moreover, system maintenance or upgrading are frequent occurrences in today's supercomputers. Without task migration, we should wait for the completion of the running

tasks before the actual maintenance or upgrading can happen. However, if live task migration is supported, we can just move all the tasks on the node to other nodes, and then we can perform the maintenance immediately.

In this work, we extend the VOCL framework to support *live task migration* across different GPUs. Our migration strategy is *transparent* to the application program. It is based on the mapping of *virtual GPUs*, consisting of the OpenCL objects used by an application, on physical GPUs. Migration is then accomplished by mapping a virtual GPU from one physical GPU to another, and tasks corresponding to the virtual GPU will automatically move from the (physical) source GPU to the (physical) destination GPU. In addition, to reduce the migration overhead, we enhance the VOCL framework in two aspects—*atomic transaction of OpenCL function calls* and *internal queuing of OpenCL function calls in the proxy*.

1.4 Application Case Studies

As case studies, we parallelize two bioinformatics applications for sequence alignment on the GPU. Sequence alignment finds similar segments or local alignments between different protein or nucleotide sequences. As mentioned above, with the arrival of next generation of sequencing technologies, genomic sequence data will be generated at an unprecedented rate, which makes more and more computational power necessary to process them. Since GPUs deliver much higher peak performance than traditional processors, using GPUs for sequence alignment is a promising approach. For sequence alignment, there are two types of approaches — Basic Local Alignment Search Tool (BLAST), which is heuristic-based,

and Smith-Waterman, which is an optimal local alignment algorithm that is based on dynamic programming. Execution of BLAST is fast, but it sometimes reports sub-optimal alignments. In contrast, Smith-Waterman can compute locally optimal alignments, but it is slow.

In this work, we use the *coarse-grained* approach to parallelize BLAST on a GPU. In this approach, each thread is responsible for the alignment of one pair of sequences. By parallelizing them on the GPU, we achieve performance improvement over executing on traditional CPUs. As for Smith-Waterman, we parallelize it with a fine-grained approach, which uses all resources on a GPU for the alignment of a single pair of sequences. As such, our fine-grained parallel implementations can align sequences of up to 8K letters in size on an NVIDIA GTX 280 GPU card, which effectively covers the alignment of all protein sequences in the NCBI GenBank.

With the parallel implementations of BLAST and Smith-Waterman on the GPU, we verify the feasibility of using remote GPUs and the performance improvement delivered by GPU synchronization, respectively. Specifically, for BLAST, we use multiple GPUs (both local and remote) to search a specific database and show the performance improvement. As for Smith-Waterman, we show the performance improvement brought by the GPU synchronization in both the local and remote GPU scenarios.

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 introduces background information, which includes the GPU architecture and its associated CUDA and OpenCL

programming models, Message Passing Interface (MPI), and general rules for programming optimization on the GPU. In Chapter 3, we present the VOCL framework for the transparent virtualization of GPUs as well as some optimizations. These optimizations reduce the data transfer overhead involved in the use of remote GPUs and are integrated into the VOCL framework. Chapter 4 proposes the GPU synchronization strategies for efficient data communication across different compute units on the GPU card. GPU synchronization is applied to application programs and is particularly useful in reducing the overhead caused by multiple kernel launches when remote GPUs are used. Chapter 5 is the extension of the VOCL framework by supporting live task migration, which can load balance across different GPUs and enable uninterrupted system maintenance in large-scale systems. Chapter 6 briefly describes the parallelization of two bioinformatics applications — BLAST and Smith-Waterman on the GPU, and we use them as examples to verify our generalization of GPU usage. Finally, Chapter 7 presents the conclusion and future work.

Chapter 2

Background

In this chapter, we provide background knowledge on the following topics: (1) NVIDIA GPU architecture, (2) programming models on the GPU, (3) MPI (Message Passing Interface), and (4) general rules for programming optimization.

2.1 NVIDIA Graphics Processing Unit Architecture

A GPU is a special processor designed specifically for accelerating certain parts of graphics tasks. With the vertex and fragment shaders added to the graphics pipeline, it is possible for programmers to replace some default operations provided by graphics cards to support additional operations. In this way, general-purpose computations can be performed. In general, an NVIDIA GPU consists of a set of SIMD streaming multiprocessors (SMs), as shown in Figure 2.1. Each SM consists of a few scalar processing (SP) cores, one or more instructional units, and one or more special functional units. There are four types of on-chip memory on each SM: registers, shared memory, a read-only constant cache, and

a read-only texture cache. On-chip memory can only be accessed by threads on that SM. In addition, off-chip memory is provided, which includes global memory and local memory. Global memory can be accessed by threads across the whole GPU card, and it can be used for data communication across different SMs [63]. Local memory is thread-based and used only when the register file is full. It is worth noting that local memory is *not local* to threads. It is off-chip instead. Comparing on-chip memory and off-chip memory, on-chip memory has a low access latency, but its size is small, tens of kilobytes (KB) in general. In contrast, off-chip memory is large, but its access latency is much higher, approximately 100 times slower than on-chip memory. One way to mitigate the high latency is to access global memory in a way that multiple data transactions can be combined in to a single data transaction, which is referred to as *coalesced memory access*. In addition, later generations of *Fermi* GPU architectures, e.g., Tesla C2050, provides L1 and L2 caches, which can help to improve the efficacy of global memory access, particularly when memory accesses are irregular. Different generations of GPUs have different memory sizes and different numbers of processing cores on each card. For example, the Tesla C1060 GPU contains 4GB global memory, 16KB shared memory and 16K 32-bit registers on each SM, and 240 processing cores; while on the Tesla C2050, there are 3GB global memory, 64KB shared memory (configurable as either 16KB L1 cache/48KB shared memory or 48KB L1 cache/16KB shared memory) and 32K 32-bit registers on each SM, and 448 processing cores.

In general, execution of a typical program on a GPU includes the following steps. The program first allocates buffers in the GPU memory. Then it copies kernel inputs to the allocated GPU memory and performs computation on the GPU. After GPU computation is

completed, the program copies the kernel output back to the host memory for program output. Finally, the program releases the buffers in the GPU memory. Compared to program execution on traditional CPUs, GPU programs need data transfers between host memory on the CPU and device memory on the GPU. As for the data transfer time, it ranges from being negligible in compute-intensive programs to significant in data movement-intensive programs, since the data movement can occupy a large portion of the total program execution time [34].

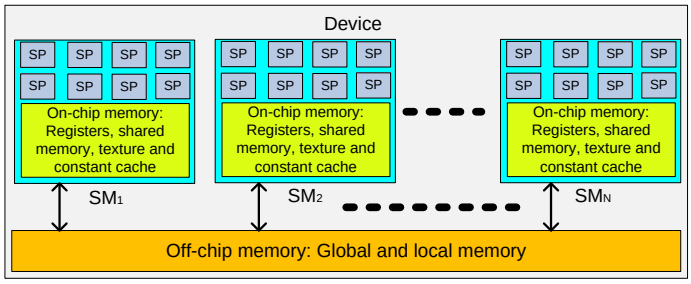


Figure 2.1: An Overview of GPU Architecture.

2.2 Programming Models on GPUs

To support general-purpose computation on their GPUs, NVIDIA provides the CUDA and OpenCL programming models to reduce programming effort. Below we give a brief description for each of them.

2.2.1 Compute Unified Device Architecture

CUDA is the parallel programming model [63] provided by NVIDIA to run programs on their GPU cards. It is an extension of the C programming language. When a program is

mapped, in general, the computation-intensive and/or data-parallel parts are parallelized to take advantage of the GPU computational horsepower. These parts are implemented as *kernels*, which are called on the host and executed on the device. In a kernel, threads are grouped as a grid of thread blocks, and each thread block contains a number of threads. Multiple blocks can be executed on the same SM, but any given block can be executed on only one SM. In addition, threads within a block are organized as groups of 32 threads, i.e. *warps*, in which the same instructions are executed across all threads if there is no divergent branching. If divergence occurs, however, instructions are executed sequentially across different execution paths within the warp, as shown in Figure 2.2.

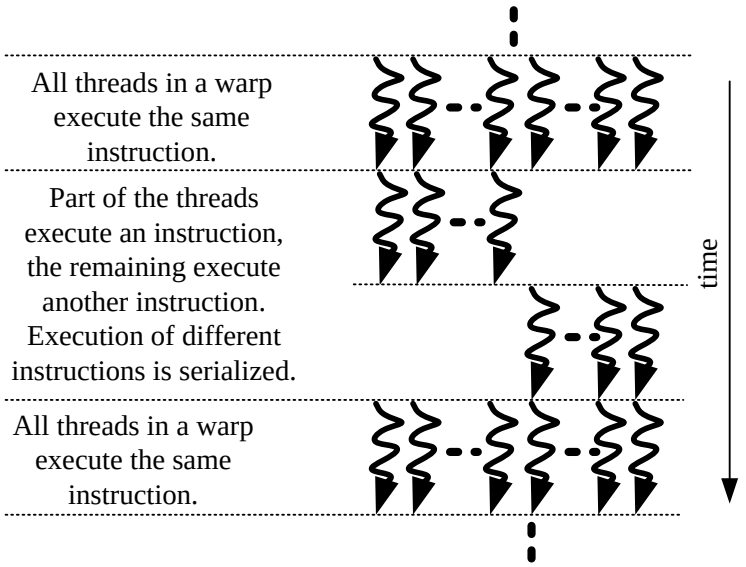


Figure 2.2: Execution of Divergent Branches in a Warp

CUDA provides a data communication mechanism for threads within a block, which can be implemented via the shared memory with the barrier function `__syncthreads()`. In addition, atomic functions are provided for some kind of data communication across

different threads. However, there is no explicit software or hardware support for data communication across different blocks, i.e. *inter-block data communication*. Currently, inter-block data communication occurs via the global memory and the needed barriers are implemented *implicitly* by "expensive" interaction with the CPU—terminating current kernel's execution, returning control to the CPU, and re-launching the kernel from the CPU.

NVIDIA GPUs follow a relaxed memory consistency model. According to the CUDA programming guide, memory consistency is guaranteed only in the following scenarios:

- Writes to the shared or global memory are visible to later reads in the same thread.
- The barrier function `__syncthreads()` can guarantee that writes to the shared and global memory are visible to threads within the same block.
- The memory fence function `__threadfence()` blocks execution of the calling thread until its writes to the shared memory and global memory are visible to all the threads in a kernel.
- Terminating a kernel ensures that the writes to the global memory are visible to all threads in the next kernel.

The current CUDA programming model only supports the utilization of NVIDIA GPUs installed locally. If a CUDA program needs to use a remote GPU, data communication mechanisms such as TCP sockets and MPI [56] are needed for the data communication between different compute nodes.

2.2.2 Open Computing Language

The Open Computing Language (OpenCL) provides another approach of programming accelerators such as CPUs and GPUs in heterogeneous computing environments. It includes a C-based programming language to write *kernels* that are executed on the aforementioned accelerators. Moreover, it provides APIs to define and control the programming context for a specific platform. In OpenCL, kernel execution is performed within a host-defined context and GPU computation is based on the following OpenCL objects — GPU devices, program objects, memory objects, kernels, command queues, etc. The dependency between the OpenCL objects is shown in Figure 2.3.

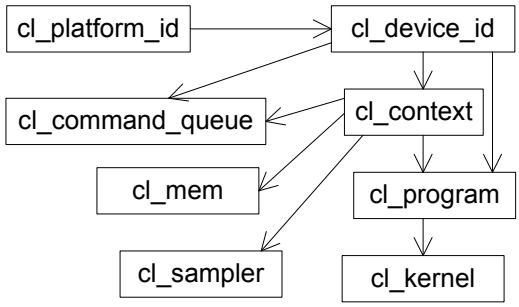


Figure 2.3: Dependency across OpenCL objects.

As with CUDA, compute-intensive and data-parallel components in applications are parallelized as kernels running on accelerators, and each kernel in OpenCL contains a few *work-groups* with each work-group containing a few *work-items*. Likewise, the OpenCL programming model can only support the utilization of accelerators installed locally. However, OpenCL is more widely supported by processors such as NVIDIA GPUs, AMD

GPUs, Cell Broadband Engine (Cell/BE) [21], and traditional multi-core processors. Moreover, in contrast to CUDA, which is based on the CUDA compiler, OpenCL is based on the OpenCL library. In this way, we can easily overload the OpenCL API functions without the need to design a new compiler. Finally, initialization of OpenCL is a little more complex compared to the CUDA, or more specifically, the CUDA runtime API. In OpenCL, API functions need to be called explicitly for the initialization; while with the CUDA runtime API, all initializations are performed implicitly by the CUDA library [55].

2.3 Message Passing Interface

The Message Passing Interface (MPI) [56] is a standard library API for writing message-passing programs in FORTRAN, C, or C++. It is widely supported by all vendors, and portable, open-source MPI implementations (MPICH2 [1] and Open MPI [2]) also exist. MPI provides rich functionality for communication, including basic point-to-point sends and receives, collective communication, and asynchronous, one-sided communication. MPI also provides the ability to dynamically establish and destroy new communication channels between different MPI processes. For these reasons, we have selected MPI as the communication runtime system on which to construct VOCL. While the VOCL runtime utilizes MPI internally, the application only needs to interface with OpenCL to take advantage of VOCL.

2.4 General Rules for Programming Optimization

According to the CUDA programming guide, programming optimizations on GPUs are mainly performed in three aspects—resource utilization, memory throughput, and instruction throughput. In the following, each of them is described.

2.4.1 Resource Utilization

In a heterogeneous computing environment with GPUs as co-processors, there are computational resources on both CPUs and GPUs. In general, CPUs are better as low-latency processors that support multiple-instruction, multiple-data (MIMD) operations; while GPUs are better as high-throughput processors that support SIMD operations. Thus, when scheduling tasks in such an environment, we should assign the sequential portions of a program to CPUs and data- and/or task-parallel portions to GPUs.

In addition, kernel launch on the GPU is an asynchronous operation. When a kernel is launched, program execution returns to the CPU before the computation on the GPU is completed. At this time, if we assign some portions of the program to the CPU, then computation can be performed on both the CPU and the GPU simultaneously, and computational resources are better utilized.

As for the resource usage on a GPU card, we should improve the utilization on both one SM and the whole GPU card. For the former, the *SM occupancy* is used as a measurement, which is the ratio of the number of active warps on an SM to the maximum number of active warps on an SM (e.g., 32 for GPUs with compute capability 1.x and 48 for GPUs with compute capability 2.x). SM occupancy depends on the kernel resource usage, which

includes the number of registers used per thread and the shared memory used per block, etc. The kernel configuration, i.e., the number of threads per block and the number of blocks in the kernel, can also affect the SM occupancy. SM occupancy can be calculated by the CUDA occupancy calculator [62]. On the whole GPU card, enough blocks should be configured to make computation performed on all the SMs.

2.4.2 Memory Throughput

Maximizing memory throughput can be achieved by minimizing data transfer between host memory and device memory since memory throughput between device memories is much larger than that between host memory and device memory. Moreover, on-chip memory on the GPU has much higher throughput than off-chip memory. Thus, the on-chip memory should be used as much as possible. Basically, the following procedure are followed to perform computation to take advantage of the on-chip memory and achieve high memory throughput:

- Load data from the off-chip memory to the on-chip memory.
- Perform computation on the on-chip memory.
- After computation is done, store the result back to the off-chip memory.

With the above procedure, accesses to the off-chip memory can be minimized and the overall memory throughput can be improved.

There are texture and constant caches on all GPU cards. Moreover, L1 and L2 caches are provided on GPUs with compute capability 2.x, which can improve the efficacy of the off-chip memory access and is transparent to application programs.

2.4.3 Instruction Throughput

On the GPU card, costs of different instructions are different. To maximize the instruction throughput, instructions of low cost should be used as many as possible. For instance, on devices of compute capability 1.0, there are instructions provided only for 24-bit integer multiplication, and 32-bit integer multiplication is implemented via multiple instructions as it is not natively supported. Thus, if possible, the 24-bit integer multiplication should be used. However, on devices of compute capability 2.0, there are native instructions for 32-bit integer multiplication, but 24-bit integer multiplication is not natively supported, which needs multiple instructions. As a result, 32-bit integer instructions should be used on devices of compute capability 2.0.

For the execution of threads within a warp, if there are divergent branches across different threads, different execution paths will be executed sequentially, as shown in Figure 2.2, and performance will be affected. So divergent branches should be minimized for threads within a warp.

When barrier synchronization is needed across different threads and the number of threads that need to be synchronized is larger, more potential time will be wasted since more threads need to wait for others to reach the barrier point. In addition to the barrier function `__syncthreads()` to synchronize all threads within a block, CUDA provides barrier functions that can synchronize only part of the threads within a block. Thus, when possible, we should only synchronize threads that are involved in the data communication to reduce the synchronization overhead.

Chapter 3

GPU Virtualization

3.1 Overview

GPUs are becoming increasingly popular as accelerators for core computational kernels in scientific and enterprise computing applications. The advent of programming models has further accelerated the adoption of GPUs by allowing many applications and high-level libraries to be ported to them. While GPUs have proliferated into high-end computing systems, current programming models require application execution to be tightly coupled to the physical GPU hardware. On the other hand, recent developments in virtualization techniques have advocated decoupling the application view of "local hardware resources" (such as processors and storage) from the physical hardware itself. That is, each application (or user) gets a "virtual independent view" of a potentially shared set of physical resources. Such decoupling has many advantages, including ease of management, ability to hot-swap the available physical resources on demand, improved resource utilization, and fault tolerance.

However, with the current GPU programming model implementations, such virtualization is not possible. To address this situation, we investigate the feasibility of virtualizing GPUs in such environments, allowing for compute nodes to ***transparently*** view remote GPUs as ***local virtual GPUs***. To achieve this goal, we propose a new implementation of the OpenCL programming model, called Virtual OpenCL, or ***VOCL***. The VOCL framework supports the OpenCL-1.1 API but with the primary difference that it allows an application to view all GPUs available in a system (including remote GPUs) as local virtual GPUs. VOCL internally uses the Message Passing Interface (MPI) [56] for data management associated with remote GPUs and utilizes several techniques, including kernel-argument caching and data-transfer pipelining, to improve performance.

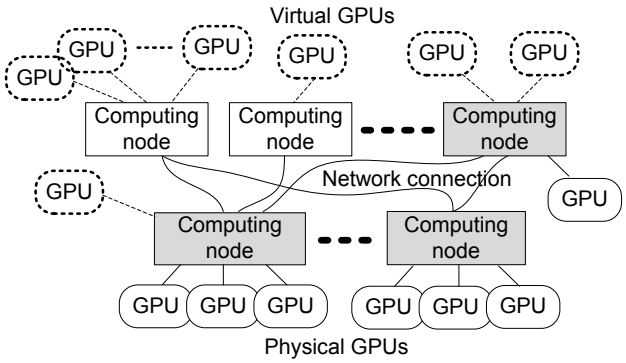


Figure 3.1: Transparent GPU Virtualization

We note that this work does not deal with using GPUs on virtual machines, which essentially provide operating system-level or even lower-level virtualization techniques (that is, full or paravirtualization). Instead, it deals with user-level virtualization of the GPU devices themselves. Unlike full or paravirtualization using virtual machines, VOCL does not handle security and operating system-level access isolation. However, it does provide similar usage and management benefits and the added benefit of being able to

transparently utilize remote GPUs. As illustrated in Figure 3.1, VOCL allows a user to construct a virtual system that has, for example, 32 virtual GPUs, even though no physical machine in the entire system might have 32 collocated physical GPUs.

We describe here the VOCL framework and the optimizations that we use to improve its performance. The optimizations mainly focus on efficient data transfer between the local node and the remote node, which include the kernel-argument caching, data-transfer pipelining, and the GPU synchronization to reduce the overhead of data communication across different nodes. We also present a detailed evaluation of the framework. The evaluation includes the performance improvement brought by our proposed optimizations on data transfer to and from GPUs associated with such virtualization. Also provided is the overheads for various OpenCL functions. Finally, we evaluate our framework with several real application kernels, including SGEMM/DGEMM, N-body computations, matrix transpose kernels, and the computational biology Smith-Waterman application.

We first, in this chapter, present the VOCL framework and the optimizations on data transfer between local host memory and remote device memory. Then, in Chapter 4, we present the GPU synchronization strategy that can be used to reduce the number of kernel launches for inter-work-group data communication. Finally, in Chapter 5, we extend the VOCL framework to support live virtual GPU migration across physical GPUs.

3.2 Related Work

Several researchers have studied GPU virtualization in large-scale heterogeneous computing systems. Till now, GPU virtualization frameworks such as rCUDA [27], vCUDA [72], and MOSIX-VCL [15] have been proposed to generalize the utility of GPUs in a system.

With these frameworks, application programs can use more GPUs than that can be installed locally, and GPUs can be shared by multiple programs. In this section, we briefly describe each of the existing frameworks and compare them to our VOCL framework in four aspects: (1) support for the virtualization of local or remote GPUs; (2) transparent utilization of GPUs in the system; (3) support for the asynchronous data transfer, which is a key feature of GPU programming models to overlap program execution and data transfers; and (4) achieved bandwidth between host memory and device memory. Among the four frameworks, rCUDA, MOSIX-VCL, and our VOCL framework use the QDR InfiniBand as the network connection, so the achieved bandwidth is an indication of how well a framework is designed and optimized.

rCUDA [27] is to support programs to use remote GPUs. With rCUDA, programs can take advantage of GPUs even though there are no GPUs installed in a compute node. rCUDA is based on the CUDA programming model, which uses some C extensions to support the kernel function call. The C extension is recognized by only the CUDA compiler `nvcc`, thus rCUDA needs to change the way of the kernel function call. In addition, rCUDA requires that all CUDA kernels be stored in a different file to be shipped to the remote node for execution. rCUDA changes the way of the kernel function call, and programmers need to change the source code to use rCUDA. Thus, it is *nontransparent*. Currently, rCUDA only addresses the use of GPUs in a single remote node, and no information is provided to use GPUs in multiple compute nodes. As mentioned above, asynchronous data transfer is a key capability in GPU programming models to overlap program execution and data transfer. In rCUDA, asynchronous data transfer is partially addressed. That is, when device to host (D2H) data transfer is in progress, asynchronous data transfer from

host memory to device memory (H2D) cannot be issued. As for the data transfer bandwidth, rCUDA achieves the overall data transfer bandwidth of 1.367 GB/s [28] when using the QDR InfiniBand as the network connection, corresponding to 51.5% of the bandwidth in a native CUDA environment.

Shi et al. [72] proposed the vCUDA framework that allows high-performance computing applications running in virtual machines (VMs) to benefit from GPU acceleration. This work considers OS-level virtualization of GPUs, and the virtualization overhead comes from the usage of VMs. Therefore, GPUs that can be used in vCUDA are restricted in a single node. In vCUDA, GPU sharing is achieved by creating multiple VM instances in a node. Using the API interception to capture CUDA function calls in a vCUDA wrapper library, vCUDA redirects API function calls on the guest operating system (OS) to the host OS, where native CUDA functions are called to perform GPU computing. Compared to rCUDA, vCUDA does not consider the use of remote GPUs and focuses on the use of local GPUs. In vCUDA, overhead is introduced by data transfer between different VM instances, and such data communication can be performed in two modes—*shared mode* and *transmission mode*. In the shared mode, vCUDA uses a memory buffer that is shared by the guest OS and the host OS. When a data transfer happens between the guest OS and the host OS, the sender writes data to the shared buffer, and the receiver then reads data from the buffer. In the transmission mode, the VM instance calls the TCP sockets for data transfer. Comparing the two data communication modes, the shared mode has much better performance than the transmission mode. vCUDA does not address the asynchronous data transfer, and there are no optimizations proposed for the data transfer in either data communication mode. Since vCUDA resolves the GPU sharing on a local node, the achieved

data transfer bandwidth is incomparable to that of the other three frameworks. Overall, vCUDA is complementary to the other three frameworks.

Barak et al. [14,15] proposed the MOSIX-VCL framework to transparently use cluster-based GPUs. MOSIX-VCL is based on the OpenCL programming model. It is to support transparent virtualization of both local and remote GPUs. With VCL, all GPUs in a system can be used as if they were installed locally, and no source code modification is needed. However, same as rCUDA and vCUDA, VCL does not address the asynchronous data transfer, and no optimization is performed to improve the data transfer bandwidth. In addition, VCL has an issue that it does not really address the data transfer between host memory and device memory. Specifically, we wrote a microbenchmark to measure the data transfer bandwidth between local host memory and remote device memory. The measured bandwidth in VCL is much higher than that of the native OpenCL, as shown in Table 3.1. [1] One possible reason for this behavior is that the data transfer is not completed even after the OpenCL flush function `clFinish()` returns.

Table 3.1: Data Transfer Bandwidth (GB/s) of the MOSIX-VCL Framework

	Native OpenCL	VCL, Local	VCL, Remote
GPU memory write	3.735	7.267	7.230
GPU memory read	3.854	11.421	11.389

Our proposed VOCL framework implements the same functionality as MOSIX-VCL. It is based on the OpenCL programming model and uses MPI for data communication between different nodes. VOCL supports the transparent virtualization of both local and

[1] We asked a VCL framework developer to help us measure the bandwidth of the native OpenCL, VCL using local GPUs, and VCL using remote GPUs with the QDR InfiniBand as the network connection.

remote GPUs in a system, and applications can utilize GPUs without any source code modification. In contrast to previous frameworks, VOCL fully addresses the asynchronous data transfer between host memory and device memory, and there is no dependency between H2D and D2H data transfers. We also proposed a performance model for optimizing the data transfer between host memory and device memory. Based on the performance model, we proposed several optimization techniques to improve the data transfer bandwidth. According to our experimental results, VOCL achieves 2.33 GB/s for the H2D data transfer and 1.985 GB/s for the D2H data transfer when using the QDR InfiniBand to connect different compute nodes, corresponding to 80% – 85% of the bandwidth in a native environment.

In summary, properties of the four aforementioned frameworks are listed in Table 3.2.

Table 3.2: Comparison of the Various GPU Virtualization Frameworks

Frameworks	Support local or remote GPUs	Source code modification	Overhead of the framework, caused by additional data transfer	Support of asynchronous data transfer	Achieved data transfer bandwidth over QDR InfiniBand
vCUDA	Local	No	Between different VMs on a single node	Not addressed	Not applicable
rCUDA	Remote	Yes	Between different compute nodes	Partially supported. H2D transfer cannot be issued when D2H transfer is in progress	1.367 GB/s over the QDR InfiniBand
MOSIX-VCL	Both	No	Remote GPUs, between compute nodes. Local GPUs, need one additional phase of data transfer	Not addressed	Invalid results as shown in Table 3.1
VOCL	Both	No	Remote GPUs, between compute nodes. Local GPUs, almost no overhead	Full support of asynchronous data transfer for both H2D and D2H, optimized	H2D 2.330GB/s, D2H 1.985GB/s

3.3 Virtual OpenCL Framework

The VOCL framework consists of the VOCL library on the local node and a VOCL proxy process on each remote node, as shown in Figure 3.2. The VOCL library exposes the OpenCL API to applications and is responsible for sending information about the OpenCL calls made by the application to the VOCL proxy and receiving kernel results from the VOCL proxy using MPI. The VOCL proxy is essentially a service provider for applications, allowing them to utilize GPUs remotely. They are expected to be set up initially (for example, by the system administrator) on all nodes that would be providing virtual GPUs to potential applications. The proxy is responsible for handling messages from the VOCL library, executing native OpenCL functions on physical GPUs, and sending results back to the VOCL library.

We chose OpenCL as the programming model for two reasons. First, OpenCL provides general support for multiple accelerators (including AMD GPUs, NVIDIA GPUs, Intel accelerators, and the Cell/BE) as well as for general-purpose multicore processors (including AMD CPUs, ARM CPUs, and Intel CPUs). By supporting OpenCL, our VOCL framework can support transparent virtualization of various accelerators and multicore processors. Second, OpenCL is primarily based on a library-based interface rather than a compiler-supported user interface such as CUDA. Thus, a runtime library can easily implement the OpenCL interface without the need to design a new compiler.

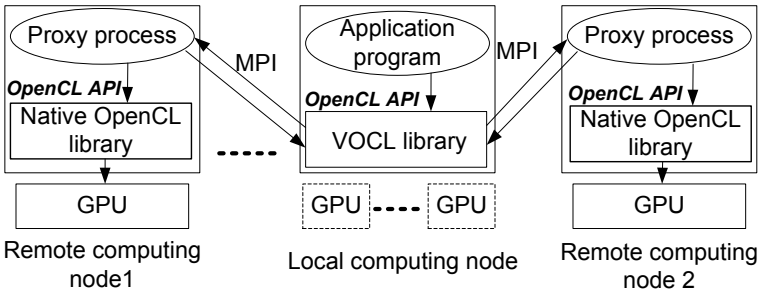


Figure 3.2: Virtual OpenCL Framework

3.3.1 VOCL Library

VOCL is compatible with the native OpenCL implementation available on the system with respect to its application programming interface (API) as well as its application binary interface (ABI). Specifically, since the VOCL library presents the OpenCL API to users, all OpenCL applications can use it without any source code modification. At the same time, VOCL is built on top of the native OpenCL library and utilizes the same OpenCL headers on the system. Thus, applications that have been compiled with the native OpenCL infrastructure need only to be relinked with VOCL and do not have to be recompiled. Furthermore, if the native OpenCL library is a shared library and the application has opted to do dynamic linking of the library (which is the common usage mode for most libraries and default linker mode for most compilers), such linking can be performed at runtime just by preloading the VOCL library through the environment variable `LD_PRELOAD`.

The VOCL library is responsible for managing all virtual GPUs exposed to the application. Thus, if the system has multiple nodes, each equipped with GPUs, the VOCL library is responsible for coordinating with the VOCL proxy processes on all these nodes. Moreover, the library should be aware of the locally installed physical GPUs and call the

native OpenCL functions on them if they are available.

3.3.1.1 VOCL Function Operations

When an OpenCL function is called, VOCL performs the following operations.

- Check whether the physical GPU to which a virtual GPU is mapped is local or remote.
- If the virtual GPU is mapped to a local physical GPU, call the native OpenCL function and return.
- If the virtual GPU is mapped to a remote physical GPU, check whether the communication channels between applications and proxy processes have been connected. If not, call the `MPI_Comm_connect()` function to establish the communication channel.
- Pack the input parameters of the OpenCL function into a structure and call `MPI_Isend()` to send the message (referred to as *control message*) to the VOCL proxy. Here, a different MPI message tag is used for each OpenCL function to differentiate them.
- Call `MPI_Irecv()` to receive output and error information from the proxy process, if necessary.
- Call `MPI_Wait()` when the application requires completion of pending OpenCL operations (e.g., in blocking OpenCL calls or flush calls).

This function call scenario is illustrated in Figure 3.3.

```
clSetKernelArg(kernel, argIndex, argValue, argSize)
{
    //check whether the proxy process is created
    checkProxyProcess();

    //message to be sent to the proxy process
    struct strSetKernelArg setKernelArg;

    //initialize the message according to inputs
    setKernelArg = kernel,argIndex,argValue,argSize;

    //send parameters to remote node
    MPI_Isend(&setKernelArg, sizeof(setKernelArg),
        MPI_BYTE, 0, SET_KERNEL_ARG, conMsgComm, ...);

    //send argument value to the remote node
    MPI_Isend((void *)arg_value, arg_size, MPI_BYTE,
        0, SET_KERNEL_ARG1, dataComm, ...);

    //wait for return code from the real OpenCL function
    MPI_Irecv(&setKernelArg, sizeof(setKernelArg),
        MPI_BYTE, 0, SET_KERNEL_ARG, conMsgComm, ...);
    //guarantee return code is received
    MPI_Waitall(requests, status);
    return setKernelArg.res;
}
```

Figure 3.3: Pseudo Code for the Function `clSetKernelArg()`

3.3.1.2 VOCL Abstraction

In OpenCL, kernel execution is performed within a host-defined context, which includes several types of objects such as devices, program objects, memory objects, command queues, and kernels. A context can contain multiple devices; therefore, objects such as command queues within the context need to be mapped onto a specific device before computation can be performed. Since OpenCL can support computation on multiple GPUs, OpenCL objects contain additional information to identify which physical GPU the object belongs to. For example, when OpenCL returns a command queue (i.e., `cl_command_queue`), this object internally has enough information to distinguish which physical GPU the command queue resides on.

With VOCL, since the physical GPUs might be located on multiple physical nodes,

the VOCL library might internally be coordinating with the native OpenCL library on multiple nodes (through the VOCL proxy). Thus, VOCL needs to add an additional level of abstraction for these objects to identify which native OpenCL library to pass each object to. We show this in Figure 3.4. Specifically, within VOCL, we define an equivalent object for each OpenCL object. For each OpenCL object, its handle is translated to a VOCL handle with a unique value even when OpenCL handles share the same value. Together with the native OpenCL handle, the VOCL object contains additional information to identify which physical node (and thus, which native OpenCL library instance) the object corresponds to.

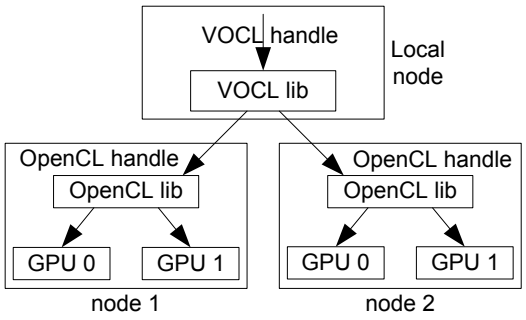


Figure 3.4: Multiple-Level Handle Translation

When a VOCL handle is used, VOCL will first translate it to the OpenCL handle. Then it sends the OpenCL handle to the corresponding compute node based on the MPI communication information contained in the VOCL object. However, care must be taken when a memory buffer object is used as a kernel input. As we know, a kernel argument is set by calling the function `clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)` and the argument `arg_value` is the pointer to the OpenCL device memory handle. But with the VOCL abstraction, `arg_value` is a pointer to a VOCL handle, as shown in Figure 3.5, and

is invalid for kernels. Moreover, from the arguments themselves in the function call of `clSetKernelArg()`, we cannot figure out whether a kernel argument is a memory handle or not. To address this problem, we write a parser to parse the kernel source code and figure out the device memory arguments in the kernel. As such, when `clSetKernelArg()` is called, VOCL can first translate a VOCL memory handle to an OpenCL memory handle based on the parser output. Then VOCL uses the OpenCL memory handle as the input to the native OpenCL function `clSetKernelArg()`. In this way, kernel arguments can be set correctly.

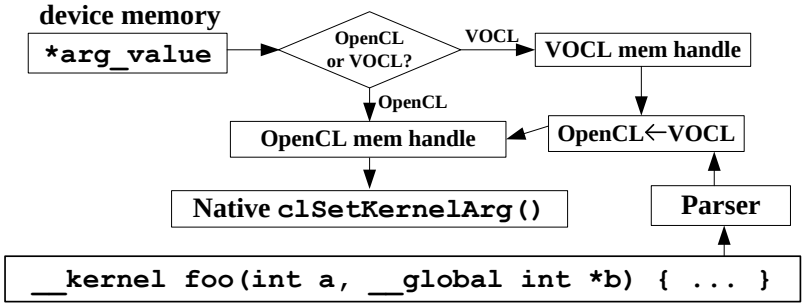


Figure 3.5: VOCL Device Memory Handle Processing in Kernel Argument Setting

3.3.2 VOCL Proxy

The VOCL proxy is responsible for (1) receiving connection requests from the VOCL library to establish communication channels with each application process, (2) receiving inputs from the VOCL library and executing them on its local GPUs, (3) sending output and error codes to the VOCL library, and (4) destroying the communication channels after the program execution has completed.

3.3.2.1 Managing Communication Channels

Communication channels between the VOCL library and the VOCL proxy are established and destroyed dynamically. Each proxy calls `MPI_Comm_accept()` to wait for connection requests from the VOCL library. When such a request is received, a channel is established between them, which is referred to as the *control message channel*. Once the application has completed utilizing the virtual GPU, the VOCL library sends a termination message to the proxy. Then `MPI_Comm_disconnect()` is called by both the VOCL library and the VOCL proxy to terminate the communication channel.

In the VOCL framework, each application can utilize GPUs on multiple remote nodes. Similarly, GPUs on a remote node can be used by multiple applications simultaneously. In addition, applications may start their execution at different times. Thus, the proxy should be able to accept connection requests from application processes at any time. To achieve this, we use an additional thread at the proxy that continuously waits for new incoming connection requests. When a connection request is received, this thread updates the communication channels such that messages sent by the VOCL library can be handled by the main proxy process, and the thread waits for the next connection request.

3.3.2.2 Handling Native OpenCL Function Calls

Once a control message channel is established between the VOCL proxy and the VOCL library, the proxy preposts buffers to receive control messages from the VOCL library (using nonblocking MPI receive operations). Each VOCL control message is only a few bytes large, so the buffers are preposted with a fixed maximum buffer size that is large enough for any control message. When a control message is received, it contains information on

which OpenCL function needs to be executed as well as information about any additional input data that need to be sent to the physical GPU. At this point, if any data need to be transferred to the GPU, the proxy posts additional receive buffers to receive this data from the VOCL library. It is worth noting that the actual data communication happens on a separate communicator to avoid conflicts with control messages; this communicator will be referred to as the *data channel*.

Specifically, for each control message, the proxy process performs the followed steps.

- When a control message is received, the corresponding OpenCL function is determined based on the message tag. Then the proxy process decodes the received message according to the OpenCL function. Depending on the specific OpenCL function, other messages may be received as inputs for the function execution in the data channel.

- Once all of the input data are available, the native OpenCL function is executed.

- Once the native OpenCL function completes, the proxy packs the output and sends it back to the VOCL library.

- If dependencies exist across different functions or if the current function is a blocking operation, the proxy waits for the current operation to finish and the result sent back to the VOCL library before the next OpenCL function is processed. On the other hand, if the OpenCL function is nonblocking, the proxy will send out the return code and continue processing other functions.

- Another nonblocking receive will be issued to replace the processed control message.

Since the number of messages received is not known beforehand, the proxy process uses an infinite loop to wait for messages. The infinite loop is terminated by a message with a specific tag. Once the termination message is received, the loop is ended and the MPI communicator is released.

The scenario on the proxy is illustrated in Figure 3.6.

```
1  MPI initialize
2
3  //prepost buffers to receive control messages
4  for i = 1 to BUFFER_NUM
5     MPI_Irecv(buff+i, size, MPI_BYTE, MPI_ANY_SOURCE,
6        MPI_ANY_TAG, conMsgComm, funcRequest+i);
7  end for
8
9  while loop
10    MPI_Waitany(numOpenCLFunc, funcRequest, &index, &status);
11    if status.MPI_TAG is SET_KERNEL_ARG
12       //receive messages
13       MPI_Irecv(argValue, argSize, MPI_BYTE, rank,
14          SET_KERNEL_ARG1, dataComm, &setArgRequest)
15       MPI_Wait(&setArgRequest, &setArgStatus);
16
17       //call the real opencl function
18       errcode_ret = clSetKernelArg(kernel, argIndex, argSize, argValue);
19
20       MPI_Isend(&errcode_ret, 1, MPI_INT, tag, rank, conMsgComm, &setArgRequest);
21       MPI_Wait(&setArgRequest, &setArgStatus);
22    end if
23
24    if status.MPI_TAG is proxyTermination
25       break;
26
27    //issue another nonblocking receive to replace the processed control message
28    MPI_Irecv(buff+index, size, MPI_BYTE, MPI_ANY_SOURCE,
29       MPI_ANY_TAG, conMsgComm, funcRequest+index);
30 end while
31
32 MPI finalize
```

Figure 3.6: Pseudo Code of the Proxy Process

3.4 VOCL Optimizations

The VOCL framework provides applications the ability of using all GPUs in the same way to accelerate their execution. However, we should reduce the virtualization overhead to as little as possible. Otherwise, performance improvement brought by using virtual GPUs would be negated by such overhead. Since VOCL internally calls native OpenCL functions for local GPUs, overhead of using local GPUs is expected to be very little as shown later. Thus in the following, our optimization focuses on reducing the overhead of using remote GPUs.

Compared to local GPUs, function calls on remote GPUs need one more phase of data transfer between the local node and the remote node. Specifically, if a function is executed without reading or writing device memory for a remote GPU, data transfer is performed only between local host memory and remote host memory. On the other hand, if device memory reads or writes are performed in a function call, data are transferred between local host memory and remote device memory, which includes two phases—between local host memory and remote host memory and between remote host memory and remote device memory. In general, overhead of data transfer depends on the amount of data to be transferred, network bandwidth, number of data transfers, as well as how data transfer is handled in the remote node.

In a typical OpenCL program, API functions for allocating and releasing OpenCL objects are called only a few times. Inputs and outputs of these functions are of tens of bytes. As a result, overhead of these functions is negligible in practice. But for functions related to kernel execution (GPU memory read/write, kernel argument setting, and kernel launch),

they can cause significant overhead for program execution in some scenarios. According to the work of Gregg et al. [34], when local GPUs are used, data transfer between host memory and device memory can slow down the program execution by 2 to 50 folds compared to the GPU processing alone. With one more phase of data transfer in using remote GPUs, it is expected that data transfer will cause even more overhead. Thus, optimizing such data transfer is of great importance for the use of remote GPUs.

To reduce these overheads, we have implemented three optimizations: (1) kernel argument caching to reduce the number of data transfers, (2) data transfer pipelining to improve the bandwidth between local host memory and remote device memory, and (3) modifications to error handling.

3.4.1 Kernel Argument Caching

When remote GPUs are used, execution of functions without accessing GPU memory needs the data transfer between local node and remote node and the data is of tens of bytes in general. If these functions are called only a few times (e.g., functions for OpenCL object allocation and release), the data transfer overhead involved in the use of remote GPUs can be ignored. But if a function is called thousands of times, it can cause significant overhead. One such example is the kernel argument set function `clSetKernelArg()`, which can be called thousands of times in some applications.

Table 3.3 presents the kernel execution overhead for VOCL (using a remote GPU) vs. the native OpenCL library for aligning 6K base-pair sequences using the Smith-Waterman application [73, 87]. We run the experiments on nodes connected with the QDR InfiniBand. Each node is installed with two Magny-Cours AMD CPUs (Each has eight cores.)

Table 3.3: Overhead (in ms) of Kernel Execution for Utilization of Remote GPUs

Function Name	Native OpenCL	VOCL Remote	Overhead
clSetKernelArg	4.33	420.45	416.12
cllEnqueueNDRangeKernel	1210.85	1316.92	106.07
Total kernel time	1215.18	1737.37	522.19

with host memory of 64 GB and two NVIDIA Tesla M2070 GPU cards each with 6GB global memory. The two GPUs are installed on two different PCI slots, one of which shares the PCIe bandwidth with an InfiniBand adapter, as shown in Figure 3.7. The compute nodes are installed with the CentOS 5.5 Linux operating system and the CUDA 3.2 toolkit. We use the MVAPICH2 [60] MPI implementation. Each of our experiments was conducted three times and the average is reported.

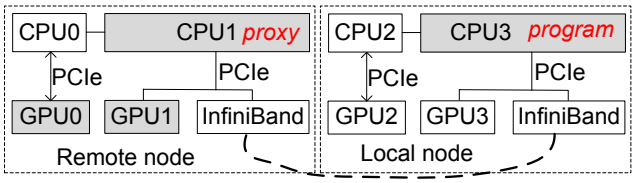


Figure 3.7: GPU Configuration and the Scenario for the Bandwidth Test

In this example, `clSetKernelArg()` has an overhead of 416.12 ms, which is four times more than that of the kernel execution. The reason is that although the overhead of each call is small, the function is called more than 86,000 times (the kernel is called 12,288 times, and 7 arguments have to be set for each call).

The basic idea of kernel argument caching is to combine the message transfers for multiple `clSetKernelArg()` calls. Instead of sending one message for each call of `clSetKernelArg()`, we send kernel arguments to the remote node only once for every kernel launch, irrespective of how many arguments the kernel has. Since all arguments

should be set before the kernel is launched, we just cache all the arguments locally at the VOCL library. When the kernel launch function is called, the arguments are sent to the proxy. The proxy performs two steps on being notified of the kernel launch: (1) it receives the argument message and sets the individual kernel arguments, and (2) it launches the kernel.

Table 3.4 shows the execution time of Smith-Waterman for aligning 6K base-pair sequences using our kernel argument caching approach. As we can see in the table, the execution time of `clSetKernelArg()` is reduced from 420.45 ms (Table 3.3) to 4.03 ms (Table 3.4). We notice a slight speedup compared with native OpenCL; the reason is that, in VOCL, kernel arguments are cached in host memory and are not passed to the GPU immediately. We also notice a slightly higher overhead for the kernel execution time (increase from 1316.92 ms to 1344.01 ms), which is due to the additional kernel argument data passed to the proxy within this call. Overall, the total kernel execution time decreases from 1737.37 ms to 1348.04 ms, or by 22.41%.

Table 3.4: Overhead (in ms) of Kernel Execution with Kernel Argument Caching Optimization

Function Name	Native OpenCL	VOCL remote	Overhead
clSetKernelArg	4.33	4.03	-0.30
clEnqueueNDRangeKernel	1210.85	1344.01	133.17
Total kernel time	1215.18	1348.04	132.71

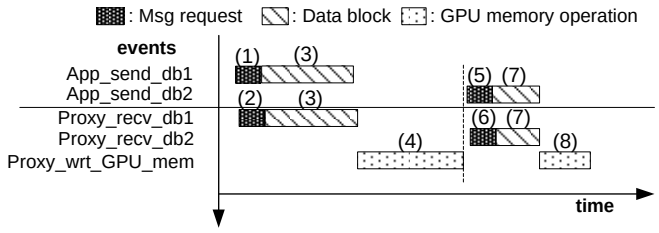
3.4.2 Data Transfer Pipelining

As mentioned above, two types of data need to be transferred between the VOCL library and the VOCL proxy in the use of remote GPUs. The first type is the input arguments to

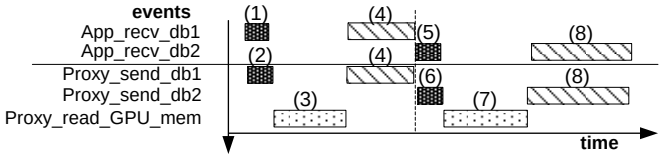
OpenCL functions without GPU memory accesses involved; this type of data is transferred from local host memory to remote host memory. The size of such input arguments is at most a few hundred bytes, and the transfer cannot be started in one function until execution of its previous functions is completed. Such data transfers cause negligible overhead and pipelining them cannot bring useful benefits for program execution.

The second type is the GPU memory accesses, in which data are transferred from local host memory to remote GPU memory. Such data transfers have two stages: (1) between the VOCL library and the VOCL proxy and (2) between the VOCL proxy and the GPU. In a naïve implementation of VOCL, these two stages would be serialized, and buffers to store the data are dynamically allocated and released in the proxy. Such an implementation, though simple, has two primary problems. First, because of the lazy memory allocation in today's operating systems (e.g., Linux), the dynamic memory allocation can adversely affect the data transfer bandwidth for both the *network communication* (between local host memory and remote host memory) and *GPU communication* (between remote host memory and remote device memory). Specifically, for GPU memory write, bandwidth of the network communication is affected, and for the GPU memory read, bandwidth of the GPU communication is affected. Second, there is no overlap between the network communication and the GPU communication. That is, the second stage can be started only after the first stage is finished since each data chunk is transferred as a single block in the naïve implementation. Furthermore, when data transfers are performed in the blocking mode, different data chunks cannot be overlapped, either, as explained in Figure 3.8.

We wrote two microbenchmarks to verify the first problem. One is for the network communication from local host memory to remote host memory using MPI, and the other



(a) Write to the GPU Memory



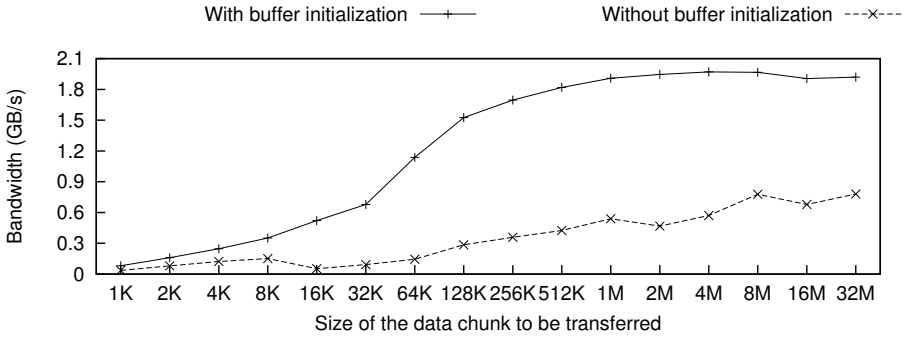
(b) Read from the GPU Memory

Figure 3.8: Blocking Data Transfer Scenarios without Pipelining. For instance, GPU memory write is performed in the following steps: (1) the host sends a data send request to the proxy; (2) the proxy receives the data send request; (3) data block is transferred from the host to the proxy; (4) after transfer of the data block is completed, the proxy transfers the data block to the GPU. Steps (5) - (8) repeat the above procedure for another data block.

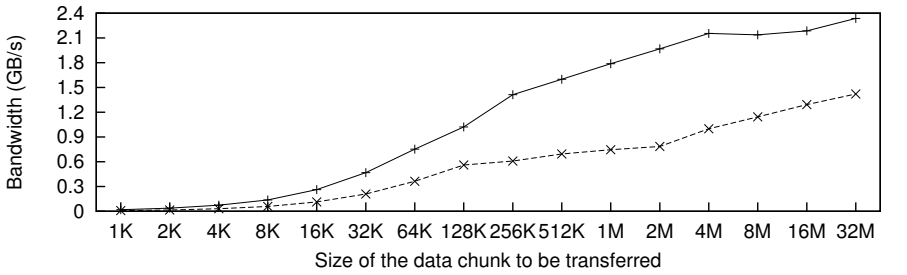
is for the GPU communication from remote device memory to remote host memory. In each microbenchmark, we measured the bandwidth in two scenarios. One is that buffers used to store data in the proxy are allocated without initialization. Because of the lazy memory allocation, when the data communication happens, the VOCL proxy needs to actually allocate the memory first, and then writes data to the memory. The other scenario is that all elements of the buffers are initialized to 0 after memory allocation. As such, buffers are actually allocated in the proxy, and when data communication happens, the proxy can write data to the buffers immediately.

Figure 3.9 shows the bandwidth in the above two scenarios; where Figure 3.9(a) is for the network communication, and Figure 3.9(b) is for the GPU communication. As we

can see, the lazy memory allocation can significantly impact the bandwidth of both the network and GPU communication. When using a buffer size of 32 MB, the lazy memory allocation can reduce the bandwidth by more than 50%.



(a) Network Communication from Local Host Memory to Remote Host Memory



(b) GPU Communication from Remote Device memory to Remote Host Memory

Figure 3.9: Impact of the Lazy Memory Allocation on Data Transfer Bandwidth

To avoid the lazy memory allocation problem, we use a buffer pool for the data storage in the proxy, as shown in Figure 3.10. The buffer pool is allocated when a control message channel is established between the VOCL library and the proxy. In addition, all the buffers are initialized to 0 to ensure that they are actually allocated to eliminate the impact of the lazy memory allocation. Each buffer is of size S bytes. During the data transfer, when a data chunk is less than S bytes, e.g. data chunks 1 and 2 in Figure 3.10, it is transferred as a contiguous block. As for a data chunk that is larger than S bytes, e.g. data chunk 3,

it is segmented into smaller blocks to fit into the buffers. Since the number of buffers in the pool is limited, we reuse the buffers in a circular fashion. Note that before we reuse a buffer, we have to ensure that the previous data transfers (both the network communication and the GPU communication) have completed. The number of buffers and the buffer size dictate how often we need to wait for such completion, and thus have to be carefully configured as explained later.

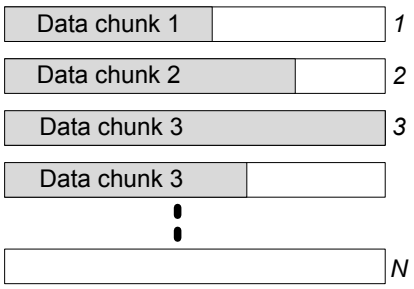
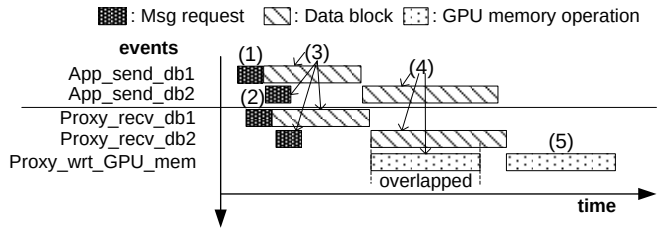


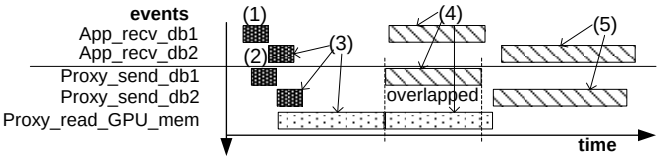
Figure 3.10: Buffer Pool in the Proxy Process

As for the problem that there is no overlap between the two stages of data transfers, we design a data pipelining mechanism and use the buffer pool for data storage in the proxy. Specifically, with pipelining, the first stage transfer of one data chunk can be done concurrently with the second stage transfer of another data chunk, as shown in Figure 3.11. In addition, for a large data chunk that needs to be segmented into multiple data blocks, transfer of the different data blocks can be overlapped, too.

Using the buffer pool and the pipelining mechanism, the data transfer between local host memory and remote device memory is affected by a few factors: (1) bandwidth, including both the network communication bandwidth and the GPU communication bandwidth, (2) buffer pool configuration, i.e., the number of buffers and the size of each buffer



(a) Nonblocking Write to the GPU Memory



(b) Nonblocking Read from the GPU Memory

Figure 3.11: Nonblocking Data Transfer Scenarios with Pipelining. For instance, GPU memory write is performed in the following steps: (1) the host sends a data send request to the proxy; (2) the proxy receives the data send request; (3) a data block is transferred from the host to the proxy. At the same time, the host sends a second data send request, and the proxy receives the second data send request; (4) after the proxy receives the whole first data block, it sends the data block to the GPU. At the same time, the second data block is transferred from the host to the proxy; (5) the proxy sends the second data block to the GPU after both the first data block is transferred to the GPU and the whole second send data block is received.

in the proxy, (3) data transfer handling in the proxy, and (4) data transfer mode, i.e., blocking data transfer and nonblocking data transfer. In the following, we first calculate the overall bandwidth between local host memory and remote device memory quantitatively. Based on the calculation, we analyze how each factor affects the overall bandwidth and how we optimize the data transfer between local host memory and remote device memory to achieve the best performance.

3.4.2.1 Performance Model of the Overall Bandwidth of GPU Memory Access

Let B_n be the network communication bandwidth, B_g be the GPU communication bandwidth, and S be the size of each buffer in the pool, then the time needed to transfer a data chunk of size C can be calculated as follows.

1. If $C \leq S$, the data chunk is transferred as a single data block. Then the data transfer time is $t = \frac{C}{B_n} + \frac{C}{B_g}$, and the overall bandwidth is $B = \frac{B_n B_g}{B_n + B_g}$.

2. If $C > S$, the data chunk needs to be segmented into multiple data blocks, which are of size S except the last one (equal or less than S). Let $\acute{n}$ be the number of blocks with size S, then $\acute{n} = \lceil \frac{C}{S} \rceil - 1$, and the size of the last block is $S_l = C - \acute{n}S$. If the network communication bandwidth is lower than that of the GPU communication, the data transfer time is

$$t = \frac{\acute{n}S}{B_n} + \frac{S_l}{B_g} + \max \begin{cases} \frac{S_l}{B_n} \\ \frac{S}{B_g} \end{cases} \tag{3.1}$$

   Otherwise, with the network communication bandwidth higher than that of the GPU communication, the data transfer time is

$$t = \frac{s}{B_n} + \frac{C}{B_g} \tag{3.2}$$

To simplify our analysis, we assume that the data chunk can be segmented into n data blocks with size S, i.e. $n = \frac{C}{S}$, and the network bandwidth is lower than the GPU communication bandwidth (This is true in most scenarios in practice.), then the total data transfer time is

$$t = n\frac{S}{B_n} + \frac{S}{B_g} \tag{3.3}$$

and the overall bandwidth is

$$B = \frac{B_n B_g}{B_g + B_n/n} \tag{3.4}$$

From Equation (3.4), we have the following observations. The overall bandwidth is less than both the network communication bandwidth and the GPU communication bandwidth. When the Gigabit Ethernet is used for the network connection, i.e. $B_g >> B_n$, the overall bandwidth B is very close to B_n. In this case, since the network bandwidth has been fully utilized, there is little room left for additional bandwidth improvement. If we use the QDR InfiniBand as the network connection, the network bandwidth becomes comparable to the GPU communication bandwidth. In this case, there is some room for the overall bandwidth improvement. As can be seen in Figure 3.9, with a larger buffer size, higher bandwidth can be achieved for both the network communication and the GPU communication. For instance, for data chunks of 128 KB, the network and GPU communication bandwidth is 1.525 GB/s and 1.022 GB/s, respectively. For data chunks of 32 MB, the bandwidth increases to 1.919 GB/s and 2.336 GB/s, respectively. Thus, if only the bandwidth is considered, we should set the buffer size in the pool to be as large as possible.

On the other hand, buffers with a larger size have other adverse impact on the overall bandwidth. Specifically, with a larger buffer size S, the number of buffers needed for a particular data chunk becomes smaller, i.e., n becomes smaller. From Equation (3.4), a smaller n corresponds to a lower overall bandwidth B. From the above analysis, either too large or too small buffer size can adversely affect the overall bandwidth, as we will demonstrate in the next section.

Another factor that can affect the bandwidth is the number of buffers in the pool. Technically, we can use only two buffers to store data. But there are at lease two issues

if only two buffers are configured. First, the number of buffers can affect how well the nonblocking transfer mode is supported. Using the GPU memory write as an example, we send data from local host memory to remote host memory by calling the MPI asynchronous data transfer function `MPI_Isend()`. Each time we send out a data chunk, we need a buffer in the proxy to receive it. Configuring more buffers in the pool can enable more data chunks to be transferred in the nonblocking way. The reason is, when buffers are used out, we need to wait for completion of a previous buffer before we can issue an MPI data send, and the data chunk is not transferred in the full nonblocking mode. Second, fewer buffers in the pool also means that the proxy process needs to switch across waiting for completion, issuing MPI data receive, and performing GPU memory write. The task switch can also impact the overall bandwidth in some degree, as we will present in the next section. From the above analysis, we should configure as many buffers in the pool as possible.

Data transfer in the proxy includes the network communication and the GPU communication, which are in different directions. In other words, when an application writes data to the GPU, the proxy reads data from the network and writes it to the GPU. When an application reads data from the GPU, the proxy reads data from the GPU memory and writes it to the network. If we use only one thread to do both the data read and write, the thread needs to handle both the network communication and the GPU communication, which can affect the data transfer bandwidth. To mitigate the impact, we create a helper thread to handle the data transfer in the proxy. With two threads, one thread handles the network communication, and the other thread handles the GPU communication. In this way, when

a data chunk is transferred, neither thread needs to switch between the network communication and the GPU communication, which in turn can improve the overall bandwidth in some degree.

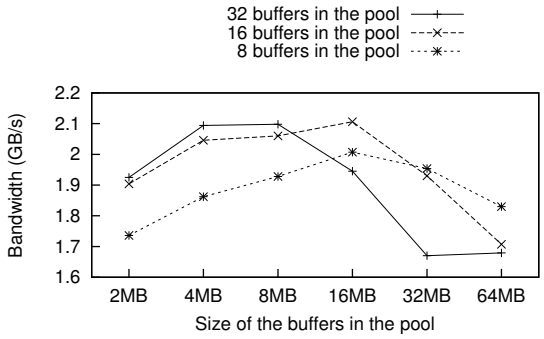
Using separate threads for network communication and GPU communication can improve the bandwidth. Moreover, we can bind each thread to a separate core considering the fact that today's CPUs consist of multiple cores in general. By binding each thread to a different CPU core, that CPU core can devote all its resources for the particular data transfer operation, which can further improve the bandwidth, particularly for the GPU memory read, as shown in the next section.

3.4.2.2 Data Transfer Bandwidth Evaluation via a Microbenchmark

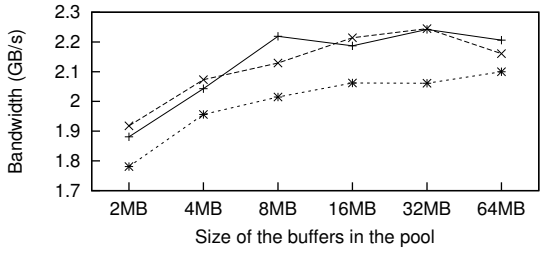
In this section, we show the achieved data transfer bandwidth of the GPU memory write and GPU memory read in using a remote GPU. We wrote a microbenchmark to transfer a data chunk of 1 GB between local host memory and remote device memory. We used a buffer pool for the data storage in the proxy as mentioned in Section 3.4.2. With the pipelining mechanism, different stages of data transfers are overlapped. We use the platform in Figure 3.7 for our evaluation. We present the data transfer bandwidth with different configurations of the buffer pool (including different buffer sizes and different number of buffers), using multiple threads to handle data transfers in the proxy, and binding each thread to a separate core.

Figures 3.12 and 3.13 show the bandwidth of the GPU memory write and read, respectively. Figures 3.12(a) and 3.13(a) show the case of using a single thread; Figures 3.12(b) and 3.13(b) are for the case of using multiple threads binded to the same core; and Figures 3.12(c) and 3.13(c) are for the case of binding each thread to a separate core.

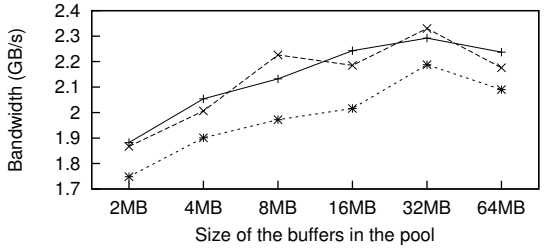
From the results in Figures 3.12 and 3.13, we have the following observations. First, for both the GPU memory write and read, using a separate thread (on the same CPU core) to handle the network communication and the GPU communication in the proxy can improve the bandwidth. Specifically, bandwidth of the GPU memory write increases from 2.106 GB/s to 2.245 GB/s, corresponding to a relative increase of 6.6%. Similarly, bandwidth of the GPU memory read increases from 1.442 GB/s to 1.446 GB/s. Second, binding each thread of the proxy process to a different CPU core can further improve the overall bandwidth, particularly for the GPU memory read. For the GPU memory write, the overall bandwidth increases from 2.245 GB/s to 2.330 GB/s with a relative increase of 3.8%. For the GPU memory read, the bandwidth increases from 1.446 GB/s to 1.985 GB/s with a relative increase of 37.3%. From the above results, using separate threads and binding each thread to a different CPU core is an effective approach to improve the bandwidth of both the GPU memory read and write. Third, the number of buffers and the size of each buffer can affect the overall bandwidth, too. When using separate threads and binding each thread to a different CPU core, increasing the buffer size increases the overall bandwidth first and decreases the bandwidth later. This is true for both the GPU memory read and write. As for the impact of the number of buffers in the buffer pool, GPU memory write achieves better performance with 16 and 32 buffers than with 8 buffers. For the GPU memory read, each number of buffers in the pool can achieve better performance than the others, if different buffer sizes are considered. Overall, the highest GPU memory write bandwidth that we can achieve is 2.330 GB/s by configuring 16 buffers, and each buffer is of 32 MB. For the GPU memory read, the highest bandwidth is 1.985 GB/s with 32 buffers configured, and each buffer is of 8MB.



(a) Single Thread in a Single Core

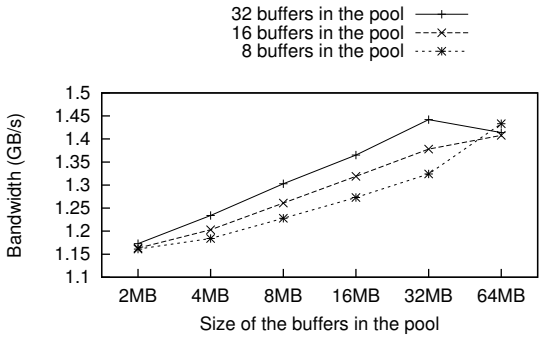


(b) Both Threads in a Single Core

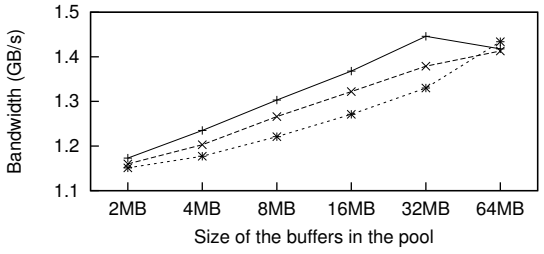


(c) Each Thread in Different Cores

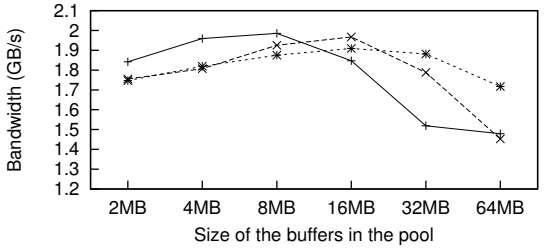
Figure 3.12: Bandwidth of Remote GPU Memory Write



(a) Single Thread in a Single Core



(b) Both Threads in a Single Core



(c) Each Thread in Different Cores

Figure 3.13: Bandwidth of Remote GPU Memory Read

In the following, configurations of the buffer pools corresponding to the highest bandwidth are used for our experiments in Section 3.5.

3.4.3 Error Return Handling

Most OpenCL functions provide a return code to the user: either `CL_SUCCESS` or an appropriate error code. Such return values, however, are tricky for VOCL to handle, especially for nonblocking operations. The OpenCL specification does not define how error codes are handled for nonblocking operations. For instance, if the GPU is not functional, is a nonblocking operation that tries to move data to the GPU expected to return an error?

While the OpenCL specification does not describe the return value in such cases, current OpenCL implementations do return an error. For VOCL, however, since every OpenCL operation is translated into a network operation, significant overhead can occur for nonblocking operations if the VOCL library has to wait until the OpenCL request is transferred over the network, a local GPU operation is initiated by the VOCL proxy, and the return code is sent back.

We believe this is an oversight in the OpenCL specification, since all other specifications or user documents that we are aware of (including MPI, CUDA, and InfiniBand) do not require nonblocking operations to return such errors—the corresponding *wait-for-completion* operation can return these errors at a later time. In our implementation, therefore, we assume this behavior and return such errors during the corresponding *wait* operation.

3.5 Experimental Evaluation

In this section, we evaluate the efficiency of the proposed VOCL framework. First, we analyze the overhead of individual OpenCL operations with VOCL via a few microbenchmarks. Then, we quantitatively evaluate the VOCL framework with several application kernels: SGEMM/DGEMM, matrix transpose, n-body [65], and Smith-Waterman [73,87]. We ran the experiments using the platform shown in Figure 3.7. Each of our experiments was conducted three times, and the average is reported.

3.5.1 Microbenchmark Evaluation

In this section, we study the overhead of various individual OpenCL operations using the SHOC benchmark suite [24] and a benchmark suite developed within our group, the Synergy Laboratory [78].

3.5.1.1 Initialization/Finalization Overheads

In this section, we study the performance of initializing and finalizing OpenCL objects within the VOCL framework. Overhead of these functions are mainly caused by the transfer of function parameters as described in Section 3.4.2. These functions and their overhead are listed in Table 3.5. As shown in the table, for most functions, the overhead caused by VOCL is minimal. The one exception to this is the `clGetPlatformIDs()` function which has the overhead of 402.68 ms. The reason for this overhead is that `clGetPlatformIDs()` is typically the first OpenCL function executed by the application in order to query the platform. Therefore, the VOCL framework performs most of its

initialization during this function, including setting up the MPI communication channels as described in Section 3.3.1.1.

Table 3.5: Overhead of OpenCL API Functions for Resource Initialization/Release (Unit: ms)

Function Name	Native OpenCL	VOCL (remote)	Overhead
clGetPlatformIDs	50.84	453.52	402.68
clGetDeviceIDs	0.002	0.173	0.171
clCreateContext	253.28	254.11	0.83
clCreateCommandQueue	0.018	0.044	0.026
clCreateProgramWithSource	0.009	0.042	0.033
clBuildProgram	488.82	480.90	-7.92
clCreateBuffer	0.025	0.051	0.026
clCreateKernel	0.019	0.030	0.011
clReleaseKernel	0.003	0.012	0.009
clReleaseMemObj	0.004	0.011	0.007
clReleaseProgram	0.375	0.291	-0.084
clReleaseCmdQueue	0.051	0.059	0.008
clReleaseContext	177.47	178.43	0.96

The overall overhead caused by all the initialization and finalization functions together is a few hundred milliseconds. However, this overhead is a one-time overhead unrelated to the total program execution time. Thus, in practice, for any program that executes for a reasonably long time (e.g., a few tens of seconds), these overheads do not adversely impact the performance of VOCL on long-running applications.

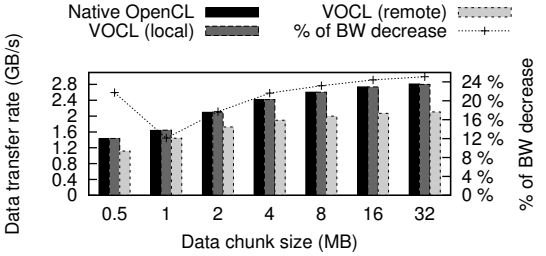
3.5.1.2 Performance of Kernel Execution on the GPU

Kernel execution on the GPU would be the same no matter which host processor launches the kernel. Thus, utilizing remote GPUs via VOCL should not affect the kernel execution on the GPU card. By evaluating the SHOC microbenchmark [24] with VOCL, we verify

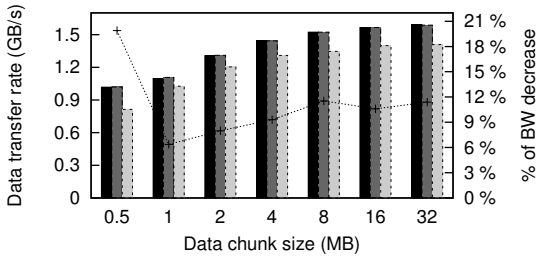
that the maximum flops (1004 GFLOPS for single precision and 501 GFLOPS for double precision), on-chip memory bandwidth (369 GB/s), and off-chip memory bandwidth (88 GB/s) are the same as that in using native OpenCL.

3.5.1.3 Data Transfer between Local Host Memory and GPU Memory

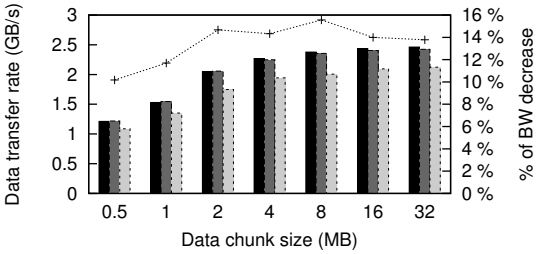
In this section, we measure the bandwidth achieved for GPU write and read operations using VOCL. The experiment is performed with different data chunk sizes. For each size, a window of 32 data chunks is issued in a nonblocking manner, followed by a flush operation to wait for their completion. The bandwidth is calculated as the total data transferred per second. A few initial "warm up" iterations are skipped from the timing loop.



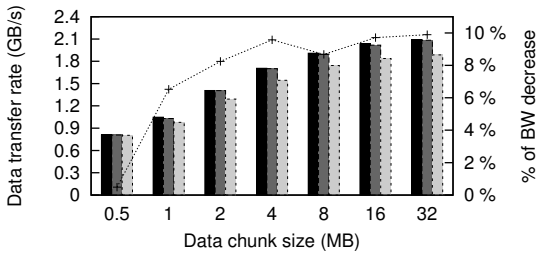
(a) Bandwidth from Host Memory to Device Memory (Local transfer is from **CPU1** to **GPU1** and remote transfer is from **CPU3** to **GPU1**.)



(c) Bandwidth from Host Memory to Device Memory (Local transfer from **CPU1** to **GPU0** and remote transfer is from **CPU3** to **GPU0**.)



(b) Bandwidth from Device Memory to Host Memory (Local transfer is from **GPU1** to **CPU1** and remote transfer is from **GPU1** to **CPU3**.)



(d) Bandwidth from Device Memory to Host Memory (Local transfer is from **GPU0** to **CPU1** and remote transfer is from **GPU0** to **CPU3**.)

Figure 3.14: Bandwidth between Host Memory and Device Memory

Figure 3.14 shows the performance of native OpenCL, VOCL when using a local GPU ("VOCL (local)"), and VOCL with a remote GPU ("VOCL (remote)"). Native OpenCL only uses the local GPU. Two scenarios are evaluated in Figure 3.7—bandwidth between CPU3 and GPU0 (Figures 3.14(c) and 3.14(d)) and between CPU3 and GPU1 (Figures 3.14(a) and 3.14(b)). In our experiments, the VOCL proxy is bound to CPU1. For native OpenCL, the application process is bound to CPU1. (Then GPU1 is used as a local GPU.)

As shown in the figures, VOCL-local has no degradation in performance as compared to native OpenCL, as expected. VOCL-remote however, has some degradation in performance because of the additional overhead of transmitting data over the network. As the data chunk size increases, the bandwidth increases for native OpenCL as well as VOCL (both local and remote), but VOCL-remote saturates at a bandwidth of 10-25% less than that of native OpenCL. Comparing the bandwidth between GPU0 and GPU1, we notice that the absolute bandwidth of native OpenCL as well as VOCL (local and remote) is smaller when using GPU0 as compared to GPU1. The reason for this behavior is that data transfer between CPU1 and GPU0 requires additional hops compared to the transfer between CPU1 and GPU1, causing some drop in performance. This lower absolute performance also results in a smaller difference between VOCL-remote (with data transfer pipeline) and native OpenCL (10% performance difference, as compared to the 25% difference when transferring from CPU1 to GPU1). The results of GPU memory read are similar.

For the remaining results, we use GPU1 because of the higher absolute performance it can achieve.

3.5.2 Evaluation with Application Kernels

In this section, we evaluate the efficiency of the VOCL framework using four application kernels: SGEMM/DGEMM for dense matrix multiplication, n-body for the motion of a group of celestial objects that interact with each other, matrix transpose, and Smith-Waterman for pairwise sequence alignment. Table 3.6 shows the computation to memory access ratios for these four kernels. The first two kernels, SGEMM/DGEMM and n-body, can be classified as compute-intensive based on their computational requirements, while the other two require more data movement.

Table 3.6: Computation and Memory Access Complexities of the Four Applications. (In matrix multiplication and matrix transpose, n is the number of rows and columns of the matrix; in n-body, n is the number of bodies; in Smith-Waterman, n is the number of letters in the input sequences.)

Application Kernels	Computation	Memory Access
SGEMM/DGEMM	$O(n^3)$	$O(n^2)$
N-body	$O(n^2)$	$O(n)$
Matrix transpose	$O(n^2)$	$O(n^2)$
Smith-Waterman	$O(n^2)$	$O(n^2)$

The difference in computational intensity of these four kernels is further illustrated in Figure 3.15, where the percentage of time spent on computation for each of these kernels is shown. As we can see in the figure, the n-body kernel spends almost 100% of its time computing. SGEMM/DGEMM spend a large fraction of time computing, and this fraction increases with increasing problem size. Matrix transpose spends a very small fraction of time computing. While Smith-Waterman spends a reasonable amount of time computing (70-80%), most of the computational kernels it launches are very small kernels which, as we will discuss later, are hard to optimize because of the large number of small message

transfers they trigger.

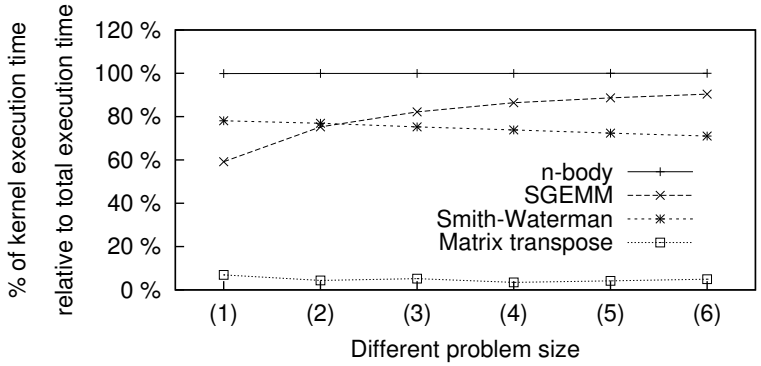
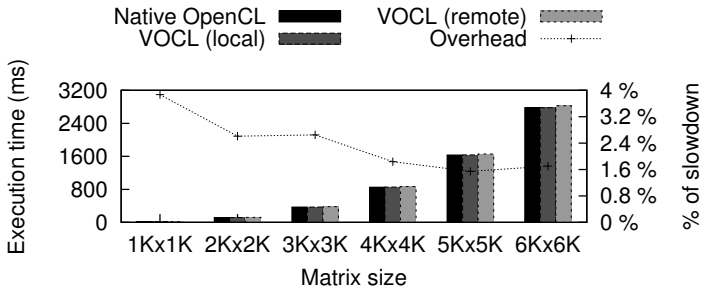


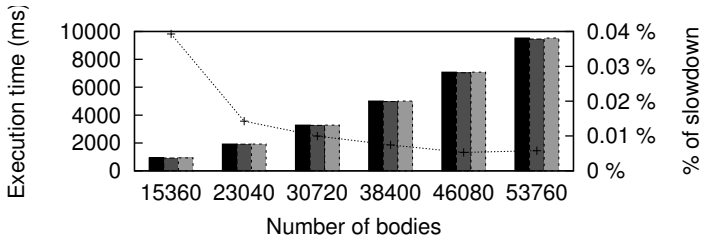
Figure 3.15: Percentage of Time Spent Executing a Kernel in the Single Precision Case (Note: Program sizes (1) – (6) indicate the following for the four application kernels. SGEMM and matrix transpose: matrix size from 1K X 1K elements to 6K X 6K elements; Smith-Waterman: sequence size from 1K letters to 6K letters; n-body: number of bodies from 15360 to 53760 with the increase of 7680.)

We evaluate the overhead of program execution time with different problem sizes. Recall that the program execution time in this experiment includes the data transfer time, kernel argument setting, and kernel execution. We run both the single-precision and double-precision implementations of all application kernels except Smith-Waterman since sequence alignment scores are usually stored as integers or single-precision floats in practice. We run multiple problem instances in a nonblocking manner to pipeline data transfer and kernel execution. After we issue all nonblocking function calls, the OpenCL function `clFinish()` is called to ensure that all computation and data transfer has completed before measuring the overall execution time.

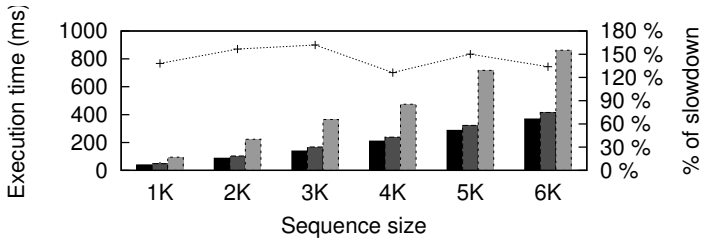
Figure 3.16 shows the performance and the overhead of the application kernels for single-precision computations. We notice that the performance of native OpenCL is almost identical to that of VOCL-local; this is expected as VOCL does very little additional



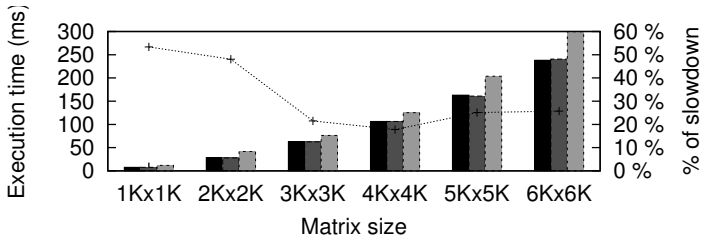
(a) SGEMM



(b) N-body



(c) Smith-Waterman



(d) Matrix Transpose

Figure 3.16: Overhead in Total Execution Time for Single-Precision Computations

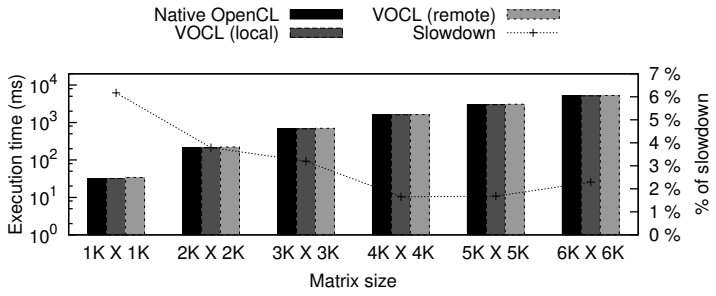
processing (e.g., translation between OpenCL and VOCL handles) in this case. For VOCL-remote, however, the performance depends on the application. For compute-intensive algorithms, the overhead of VOCL is very small; 1-4% for SGEMM and nearly zero for n-body. This is because for these applications the total execution time is dominated by the kernel execution. For SGEMM, we further notice that the overhead decreases with increasing problem size. This is because the computation time for SGEMM increases as $O(N^3)$ while the amount of data that needs to be transferred only increases as $O(N^2)$; thus, the computation time accounts for a larger percentage of the overall execution time for larger problem sizes as shown in Figure 3.15.

For algorithms requiring more data movement between host memory and device memory, the overhead of VOCL is higher. For matrix transpose, for example, this is between 20-55%, which is expected because it spends a large fraction of its execution time in data transfer (based on Figure 3.15, matrix transpose spends only 7% of its time in computing). With VOCL-remote, such data transfer causes significant overhead. For Smith-Waterman, the overhead is much higher and close to 150%. This is because of two reasons. First, since the VOCL proxy is a multi-threaded process, the MPI implementation has to be initialized to support multiple threads. It is well known in the MPI literature that multi-threaded MPI implementations can add significant overhead in performance, especially for small messages [12, 13, 26, 30]. Second, Smith-Waterman relies on a large number of kernel launches for a given amount of data [87]. For a 1K sequence alignment, for example, more than 2000 kernels are launched causing a large number of small messages to be issued, which, as mentioned above, cause a lot of performance overhead. We verified this by artificially initializing the MPI library in single-threaded mode and noticed that the

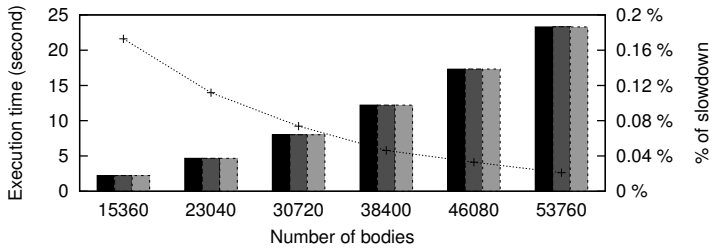
overhead with VOCL comes down to around 35% by doing so.[2] Since each kernel launch needs a few small messages to be transferred between the local node and remote node, we have no way to reduce the number of messages to be transferred in the VOCL framework if the number of kernel launches keeps the same. But for applications using multiple kernel launches for data communication across different work-groups, we can use the *GPU synchronization* strategy for such data communication as proposed in Chapter 4. With the GPU synchronization, when inter-work-group data communication is needed, we can just call the barrier function instead of terminating the kernel execution and re-launching the kernel. In this way, the kernel is launched only once and the overhead caused by transferring large number of small messages can be avoided.

Figure 3.17 shows the performance and the overhead of the application kernels for double precision computations. The observed trends for double precision are nearly identical to the single-precision cases. This is because the amount of data transferred for double-precision computations is double that of single-precision computations; and on the NVIDIA Tesla M2070 GPU, the double-precision computations are about twice as slow as single-precision computations. Thus, both the computation time and the data transfer time double and result in no relative difference. On other architectures such as the older generations of NVIDIA GPU cards where the double-precision computations were much slower than single-precision computations, we expect this balance to change and the relative overhead of VOCL to reduce since percentage of time spent executing the kernel will be higher than that on the Tesla M2070.

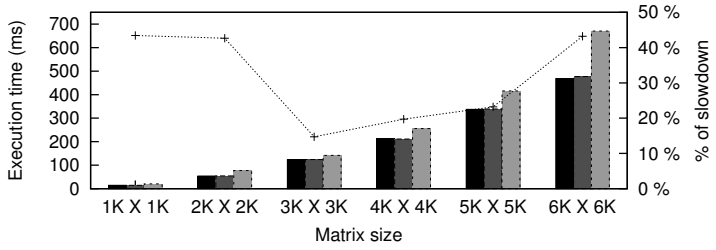
[2]Note that, in this case, the VOCL proxy can accept only one connection request each time it is started. After an application finishes its execution and disconnects the communication channel, we would need to restart the proxy process for the next run; a process that is unusable in practice. We only tried this approach to understand the overhead of using a multi-threaded vs. single-threaded MPI implementations.



(a) DGEMM



(b) N-body



(c) Matrix Transpose

Figure 3.17: Overhead in Total Execution Time for Double-Precision Computations

3.5.3 Multiple Virtual GPUs

OpenCL allows applications to query for the available GPUs and distribute their problem instances on them. Thus, with native OpenCL, an application can query for all the local GPUs and utilize them. With VOCL, on the other hand, the application would have access to all the GPUs in the entire system; thus, when the application executes the resource query function, it would look like it has a very large number of GPUs.

In this section, we perform experiments with a setup that has 16 compute nodes, each with 2 local GPUs; thus, with VOCL, it would appear to the applications running in this environment, that they have 32 local (virtual) GPUs and the application can distribute its work on 32 GPUs instead of the 2 GPUs that it would use with native OpenCL on a per-node basis. Figure 3.18 shows the total speedup achieved with 1, 2, 4, 8, 16, and 32 virtual GPUs utilized. With one and two GPUs, only local GPUs are used. In the other cases, two of the GPUs are local, and the remaining are remote.

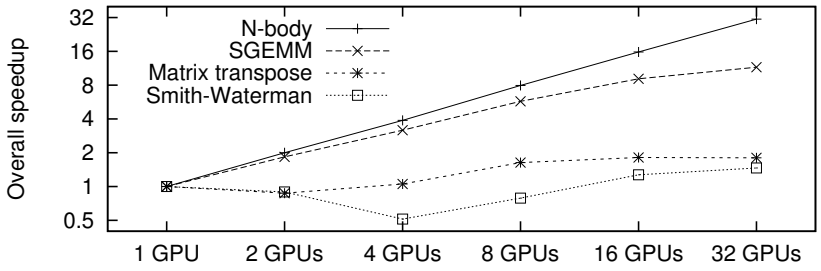


Figure 3.18: Performance Improvement with Multiple Virtual GPUs Utilized (single precision)

As shown in the figure, for compute-intensive applications such as SGEMM and n-body, the speedup can be significant; for instance, with 32 GPUs, the overall speedup of n-body is about 31-fold. For SGEMM, the overall speedup is 11.5-fold (some scalability is lost because of the serialization of the data transfer through a single network link). For applications such as matrix transpose and Smith-Waterman that require more data movement, on the other hand, there is almost no performance improvement; in fact, the performance degrades in some cases. For the matrix transpose, the reason for this behavior is that most of the program execution time is for data transfer between host memory and device memory. As data transfer is serialized to different GPUs, program execution still

takes approximately the same amount of time as with the single GPU case. As for Smith-Waterman, as shown in the previous section, using remote GPUs can cause significant overhead. When part of the instances are computed on remote GPUs, it is possible that the overall performance is worse than the single GPU case.

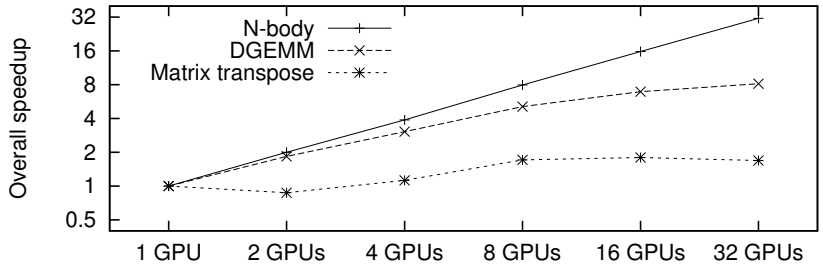


Figure 3.19: Performance Improvement with Multiple Virtual GPUs Utilized (double precision)

Figure 3.19 illustrates a similar experiment but for double-precision computations. Again, we notice almost identical trends as single-precision computations because there is no relative difference in data transfer time and computation time, as explained in Section 3.5.2.

3.6 Summary

GPUs have been widely adopted to accelerate general-purpose applications. However, the current programming models, such as CUDA and OpenCL, only support the use of GPUs in local compute nodes. In this work, we propose the VOCL framework to support the transparent virtualization of GPUs, which in turn allows applications to use both local and remote GPUs in a system as if they were installed locally. In GPU computing, data transfer

between host memory and device memory is a bottleneck, which can slow down the program execution by 2 to 50 times. When remote GPUs are used, the data transfer overhead can become even higher. As such, we carefully optimized the data transfer between local host memory and remote device memory based on a performance model. For evaluation, we studied the overhead of the VOCL framework using various microbenchmarks as well as four application kernels with various properties of compute and data transfer between host memory and device memory. Our experimental results indicate that the VOCL framework can support the transparent virtualization of GPUs in a system at a reasonable cost, particularly for compute-intensive applications.

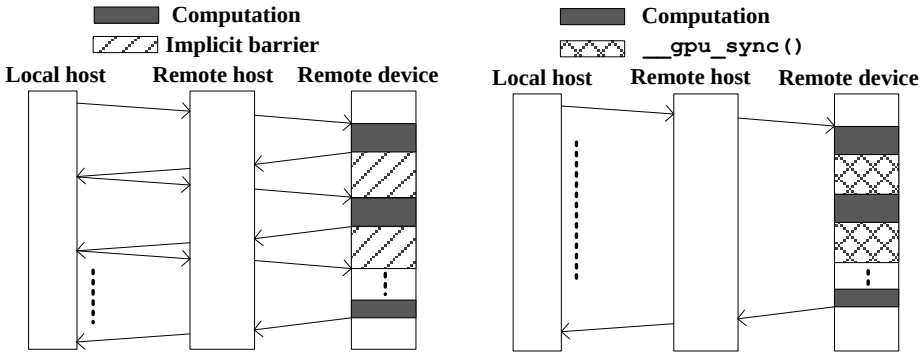
Chapter 4

GPU Synchronization

4.1 Overview

In general, the GPU architecture only maps well to data- or task- parallel applications that have minimal or no data communication across different blocks on a GPU card. One of the reasons can be the lack of explicit hardware and software support for such data communication. On a GPU card, data communication across different blocks occurs via the global memory and barrier synchronization is needed. Currently, such barrier synchronization is implicitly achieved by terminating current kernel's execution and re-launching the kernel, which is a costly operation. This implicit synchronization approach will be even more expensive when a remote GPU is used since each kernel launch becomes a two-step scenario. Specifically, when a kernel is launched, the local node sends a message to the remote node, which then launches the kernel to the GPU as shown in Figure 4.1(a). This two-step scenario can cause large overhead to applications whose execution requires data communication across different blocks on remote GPUs, see Figure 3.16(c).

In this chapter, we propose the barrier synchronization required for inter-block data communication on GPUs. The proposed barrier synchronization approaches can both ease the task of programming GPUs and achieve better performance than the commonly used approach of multiple kernel launches, particularly in using remote GPUs as presented in the previous chapter. As shown in Figure 4.1(b), with the GPU synchronization strategy, the kernel is launched only once and the needed barrier is achieved by calling our barrier function `__gpu_sync()`.



(a) CPU synchronization (b) GPU synchronization

Figure 4.1: Barrier Synchronization Scenarios in Remote GPU Utilization

We begin by showing the time components in kernel execution and the high percentage of time that inter-block data communication consumes in the total execution time for some applications. Then we describe the related work for inter-thread data communication. With the above background information, we present our proposed GPU synchronization approaches—*GPU lock-based synchronization* and *GPU lock-free synchronization* and the time consumed in each of them. For performance evaluation, we integrate our GPU synchronization approaches in a microbenchmark and three well-known algorithms—fast Fourier transformation (FFT), Smith-Waterman, and bitonic sort. Based on the parallel

implementations, we show the performance improvement brought by the GPU synchronization and characterize the time consumption of each operation in the GPU synchronization in a fine-grained way. In addition, we explain the possible errors in using the GPU synchronization and the corresponding solutions for that. Finally, to reduce the overhead of the inter-block data communication in using remote GPUs based on VOCL, we extend the GPU synchronization strategy to the OpenCL programming model and demonstrate the performance improvement brought by the GPU synchronization.

4.2 Time Composition of Kernel Execution

In general, a kernel's execution time on a GPU consists of three components—*kernel launch time*, *computation time*, and *synchronization time*, which can be represented as

$$T = \sum_{i=1}^{M} \left(t_O^{(i)} + t_C^{(i)} + t_S^{(i)} \right) \tag{4.1}$$

where M is the number of kernel launches, $t_O^{(i)}$ is the kernel launch time, $t_C^{(i)}$ is the computation time, and $t_S^{(i)}$ is the synchronization time for the i^{th} kernel execution, as shown in Figure 4.2. Each of the three time components is impacted by a few factors. For instance, the kernel launch time depends on the data transfer bandwidth from the host to the device as well as the size of kernel code and kernel arguments. For the computation time, it is affected by memory access methods, kernel configuration (number of threads per block and number of blocks per grid), etc. Similarly, the synchronization time will be different for different synchronization approaches.

Figure 4.3 shows the pseudo-code of implementing barrier synchronization via kernel launches, where Figure 4.3(a) is the function call of *CPU explicit synchronization*

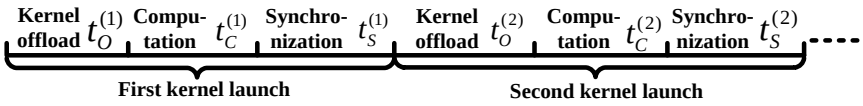


Figure 4.2: Total Kernel Execution Time Composition

```
for() {
  __kernel_func<<<grid, block>>>();
  cudaThreadSynchronize();
}
```

(a) CPU explicit synchronization

```
for() {
  __kernel_func<<<grid, block>>>();
  //without cudaThreadSynchronize()
}
```

(b) CPU implicit synchronization

Figure 4.3: CPU Explicit/Implicit Synchronization Function Call

and Figure 4.3(b) is for *CPU implicit synchronization*. As we can see, in the CPU explicit synchronization, the kernel function `__kernel_func()` is followed by the function `cudaThreadSynchronize()`, which will *not* return until all prior operations on the device are completed. As a result, the three operations—kernel launch, computation, and synchronization are executed sequentially with the CPU explicit synchronization. In contrast, the CPU implicit synchronization does *not* call `cudaThreadSynchronize()`. Since kernel launch is asynchronous, if there are multiple kernel launches, kernel launch time can be overlapped by previous kernels' computation time and synchronization time. So, in the CPU implicit synchronization approach, except for the first kernel, kernel launch time is overlapped by the previous kernel's execution, and the execution time of multiple kernels can be represented as

$$T = t_O^{(1)} + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{CIS}^{(i)} \right) \tag{4.2}$$

where, M is the number of kernel launches, $t_O^{(1)}$ is the launch time for the first kernel, $t_C^{(i)}$ and $t_{CIS}^{(i)}$ are the computation time and synchronization time for the i^{th} kernel, respectively.

With respect to the *GPU synchronization*, Figure 4.4 shows the pseudo-code of how functions are called. In this approach, a kernel is launched only once. When barrier synchronization is needed, we just call a barrier function `__gpu_sync()` instead of re-launching the kernel. In Figure 4.4, the function `__device_func()` implements the same functionality as the kernel function `__kernel_func()` in Figure 4.3, but it is a device function instead of a global one, so it is called on the device rather than on the host. With the GPU synchronization, kernel execution time can be expressed as

$$T = t_O + \sum_{i=1}^{M} \left(t_C^{(i)} + t_{GS}^{(i)} \right) \tag{4.3}$$

where, M is the number of barriers needed for the kernel's execution, t_O is the kernel launch time, $t_C^{(i)}$ and $t_{GS}^{(i)}$ are the computation time and synchronization time for the i^{th} loop, respectively.

```
__global__ void __kernel_func1()
{
    for () {
        __device_func();
        __gpu_sync();
    }
}
```

Figure 4.4: GPU Synchronization Function Call

From Equations (4.1), (4.2), and (4.3), we can accelerate an algorithm by decreasing any of the three time components. With properties of the kernel launch time considered,[1] we ignore the kernel launch time in the following discussion. If the synchronization time is reduced, according to the Amdahl's Law, the maximum kernel execution speedup is

[1]Three properties are considered: first, kernel launch time can be combined with the synchronization time in the CPU explicit synchronization; second, it can be overlapped in CPU implicit synchronization; and third, kernel is launched only once in the GPU synchronization.

constrained by

$$
\begin{aligned}
S_T &= \frac{T}{t_C + (T - t_C)/S_S} \\
&= \frac{1}{\left(\frac{t_C}{T}\right) + \left(1 - \frac{t_C}{T}\right)/S_S} \\
&= \frac{1}{\rho + (1-\rho)/S_S}
\end{aligned}
\tag{4.4}
$$

where S_T is the kernel execution speedup gained by reducing the synchronization time, $\rho = \frac{t_C}{T}$ is the percentage of the computation time t_C in the total kernel execution time T, $t_S = T - t_C$ is the synchronization time of the CPU implicit synchronization, which is our baseline for performance evaluation later. S_S is the synchronization speedup. Similarly, if only computation is accelerated, the maximum overall speedup is constrained by

$$
S_T = \frac{1}{\rho/S_C + (1-\rho)} \tag{4.5}
$$

where S_C is the computation speedup. In Equation (4.4), the smaller the ρ is, the more speedup can be achieved with a fixed S_S; while in Equation (4.5), the larger the ρ is, the more speedup can be obtained with a fixed S_C. In practice, different algorithms have different ρ values.

4.3 Time Profile for Barrier Synchronization

Many applications need inter-block data communication when they are executed on a GPU. In this section, we profile the execution of the aforementioned three algorithms—FFT, Smith-Waterman, and bitonic sort on the GTX 280 GPU with the current state-of-the-art barrier synchronization, i.e., multiple kernel launches. As we can see in Table 4.1, inter-block data communication can occupy more than 50% of the kernel execution time

in some applications. According to the kernel execution time model in Section 4.2, if we only optimize the computation for algorithms such as the bitonic sort, the maximum speedup will be less than 2 times, thus it is of great importance to improve the efficacy of inter-block data communication on GPUs.

Table 4.1: Percentage of Time Spent on Inter-Block Communication on the GTX 280

Algorithms	FFT	Smith-Waterman	Bitonic sort
% of time spent on inter-block communication	17.8%	49.2%	59.6%

4.4 Existing Work for Inter-Thread Data Communication

Many types of software barriers have been designed for shared-memory environments [6, 17, 35, 40, 52], but none of them can be directly applied to GPU environments. This is because multiple CUDA thread blocks can be scheduled to be executed on a single SM and the CUDA blocks do not yield to the execution. That is, blocks run to completion once spawned by the CUDA thread scheduler. This may result in deadlocks, and thus, cannot be resolved in the same way as that in traditional CPU processing environments, where one can yield the waiting process to execute other processes. One way of addressing this issue is ensuring a one-to-one mapping between the streaming multiprocessors (SMs) and the thread blocks.

Cederman et al. [20] implemented a dynamic load-balancing method on the GPU that is based on the lock-free synchronization method used on traditional multi-core processors. However, this scheme controls task assignment instead of addressing inter-block communication. In addition, we note that lock-free synchronization generally performs

worse than lock-based methods on traditional multi-core processors, but its performance is better than that of the lock-based method on the GTX 280 GPU in our work.

The work of Stuart et al. [76] focuses on data communication between multiple GPUs, i.e., inter-GPU communication. Though their approach can be used for inter-block communication across different SMs on the same GPU, the performance is projected to be quite poor because data needs to be moved to the CPU host memory first and then transferred back to the device memory, which is unnecessary for data communication on a single GPU card.

The most closely related work to ours is that of Volkov et al. [82]. Volkov et al. proposed a global software synchronization method that does not use atomic operations to accelerate dense linear-algebra constructs. However, as they noted, their synchronization method has not been implemented into any real application to test the performance improvement. Furthermore, their proposed synchronization cannot guarantee that previous accesses to all levels of the memory hierarchy have completed. Finally, Volkov et al. used only one thread to check all *arrival* variables, hence serializing this portion of inter-block synchronization and adversely affecting its performance. In contrast, our proposed GPU synchronization approaches guarantee the completion of memory accesses with the existing memory-access model in CUDA. This is due to the memory fence function `__threadfence()` introduced in CUDA 2.2, which can guarantee that all writes to global memory are visible to other threads, so the correctness of reads after the barrier function can be guaranteed. In addition, we integrate our GPU synchronization approaches in a microbenchmark and three well-known algorithms for performance evaluation. Finally, we use multiple threads in a block to check all the *arrival* variables, which can be

executed in parallel, thus achieving a good performance.

4.5 Proposed GPU Barrier Synchronization

In the following discussion, we will present two alternative GPU synchronization designs: *GPU lock-based synchronization* and *GPU lock-free synchronization*. The lock-based design makes use of mutually exclusive (mutex) variables and CUDA atomic operations; while the lock-free design uses a decentralized approach that avoids the use of the CUDA atomic operations.

4.5.1 GPU Lock-Based Synchronization

The basic idea of GPU lock-based synchronization [87] is to use a global mutex variable to count the number of thread blocks that reach the synchronization point. As shown in Figure 4.6, in the barrier function `__gpu_sync()`, after a block completes its computation, one of its threads (i.e., the *leading thread*) will atomically add one to `g_mutex`. The leading thread will then repeatedly compare `g_mutex` to a target value `goalVal`. If `g_mutex` is equal to `goalVal`, the synchronization is completed and each thread block can proceed with its next stage of computation, as shown in Figure 4.5. In our design, `goalVal` is set to the number of blocks N in the kernel when the barrier function is first called. The value of `goalVal` is then incremented by N each time when the barrier function is successively called. This design is more efficient than keeping `goalVal` constant and resetting `g_mutex` after each barrier because the former saves the number of instructions and avoids conditional branches.

In the GPU lock-based synchronization, the execution time of the barrier function

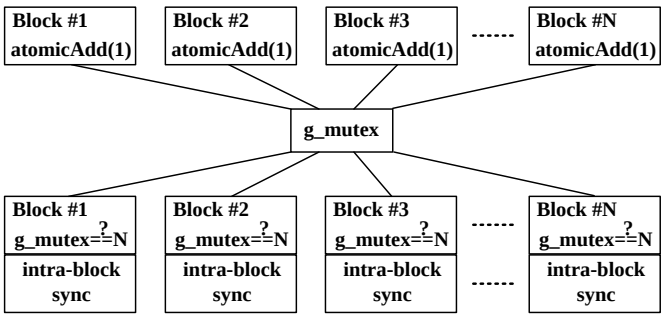


Figure 4.5: Operations in GPU Lock-Based Synchronization

```
1  //the mutex variable
2  __device__ volatile int g_mutex;
3
4  //GPU lock-based synchronization function
5  __device__ void __gpu_sync(int goalVal)
6  {
7      //thread ID in a block
8      int tid_in_block = threadIdx.x * blockDim.y + threadIdx.y;
9
10     // only thread 0 is used for synchronization
11     if (tid_in_block == 0)   {
12         atomicAdd((int *)&g_mutex, 1);
13
14         //only when all blocks add 1 to g_mutex will g_mutex equal to goalVal
15         while(g_mutex != goalVal)   {
16             //Do nothing here
17         }
18     }
19     __syncthreads();
20 }
```

Figure 4.6: Pseudo Code of GPU Lock-Based Synchronization

`__gpu_sync()` consists of three parts—atomic addition, checking of a mutex variable `g_mutex`, and synchronization of threads within a block via `__syncthreads()`. The atomic addition can only be executed sequentially by different blocks, while the `g_mutex` checking and intra-block synchronization can be executed in parallel. Assume there are N blocks in the kernel, the intra-block synchronization time is t_s, time of each atomic addition and `g_mutex` checking is t_a and t_c, respectively, if all blocks finish their computation

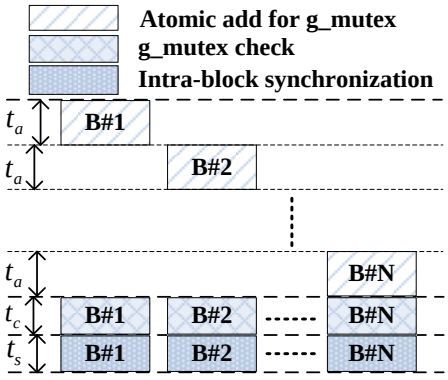


Figure 4.7: Time Composition of GPU Lock-Based Synchronization

at the same time as shown in Figure 4.7, then the time to execute `__gpu_sync()` is

$$t_{GBS} = N \cdot t_a + t_c + t_s \tag{4.6}$$

where N is the number of blocks in the kernel. From Equation (4.6), the cost of GPU lock-based synchronization increases linearly with N.

4.5.2 GPU Lock-Free Synchronization

In the GPU lock-based synchronization, the mutex variable `g_mutex` is added with the atomic function `atomicAdd()`. This means the addition of `g_mutex` can only be executed sequentially even though these operations are performed in different blocks. In this section, we propose a lock-free synchronization approach that does not use atomic operations. The basic idea of this approach is to assign a synchronization variable to each thread block so that each block can record its synchronization status independently without competing for a single global mutex variable, as shown in Figure 4.8.

As shown in Figure 4.9, our lock-free synchronization approach uses two arrays `Arrayin` and `Arrayout` to coordinate the synchronization requests from various blocks. In these

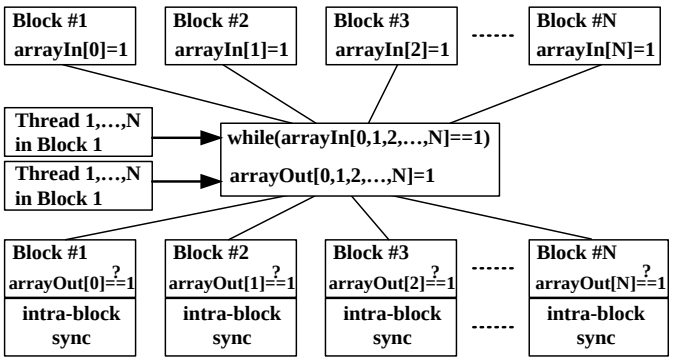


Figure 4.8: Operations in GPU Lock-Free Synchronization

two arrays, each element is mapped to a thread block in the kernel, i.e., element i is mapped to thread block i. The algorithm is outlined into three steps as follows:

1. When block i is ready for communication, its leading thread (thread 0) sets element i in `Arrayin` to the goal value `goalVal`. The leading thread in block i then busy-waits on element i of `Arrayout` to be set to `goalVal`.

2. The first N threads in block 1 repeatedly check if all elements in `Arrayin` are equal to `goalVal`, with thread i checking the i^{th} element in `Arrayin`. After all elements in `Arrayin` are set to `goalVal`, each checking thread then sets the corresponding element in `Arrayout` to `goalVal`. Note that the intra-block barrier function `__syncthreads()` is called by each checking thread before updating elements of `Arrayout`.

3. A block will continue its execution once its leading thread sees the corresponding element in `Arrayout` is set to `goalVal`.

It is worth noting that in the step 2) above, rather than having one thread check all elements of `Arrayin` *in serial* as in [82], we use N threads to check the elements of

`Arrayin` *in parallel*. This design choice reduces synchronization overhead considerably, as shown in Section 4.7. Note also that `goalVal` is incremented each time when the function `__gpu_sync()` is called, similar to the implementation of the GPU lock-based synchronization.

```
1  //GPU lock-free synchronization function
2  __device__ void __gpu_sync(int goalVal, volatile int *Arrayin, volatile int *Arrayout)
3  {
4      // thread ID in a block
5      int tid_in_blk = threadIdx.x * blockDim.y + threadIdx.y;
6      int nBlockNum = gridDim.x * gridDim.y;
7      int bid =  blockIdx.x * gridDim.y + blockIdx.y;
8
9      // only thread 0 is used for synchronization
10     if (tid_in_blk == 0)    {
11         Arrayin[bid] = goalVal;
12     }
13
14     if (bid == 1)    {
15         if (tid_in_blk < nBlockNum)    {
16             while (Arrayin[tid_in_blk] != goalVal){
17                 //Do nothing here
18             }
19         }
20         __syncthreads();
21
22         if (tid_in_blk < nBlockNum)    {
23             Arrayout[tid_in_blk] = goalVal;
24         }
25     }
26
27     if (tid_in_blk == 0)    {
28         while (Arrayout[bid] != goalVal)    {
29             //Do nothing here
30         }
31     }
32     __syncthreads();
33 }
```

Figure 4.9: Pseudo Code of GPU Lock-Free Synchronization

From Figure 4.9, there is no atomic operation in the GPU lock-free synchronization. All the operations can be executed in parallel. Synchronization of different thread blocks is controlled by threads in a single block, which can be synchronized efficiently by calling the barrier function `__syncthreads()`. From Figure 4.10, the execution time of

`__gpu_sync()` is composed of six parts and calculated as

$$t_{GFS} = t_{SI} + t_{CI} + 2t_s + t_{SO} + t_{CO} \tag{4.7}$$

where, t_{SI} is the time for setting an element in `Arrayin`, t_{CI} is the time to check an element in `Arrayin`, t_s is the intra-block synchronization time, t_{SO} and t_{CO} are the time for setting and checking an element in `Arrayout`, respectively. From Equation (4.7), execution time of `__gpu_sync()` is unrelated to the number of blocks in a kernel.[2]

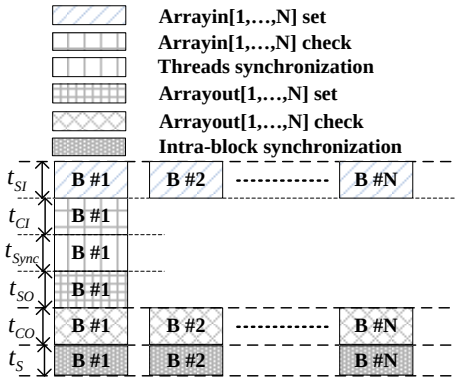


Figure 4.10: Time Composition of GPU Lock-Free Synchronization

4.6 Analysis of Inter-Block Data Communication Correctness

According to the relaxed memory consistency model on the GPU, there is no guarantee that previous writes to the global memory are visible to all SMs after the global barrier function. As such, even though the execution of different blocks are synchronized,

[2]Since there are at most 30 blocks that can be set on a GTX 280, threads that check `Arrayin` are in the same warp, which are executed in parallel. When there are more than 32 blocks in the kernel, threads in more than one warp are needed for checking `Arrayin`, and different warps are executed serially on an SM.

the correctness of inter-block data communication cannot be guaranteed. To remedy this problem, CUDA introduced the memory fence function `__threadfence()`, which will block the calling thread until its previous writes to the global memory are visible to all threads on the GPU card. By inserting the memory fence function to the global barrier function `__gpu_sync()`, we can guarantee the correctness of inter-block data communication. For example, consider a thread that needs to read some data written by other threads after the barrier, since the thread has passed the barrier, all threads in the kernel have passed memory fence function `__threadfence()`, too (otherwise, the considered thread cannot pass the barrier.). As such, writes to the global memory of all threads have been visible to all others in the kernel. Then what the thread reads from the global memory are those written by other threads, and inter-block data communication correctness is guaranteed.

It is expected that overhead will be introduced by integrating the memory fence function `__threadfence()` into the global barrier function as demonstrated later in Section 4.7.5.

4.7 Performance Evaluation

4.7.1 Overview

To evaluate the performance of our proposed GPU synchronization approaches, we implement them in a microbenchmark as well as three algorithms—FFT, Smith-Waterman, and bitonic sort. In the microbenchmark, results of all synchronization approaches are shown. Since the performance of the CPU explicit synchronization is far worse than that

of the CPU implicit synchronization, we show performance of only the CPU implicit synchronization and the GPU synchronization in the three real algorithms. Specifically, their performance is evaluated in three aspects: 1) the impact on kernel execution time and its variation against the number of blocks in the kernel; 2) the percentage of time spent in computation versus synchronization; and 3) the cost of guaranteeing inter-block data communication correctness via the memory fence function.

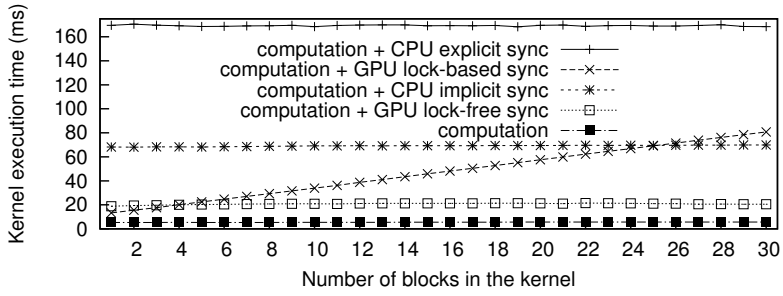
Our experiments are performed on the NVIDIA GeForce GTX 280 and the NVIDIA Tesla Fermi C2050 GPU cards. The GTX 280 has 30 SMs and 240 processing cores with the clock speed 1296MHz. Each SM contains 16K registers and 16KB shared memory. For the entire GPU card, there is 1GB GDDR3 global memory with 141.7GB/second of memory bandwidth. As for the Tesla Fermi C2050, it has 14 SMs and 448 scalar-processors (SPs) with the clock speed 1.15GHz. There are 32K registers and 64KB shared memory on each SM and the total global memory size is 3GB. On the host machine, there are two AMD Magny-Cours processors (Each has eight cores.) with the clock speed 800MHz and 32GB host memory is equipped on the machine. The operating system on the host machine is the 64-bit CentOS 5.5. The NVIDIA CUDA 4.0 is used for all the program execution. In our experiments, each result is the average of three runs.

4.7.2 Synchronization Time Verification via a Microbenchmark

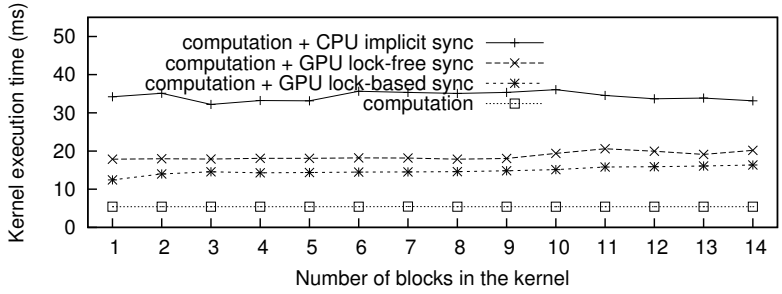
In this experiment, we write a microbenchmark to verify the synchronization time for each synchronization method. The microbenchmark is to compute the mean of two floats for 10,000 times. If the CPU synchronization is used, each kernel calculates the mean once and the kernel is launched 10,000 times. With the GPU synchronization, the kernel

calculates the mean for 10,000 times using a 10,000-iteration `for` loop with the GPU barrier function called in each loop. In the microbenchmark, each thread computes one mean value. That is, the more blocks and threads are set, the more elements are computed, i.e., weak scaling. As such, the computation time should be approximately constant with different number of blocks and threads configured in the kernel.

Figure 4.11 shows the kernel execution time for different synchronization methods: Figure 4.11(a) is on the GTX 280 and Figure 4.11(b) is on the Tesla Fermi C2050. From this figure, we have the following observations:



(a) GTX 280



(b) Tesla C2050 (Fermi) (Note: kernel execution time with the CPU explicit synchronization is 247.35ms.)

Figure 4.11: Execution Time of the Microbenchmark.

On the GTX 280, 1) the CPU explicit synchronization takes 170 ms, which is 2.46

times that of the CPU implicit synchronization, about 69 ms. In the CPU explicit synchronization, kernel launch, computation, and synchronization are executed sequentially; while in the CPU implicit synchronization, kernel launch is overlapped by computation of its previous kernels. 2) Even for the CPU implicit synchronization, the synchronization overhead is still quite high. As shown in the figure, the computation time is only about 5ms, while the time for the CPU implicit synchronization is about 60ms, which is 12 times the computation time. 3) For the GPU lock-based synchronization, the synchronization time is linear to the number of blocks in the kernel. The more blocks are configured in the kernel, the more synchronization time is needed, which matches very well to Equation (4.6) in Section 4.5.1. Compared to the CPU implicit synchronization, if the number of blocks is less than 24, the GPU lock-based synchronization takes less time; otherwise, it needs more time. The reason is that, as analyzed in Section 4.5.1, more blocks means more atomic add operations are executed for each barrier synchronization and atomic add can only be executed sequentially. 4) As for the GPU lock-free synchronization, since there are no atomic operations used, all operations can be executed in parallel, which makes its synchronization time unrelated to the number of blocks in a kernel. As such, the synchronization time is relatively constant. Furthermore, the synchronization time is much less (for more than 3 blocks configured in the kernel) than that of all other synchronization methods.

On the Tesla Fermi C2050, 1) similar to the GTX 280, the CPU explicit synchronization takes 247.35 ms, which is 212 ms more than that of the CPU implicit synchronization. The reason is the same with the GTX 280, the CPU explicit synchronization serializes the different stages of a kernel's execution. 2) For the CPU implicit synchronization, it is more

expensive than both GPU synchronization approaches. Specifically, with 14 blocks configured in the kernel, synchronization time of the CPU implicit synchronization is 27.7ms, and that of the GPU lock-based and lock-free synchronization approaches are 10.9ms and 14.73ms, respectively. 3) For the lock-free synchronization approach, in contrast to the performance on the GTX 280, its synchronization time slightly increases with more blocks configured in the kernel. One reason can be that more data accesses are needed with more blocks in the kernel, which increases the kernel execution time. For the GPU lock-based synchronization, its synchronization increases with more blocks configured in the kernel, too. Compared to the lock-free synchronization approach, the rate of increase for the lock-based approach is even higher. Why? In addition to more memory accesses with more blocks in the kernel, the GPU lock-based synchronization needs the atomic add operations, which can only be executed sequentially, even in different blocks.

Comparing the performance on the GTX 280 and the Tesla Fermi C2050, we observe the following. First, the CPU explicit synchronization on the Tesla Fermi C2050 is more expensive than that on the GTX 280, but the performance of the CPU implicit synchronization is opposite. Second, as for the GPU synchronization on the GTX 280, the GPU lock-based synchronization is slower than the GPU lock-free synchronization; while on the Tesla Fermi C2050, the GPU lock-free synchronization is slower than the GPU lock-based approach. The reason for the performance difference is that the atomic add is much faster on the Tesla Fermi C2050 than on the GTX 280 as shown later in Section 4.7.3.

From the microbenchmark results, the CPU explicit synchronization needs 2.4-fold and 7.1-fold more time on the GTX 280 and the Tesla Fermi C2050, respectively, than the CPU implicit synchronization. So, in the following sections, we will not use it any more,

and only the CPU implicit synchronization and the two GPU synchronization approaches are compared and analyzed.

4.7.3 Fine-Grained Analysis of GPU Barrier Synchronization

In this section, we analyze the synchronization overhead by partitioning it into the time consumed by each operation within the synchronization [29]. As an example, we analyze the GPU lock-based synchronization, which contains a superset of all the operations that are used in the GPU lock-free synchronization. For the barrier implementation in Section 4.5.1 with the memory fence function used, there are four types of operations in the GPU lock-based synchronization, and its synchronization time can be expressed as

$$T_S = t_a + t_c + t_s + t_f \tag{4.8}$$

where t_a is the overhead of atomic add, t_c is the mutex variable checking time, t_s is the time consumed by `__syncthreads()`, and t_f is the execution time of the `__threadfence()`. Since the execution times of these component operations cannot be measured directly on the GPU, we use an indirect approach to infer the times. Specifically, we measure the kernel execution time in different scenarios and then calculate the execution time of each of the above operations. Based on the kernel execution time model in Section 4.2, a kernel's execution time can be expressed as

$$T = t_O + t_{Com} + t_S \tag{4.9}$$

where t_O is the kernel launch time, t_{Com} is the computation time, and t_S is the synchronization time. By combining Equations (4.8) and (4.9), the kernel execution time can be

represented as

$$T = t_O + t_{Com} + t_a + t_c + t_s + t_f \tag{4.10}$$

From Equation (4.10), we can calculate the overhead of a particular operation, e.g., `__threadfence()` by measuring the kernel execution time both with and without `__threadfence()` and taking the time difference as its overhead.

With the above indirect approach, we use the same microbenchmark in the previous section and measure the kernel execution times in the following scenarios:

1. Sum of the kernel launch and computation time, i.e., $t_1 = t_O + t_{Com}$, which is the kernel execution time of a GPU synchronization implementation but without the barrier function `__gpu_sync()` called.

2. Kernel execution time with one `atomicAdd` called in each block, i.e., $t_2 = t_a$.

3. Sum of the time for kernel launch, computation, and `__syncthreads()`, i.e., $t_3 = t_O + t_{Com} + t_s$.

4. Kernel execution time with the GPU lock-based synchronization, so $t_4 = t_O + t_{Com} + t_s + t_a + t_c$.

5. Kernel execution time of the GPU lock-based synchronization with `__threadfence()`, thus $t_5 = t_O + t_{Com} + t_s + t_a + t_c + t_f$.

Figure 4.12 shows the measured execution times of t_1 to t_5 noted above, where Figure 4.12(a) shows the results on the GTX 280 and Figure 4.12(b) is for the Tesla Fermi C2050. With these times, execution time of the four types of operations in the GPU lock-based synchronization can be calculated as:

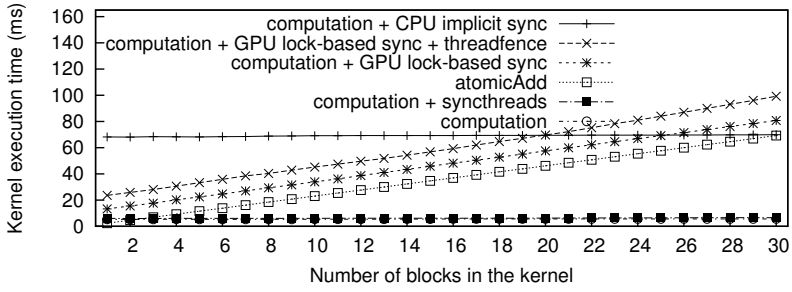
1. Time for executing the atomic add is t_2, i.e., $t_a = t_2$;

2. Time of the mutex variable checking is $t_c = t_4 - t_3 - t_2$;

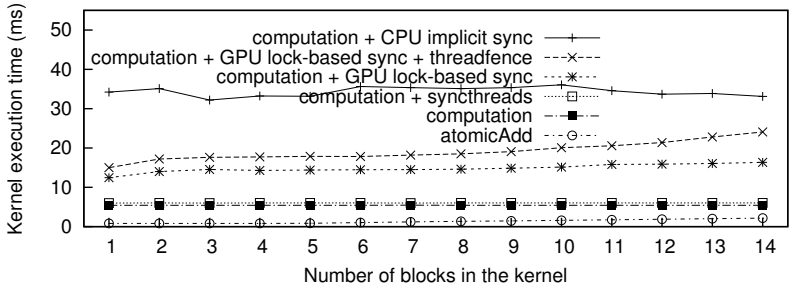
3. Time consumption of `__syncthreads()` is $t_s = t_3 - t_1$;

4. Overhead of the function `__threadfence()` is $t_f = t_5 - t_4$.

Thus, for 10,000 iterations of execution on the GTX 280, $t_s = 0.718$, $t_a = 2.307 \times n$, $t_c = 4.937$, and $t_f = 0.280 \times n + 8.563$, where n is the number of blocks in the kernel, and the units are in *milliseconds*. On the Tesla Fermi C2050, $t_s = 0.589$, $t_a = 0.106 \times n$, $t_c = 7.613$, and $t_f = 0.396 \times n + 2.294$.



(a) GTX 280



(b) Tesla C2050 (Fermi)

Figure 4.12: Profile of GPU Lock-Based Synchronization via a Microbenchmark

These results show the following. First, the intra-block synchronization function `__syncthreads()`

consumes very little time. With 10,000 iterations of execution, the execution time on the GTX 280 is only 0.718 ms, which is about 93 clock cycles per call and is a constant value unrelated to the number of threads that call it. Similarly, the execution time of `__syncthreads()` on the Tesla Fermi C2050 is 0.589 ms, corresponding to 68 clock cycles per call. Second, similar to `__syncthreads()`, the execution time of the mutex variable checking is a constant value, which is unrelated to the number of blocks in the kernel. This operation takes 4.937 ms and 7.631 ms on the GTX 280 and Tesla Fermi C2050, respectively, for 10,000 iterations. Third, if we analyze the `atomicAdd()` function, its execution time is linear to the number of blocks that call it, as shown in Figure 4.12. On the GTX 280, atomic add takes 2.307ms per 10,000 execution. On the Tesla Fermi C2050, this time is 0.106 ms for the same number of operations, which is much more efficient than that on the GTX 280. Because of this reason, the GPU lock-based synchronization has worse performance than the lock-free one on the GTX 280, but the former has better performance than the latter on the Tesla Fermi C2050. Fourth, for execution time of the memory fence function `__threadfence()`, it is related to the number of threads in the kernel. With the max number of blocks that can be configured in the kernel using the GPU synchronization (30 blocks on the GTX 280 and 14 blocks on the Tesla Fermi C2050), `__threadfence()` needs more time on the GTX 280 than on the Tesla Fermi C2050.

4.7.4 Evaluation in Real Algorithms

In this section, we evaluate our proposed algorithms for GPU synchronization in three algorithms—FFT, Smith-Waterman, and bitonic sort. In this experiment, we exclude the data copy time and consider only the kernel execution time. Also, we record the total

execution time by repeating the kernel execution 1000 times and report the average.

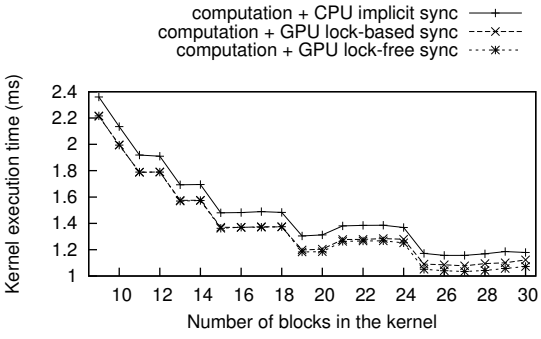
4.7.4.1 Kernel Execution Time

Figures 4.13 and 4.14 show the kernel execution time with different synchronization approaches on the GTX 280 and the Tesla Fermi C2050, respectively. Also shown are the time change with regard to the number of blocks in the kernel. In these figures, we demonstrate the kernel execution time with the number of blocks varying from 9 to 30 on the GTX 280 and 4 to 14 on the Tesla Fermi C2050. The reason is that performance of the applications with the number of blocks outside this range is worse than that with the number of blocks within this range. Another reason is, with the GPU synchronization approach used, the maximum number of blocks in a kernel is 30 and 14 on the GTX 280 and the Tesla Fermi C2050, respectively. In each block, on the GTX 280, the number of threads per block is 448, 256, and 512 for FFT, Smith-Waterman, and bitonic sort, respectively. On the Tesla Fermi C2050, the number of threads in each block is 640, 256, and 1024 for the three applications. Figures 4.13(a) and 4.14(a) show the performance of FFT, Figures 4.13(b) and 4.14(b) are for Smith-Waterman, and Figures 4.13(c) and 4.14(c) display the kernel execution time of bitonic sort.

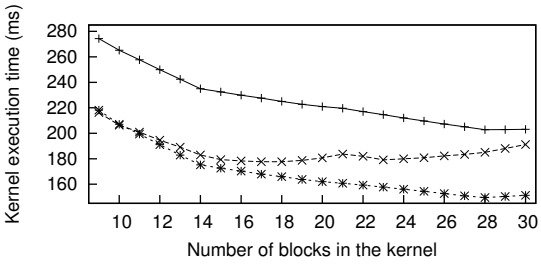
From Figure 4.13, we can see the following trends on the GTX 280. First, kernel execution time decreases with more blocks configured in the kernel. The reason is, with more blocks (from 9 to 30) in the kernel, more GPU resources are used for the computation. As a result, less time is needed for the program execution. Second, with our GPU synchronization used, performance improvement is observed in all the three algorithms. For example, kernel execution time of FFT decreases from 1.179ms with the CPU implicit synchronization to 1.072ms with the GPU lock-free synchronization (30 blocks are configured in the

kernel), corresponding to a 9.08% decrease. For Smith-Waterman and bitonic sort, these values are 25.47% and 40.39%, respectively. Third, the kernel execution time difference between the CPU implicit synchronization and the proposed GPU synchronization of FFT is much less than that of Smith-Waterman and bitonic sort. The reason is, in FFT, the computation load between two barriers is much more than that of Smith-Waterman and bitonic sort. According to Equation (4.4), the kernel execution time change caused by the synchronization time decrease in FFT is not as much as that in Smith-Waterman and bitonic sort.

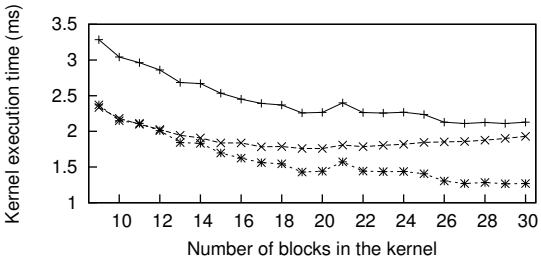
For the kernel execution time on the Tesla Fermi C2050, overall, similar trends are observed as that on the GTX 280. As can be seen in Figure 4.14, first, kernel execution time decreases with more blocks configured in the kernel. The reason is the same as that on the GTX 280, i.e., more blocks means more resources are used for the GPU computing. Second, performance of all three applications is improved with the GPU synchronization applied. Specifically, the execution time of FFT is 0.708ms with the CPU implicit synchronization. This time decreases to 0.661ms with the GPU lock-based synchronization, corresponding to a decrease of 6.64%. With the GPU lock-free synchronization, the time of FFT is 0.638ms, a relative decrease of 9.75%. Similarly, the execution time of Smith-Waterman decreases from 136.461ms with the CPU synchronization to 102.620ms and 109.236ms with the GPU lock-based and lock-free synchronization, respectively, corresponding to a performance improvement of 24.80% with the GPU lock-based synchronization and 29.95% with the GPU lock-free synchronization. As for bitonic sort, performance improvement is 13.13% with the GPU lock-based synchronization and 9.19% with the GPU lock-free synchronization compared to the CPU implicit synchronization.



(a) FFT

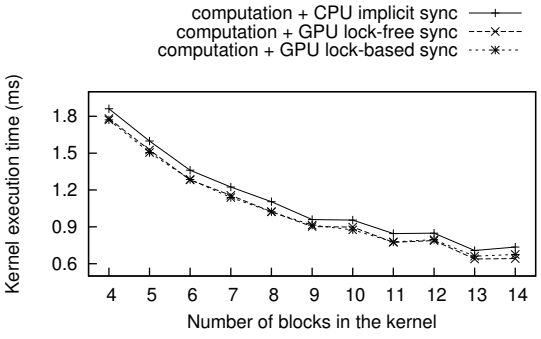


(b) Smith-Waterman

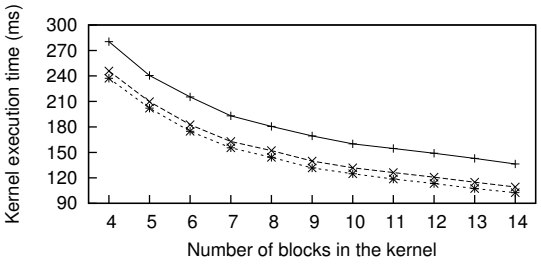


(c) Bitonic sort

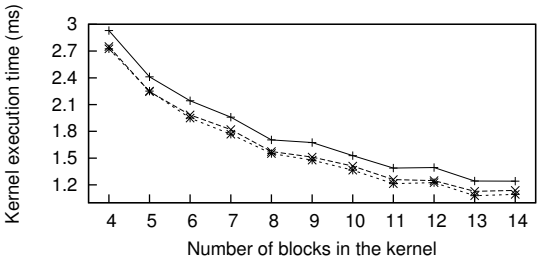
Figure 4.13: Kernel Execution Time versus Number of Blocks in the Kernel on the GTX 280



(a) FFT



(b) Smith-Waterman



(c) Bitonic sort

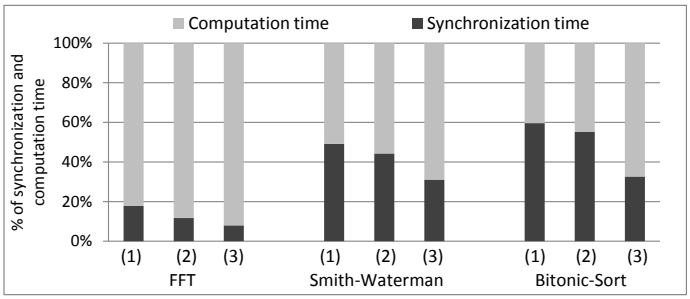
Figure 4.14: Kernel Execution Time versus Number of Blocks in the Kernel on the Tesla Fermi C2050

Comparing the performance on the GTX 280 and the Tesla Fermi C2050, first, each application needs more time on the GTX 280 than on the Tesla Fermi C2050. The reason is that Tesla Fermi C2050 is a newer generation of GPU and it is more powerful than the GTX 280 (The Tesla Fermi C2050 has the peak performance of 1030.4 GFLOPS for single-precision and the global memory bandwidth of 153.6 GB/s, and that of the GTX 280 are 622.1 GFLOPS and 141.7 GB/s, respectively.). Second, for the two GPU synchronization approaches, GPU lock-free synchronization achieves much better performance than the GPU lock-based one on the GTX 280. But on the Tesla Fermi C2050, their performance difference is very little. Either approach can achieve better performance than the other. As we can see, FFT achieves better performance with the GPU lock-free synchronization; while Smith-Waterman and bitonic sort have better performance with the GPU lock-based synchronization. The reason is that the atomic add used in the GPU lock-based synchronization is much more efficient on the Tesla Fermi C2050 than on the GTX 280, which significantly improves the performance of the GPU lock-based synchronization on the Tesla Fermi C2050.

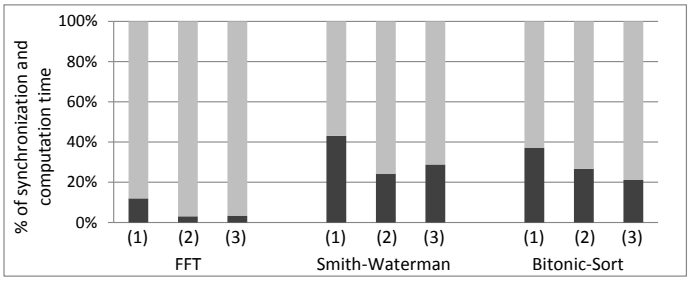
4.7.4.2 Percentages of the Computation Time and the Synchronization Time

Figure 4.15 shows the performance breakdown in percentage of the three algorithms when different synchronization approaches are used. As we can see, on both the GTX 280 and the Tesla Fermi C2050, the percentage of synchronization time in the FFT is much less than that in Smith-Waterman and bitonic sort. As a result, changes in synchronization have less impact on the total kernel execution time. As shown in Figures 4.13 and 4.14, the kernel execution times of FFT are very close with the different synchronization approaches; while the program execution time changes a lot in Smith-Waterman and bitonic sort. In

addition, on the GTX 280, the synchronization time consumes 50% and 60% in Smith-Waterman and bitonic sort, respectively, when the CPU implicit synchronization approach is used. Though percentages of the synchronization time are less on the Tesla Fermi C2050 than that on the GTX 280, they are still up to 43% and 37% for the Smith-Waterman and the bitonic sort, respectively. This indicates that inter-block communication occupies a large part of the total execution time in some algorithms and decreasing the synchronization time is important to achieve good performance.



(a) GTX 280



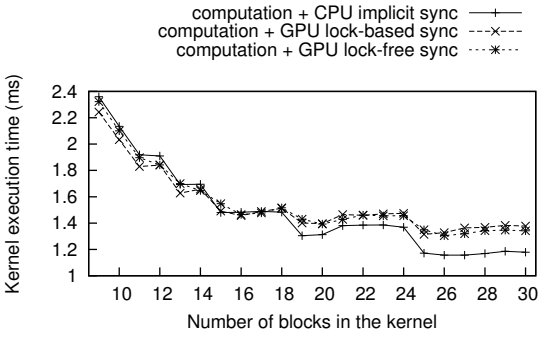
(b) Tesla Fermi C2050

Figure 4.15: Percentages of Computation Time and Synchronization Time (Note: (1) CPU implicit synchronization (2) GPU lock-based synchronization (3) GPU lock-free synchronization)

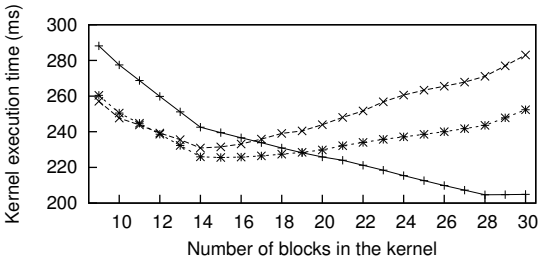
4.7.5 Cost of Guaranteeing Inter-Block Communication Correctness

In this section, we show the cost of the memory fence function to guarantee the data communication correctness. Figures 4.16 and 4.17 show the kernel execution time versus the number of blocks in kernels, where Figures 4.16(a) and 4.17(a) show the results of FFT, Figures 4.16(b) and 4.17(b) are the results of Smith-Waterman, and Figures 4.16(c) and 4.17(c) are for bitonic sort.

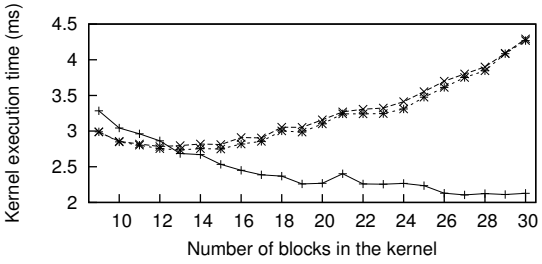
As can be seen, on the GTX 280, the more blocks are configured in kernels, the more overhead is caused, which can even exceed the kernel execution time using the CPU implicit synchronization. With the GPU lock-free synchronization, when the number of blocks in the kernel is larger than 14, the execution of FFT needs more time than that using the CPU implicit synchronization. The threshold values are 18 and 12 for Smith-Waterman and bitonic sort, respectively. In contrast, overhead of the memory fence function on the Tesla Fermi C2050 is much less than that on the GTX 280. As we can see, even with the memory fence function called, both GPU synchronization approaches achieve better performance than the CPU implicit synchronization approach for all three applications. From these results, our proposed barrier synchronization can synchronize the execution of different blocks in a more efficient way than the commonly used CPU implicit synchronization. As for the cost of the memory fence function `__threadfence()`, it is very high on the GTX 280, which means using `__threadfence()` to guarantee inter-block data communication correctness is an inefficient approach. But on the Tesla Fermi C2050, this approach is effective. It is worth noting that even without `__threadfence()` called in our barrier functions, all program results are correct with thousands of runs.



(a) FFT

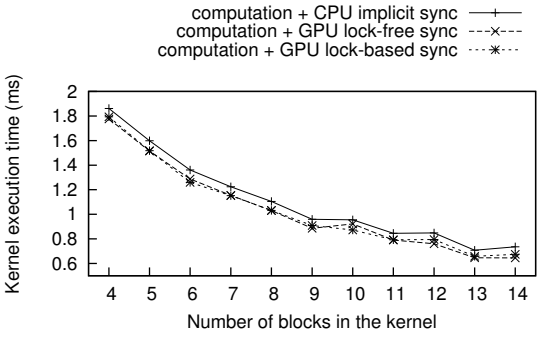


(b) Smith-Waterman

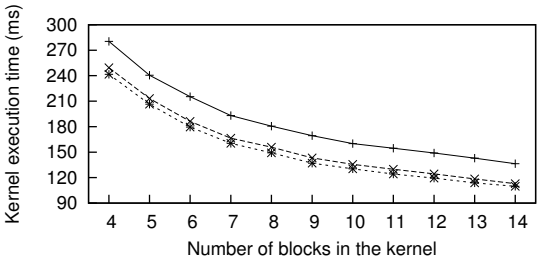


(c) Bitonic sort

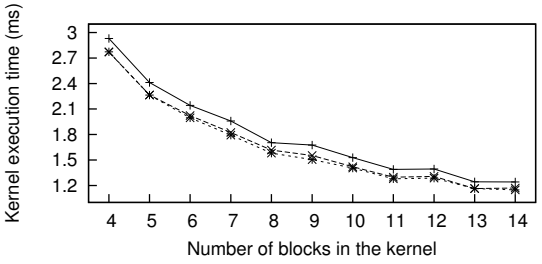
Figure 4.16: Kernel Execution Time versus Number of Blocks in the Kernel with `__threadfence()` Called on the GTX 280 GPU



(a) FFT



(b) Smith-Waterman



(c) Bitonic sort

Figure 4.17: Kernel Execution Time versus Number of Blocks in the Kernel with `__threadfence()` Called on the Tesla Fermi C2050 GPU

4.8 Extension to the OpenCL Programming Model

In previous sections, we presented the inter-block barrier synchronization in CUDA. In practice, there are other GPU cards (e.g., AMD GPUs) as well as other programming models such as the OpenCL. As we know, AMD GPUs have higher peak performance than their NVIDIA counterparts and the OpenCL programming model can support GPUs, CPUs, and even the Cell Broadband Engine (Cell/BE) [21]. Thus, GPU synchronization in the OpenCL programming model is of great importance in practice. In this section, we analyze the feasibility of extending the GPU synchronization strategy to the OpenCL programming model and demonstrate the performance of the OpenCL GPU synchronization using the OpenCL version of the microbenchmark in Section 4.7.2.

As noted in Section 4.5, the implementation of the GPU synchronization needs support from the programming model. Specifically, in the OpenCL programming model, we need the following: 1) built-in index for identifying different work-items within a work-group. 2) barrier synchronization across work-items within a work-group; 3) atomic operations (e.g., atomic add) for global mutex variables; and 4) a memory fence function to guarantee memory consistency across different work-groups. In the OpenCL programming model, the first three features, i.e., built-in work-item index, barrier synchronization within a work-group, and atomic add for global variables are supported. As to the memory fence function for guaranteeing global memory consistency, according to [3], OpenCL provides a memory fence function `memfence()` for intra-block memory consistency, i.e., it is the counterpart of `__threadfence_block()` in CUDA. However, there is no memory fence function for memory consistency across the whole GPU card corresponding to

Table 4.2: OpenCL CPU Synchronization Time

Synchronization Approach	GTX 280 (ms)	Tesla Fermi C2050 (ms)
CPU explicit synchronization	468.43	356.30
CPU implicit synchronization	118.84	87.94

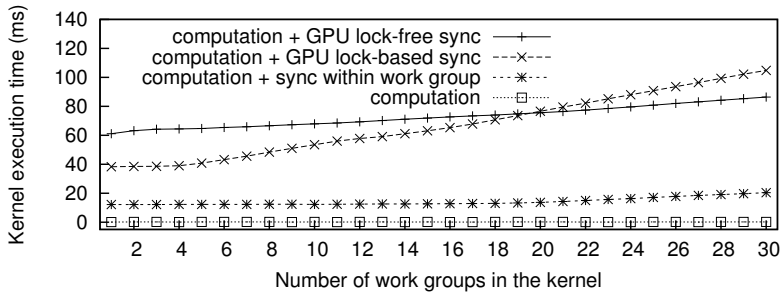
`__threadfence()`. As such, we can synchronize the execution of different work-items in different work-groups; but if inter-work-group data communication is needed, inter-work-group data communication correctness cannot be guaranteed.

Though memory consistency across different work-groups cannot be guaranteed, we demonstrate the possible performance improvement brought by the OpenCL GPU synchronization, which can help in evaluating the usefulness of the OpenCL GPU synchronization in practice and possibly motivate the introduction of a memory fence function across the whole GPU card.

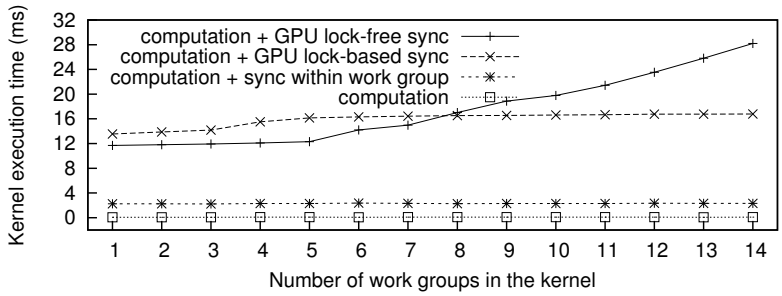
To evaluate the performance of the OpenCL GPU synchronization, we translate the microbenchmark from Section 4.7.2 into the OpenCL programming model. The translation is straightforward, i.e., the intra-block function `__syncthreads()` is translated into the `barrier()`, with both the local and global memory indicated to be synchronized. Similarly, the atomic function `atomicAdd()` is changed to `atom_add()`, etc.

Figure 4.18 shows the kernel execution time with the various synchronization approaches in OpenCL on the GTX 280 and the Tesla Fermi C2050. Note that the kernel execution times with the CPU synchronization are almost constant, and we list them in Figure 4.2 to make the figures easy to read.

From Figure 4.18, we have the following observations. On the GTX 280, CPU synchronization takes much more time in OpenCL 1.0 than in CUDA. Specifically, kernel execution time with the CPU implicit synchronization in CUDA is 81ms, while that in



(a) GTX 280



(b) Tesla Fermi C2050

Figure 4.18: OpenCL Barrier Synchronization Time (Note: To make the figure easy to read, kernel execution times with CPU synchronization are listed in Table 4.2.)

OpenCL is 119 ms. With the CPU explicit synchronization, kernel execution times are 169 ms and 468 ms for CUDA and OpenCL, respectively. One of the reasons can be that OpenCL is based on CUDA for NVIDIA GPU usage. As such, execution of OpenCL programs needs an additional level of function call. Second, the barrier function within a work-group in OpenCL takes more time than in CUDA, too. In OpenCL, the barrier function takes 13.98 ms for 10,000 times of execution; while in CUDA, this time is only 0.72 ms. Third, for the GPU synchronization, if the number of work-groups is less than 20, the GPU lock-based synchronization has better performance; otherwise, the GPU lock-free synchronization has better performance. Fourth, the kernel execution time will increase with more work-groups configured in the kernel for both GPU synchronization strategies.

For the GPU lock-based synchronization, there are two reasons for that. One is the atomic add, which can only be executed sequentially even in different work-groups. Another reason is the barrier function within the work-group. As can be seen, kernel execution time with only the intra-work-group barrier function increases with more blocks in the kernel. Since the barrier function is called in the our inter-work-group barrier function, it causes more overhead to kernels with more work-groups. As to the GPU lock-free synchronization, there are three intra-work-group barrier functions called, thus kernel execution needs more time with more work-groups configured in the kernel.

On the Tesla Fermi C2050, similar trends are observed. First, CPU synchronization takes much more time in OpenCL than in CUDA. When using the CPU implicit synchronization, kernel execution takes 34 ms and 88 ms in CUDA and OpenCL, respectively. With the CPU explicit synchronization, the kernel time in CUDA and OpenCL is 247 ms and 356 ms, respectively. Second, the barrier function within a work-group takes much more time in OpenCL than in CUDA. The reason is the same as above, OpenCL functions are on top of the CUDA for NVIDIA GPUs, which makes OpenCL function calls need another level of function calls. Third, for the GPU synchronization, if the number of work-groups in the kernel is less than 10, GPU lock-based synchronization has better performance. Otherwise, the GPU lock-free synchronization has better performance. The reason is the same as that on the GTX 280, GPU lock-based synchronization needs atomic functions called in the kernel, while the GPU lock-free synchronization does not need that.

Comparing the performance on the GTX 280 to the Tesla Fermi C2050, the same synchronization approach takes less time on the latter than on the former, which is true for both the CPU synchronization and the GPU synchronization. The reason is that the

Tesla Fermi C2050 is more efficient than the GTX 280 in the operations—global memory bandwidth, intra-work-group barrier function, and atomic operations that are used in the barrier function.

4.9 Summary

In this chapter, we proposed two GPU synchronization strategies to synchronize the execution of different blocks on the GPU card. With the proposed GPU synchronization approaches, data communication between GPU blocks can be performed without launching the kernels multiple times, thus the overhead of switching back and forth between the host and the device can be avoided. For evaluation, we integrated our proposed synchronization strategies into a microbenchmark and three well-known algorithms. We compared the program execution time using CPU and GPU synchronization approaches. From our experimental results, application performance can be improved with fast synchronization. On the Tesla Fermi C2050, performance improvement was observed with our proposed GPU synchronization approaches. However, on the GTX 280, the low efficiency of the memory fence function made the cost of guaranteeing correctness extremely high. As a consequence, there was no performance improvement when compared to the commonly used CPU synchronization approach.

Chapter 5

Task Migration

5.1 Overview

In Chapter 3, we proposed the VOCL framework, an implementation of the OpenCL programming model that provides the ability to use and share non-local computational accelerators through device virtualization. VOCL establishes device proxies which manage the mapping of virtual to physical OpenCL contexts and forward OpenCL commands from the application to the physical device. VOCL can be used immediately by any OpenCL application and provides a runtime system that automatically manages the mapping of virtual to physical GPUs.

The ability to migrate virtual devices is a key capability in any virtualized environment. In the context of VOCL, migration can enable the ability to migrate virtual GPUs when a failure is detected; on-demand maintenance of compute resources; and dynamic adjustment of the mapping of virtual to physical devices to manage resource allocation and balance the workload.

In this chapter, we extend VOCL to support transparent, live migration of virtual OpenCL GPUs. Migration is achieved by transparently moving the virtual GPU state between VOCL proxies and physical devices and remapping the virtual-to-physical translation. Asynchronous, one-sided communication is used to decouple and coordinate migration. In addition, a command queueing strategy is introduced that allows the proxy greater control over the migratability of the virtual GPU and increases its responsiveness to migration events. We evaluate our migration framework on four application benchmarks from three application domains: dense linear algebra, n-body calculations, and data intensive bioinformatics. Results indicate that with our queueing technique, VOCL incurs low migration overhead while maintaining fast response time to migration events. In addition, through migration-enabled load balancing, applications achieve speedups from 1.7 to 1.9-fold.

The remainder of this chapter is organized as follows. Section 5.2 discusses the related work of task migration. In Sections 5.3 and 5.4, we present the virtual GPU migration framework and evaluate the overhead of migration and benefits from migration, respectively. Finally, we conclude this chapter with Section 5.5.

5.2 Related Work

Our work is related to task migration across different GPUs and different nodes. There have been many studies available for migration in large-scale computing systems.

Process migration can be achieved by the checkpointing approach. One study is the Berkeley Lab Checkpoint/Restart (BLCR) [37]. It writes the process image to a file and

then restarts the process from the process image file. This approach can be used for migrating a process from one node to another, but it only considers the images of CPU processes.

Based on BLCR, Ouyang [67] used a proactive job migration scheme to enhance the fault tolerance of the MVAPICH2 [60]. This work implemented the checkpoint/restart procedure by transferring the process image to a healthy spare node for the purpose of resuming the process. Wang et al. [85] proposed a process-level live migration mechanism to support continued execution of MPI processes. This work is integrated into an MPI execution environment to transparently sustain health-inflated node failures, which eradicates the need to restart and requeue MPI jobs. These studies are related to the task migration in our VOCL framework in that VOCL supports live migration of *virtual GPUs* (*VGPUs*) from one physical GPU to another and migration is transparent to the program execution.

Takizawa et al. [36, 79] demonstrated the feasibility of migrating a GPU program from one node to another. This work is similar to ours but differs in the following ways. First, in their work, an API proxy is added to store the image file, which makes OpenCL function calls become a two-phase procedure. As such, when large amounts of data are transferred between host memory and device memory, significant overhead can be incurred during program execution even for local GPU usage. In contrast, in our VOCL framework, there is no such API proxy on the local node, and no additional overhead is caused to the usage of local GPUs. Second, when migration is triggered, in Takizawa's work, execution of the process must be terminated and restarted on the target machine. In contrast, migration in the VOCL framework does not require process termination and restart. It is transparent to the application program and happens during its execution. Third, the process image is stored on the hard disk in Takizawa's migration approach, which unduly burden the storage

subsystem and can cause significant overhead for restarting the process. In contrast, VOCL does not use the hard disk and all data is transferred over the network.

Another strategy for task migration is based on virtual machines, such as Xen [16], which enable migration of virtual OS instances across different compute nodes. One such example is vCUDA [72]. In this approach, all API function calls on the target OS need to be redirected to the host OS when migration happens. As a result, it causes significant migration overhead on both the host and the target nodes.

5.3 Transparent Virtual GPU Migration

In this section, we discuss the extension of the VOCL framework with the ability to migrate virtual GPU images across different physical GPUs. Overall, task migration is performed based on *virtual GPU*, which represents the GPU resources utilized by an application process on each physical GPU and is the unit to be migrated across physical GPUs. We first describe the virtual GPU abstraction, followed by the virtual GPU migration algorithm, and finally we introduce a queueing mechanism, which can be used to improve the performance of migration.

5.3.1 The Virtual GPU Abstraction

A *virtual GPU* (or *VGPU*) represents the resources used by an application process on a physical GPU, which includes OpenCL contexts, command queues, memory buffers, programs, and kernels. It also stores the dependencies across the resources, as shown in Figure 5.1. An application process could use multiple physical GPUs, with each represented by a virtual GPU. Similarly, one physical GPU could be shared by multiple applications.

In such a case, an individual virtual GPU will be created for each application. We illustrate the two cases in Figure 5.2.

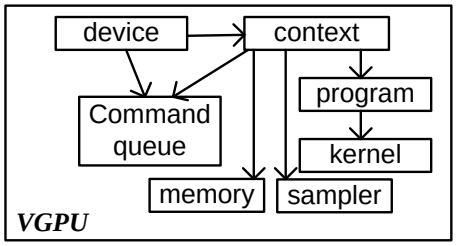


Figure 5.1: Virtual GPU Components and their Dependencies

In the VOCL framework, virtual GPU objects exist in both the VOCL library and the proxy with a one-to-one mapping relationship, as shown in Figure 5.2. That is, for each virtual GPU in a proxy, there is a corresponding virtual GPU in the VOCL library. A virtual GPU in the VOCL library contains VOCL resources, which is referred to as *VOCL VGPU*; while a virtual GPU in the proxy contains OpenCL resources and is referred to as *OpenCL VGPU*. In this chapter, we use the *source GPU* to indicate a GPU where migration is originated from and *destination GPU* for the migration destination. A *source proxy* is the proxy that contains the source GPU. Similarly, the destination GPU belongs to the *destination proxy*.

An OpenCL VGPU in the proxy is identified by the OpenCL device ID and the index of the application that is using the device. Once an application selects a physical GPU, a VGPU is created and all OpenCL resources created by the application on the physical GPU will be saved in its VGPU. Besides OpenCL handles, information used to create the handles are also stored. For instance, when an OpenCL program is created, besides the program handle, we need to store its source code and build options. The reason is that OpenCL handles created on one physical GPU may be invalid on another. Therefore, when

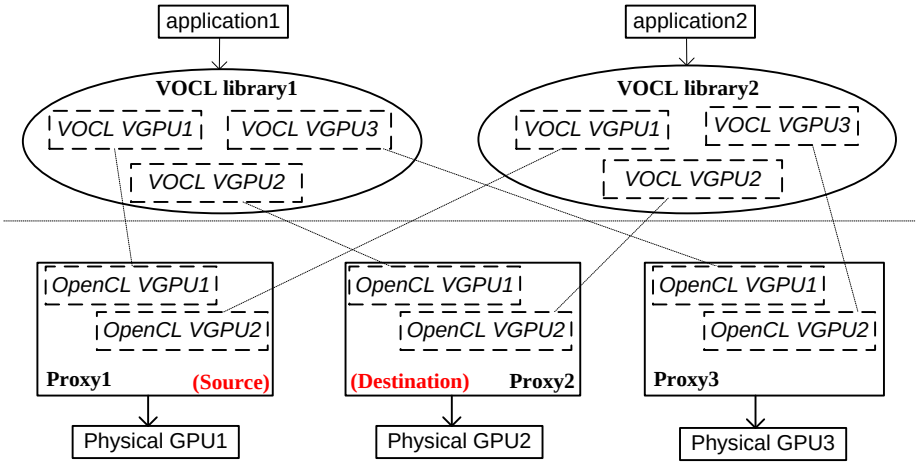


Figure 5.2: Virtual GPUs

a VGPU is migrated, we need to re-create all the OpenCL resources on the destination GPU, which requires all the information.

VOCL VGPUs reside in the VOCL library and have a one-to-one correspondence with OpenCL VGPUs. Each VOCL VGPU is identified by the VOCL device ID and the index of the proxy where the device is located. VOCL VGPUs store information such as VOCL contexts, VOCL command queues, and VOCL programs. In contrast to OpenCL VGPUs, which are created in the destination proxy and released in the source proxy in a migration, the update of a VOCL VGPU propagates from the source OpenCL VGPU to the destination OpenCL VGPU, as shown in Figure 5.3. Specifically, for each VOCL handle in the VOCL VGPU, its corresponding OpenCL handle and MPI data communication information will be replaced by its counterpart in the destination OpenCL VGPU. As a result, all OpenCL function calls will be directed to the destination proxy and GPU computation will be performed on the destination GPU. Since we keep the same VOCL handle in the VOCL VGPU, migration is transparent to the application.

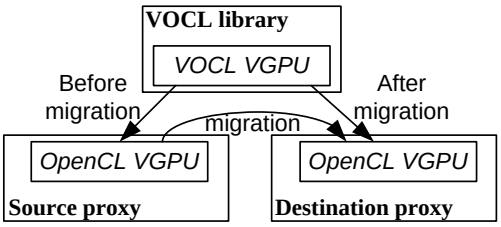


Figure 5.3: Migration Scenario

5.3.2 Migrating Virtual GPUs

Migrating a VGPU across physical GPUs requires manipulation of the OpenCL and VOCL VGPU objects as well as management of the GPU device and transfer of the VGPU image. When a migration is initiated, the OpenCL VGPU is migrated from the source proxy to the destination proxy. In the VOCL library, the corresponding VOCL VGPU must also be mapped from the source OpenCL VGPU to the destination OpenCL VGPU. As a result, GPU computation on the source physical GPU will be performed on the destination physical GPU after migration, as shown in Figure 5.3. However, careful synchronization must be employed to preserve data consistency and provide migration that is transparent to the user.

We argue that migration should start as quickly as possible with a minimum overhead. To achieve this, we extend the VOCL framework in the following two aspects: 1) an internal command queue in the proxy to reduce the time for waiting the issued kernels to be completed, which in turn supports a quick start of migration and 2) atomic enqueueing of OpenCL function calls to reduce the migration overhead. In the following, we first describe the conditions in which migration should be triggered, then we explain the steps involved in the migration. Finally, the above extension to the VOCL framework to improve the migration performance is presented.

Currently, we consider two scenarios in which migration can be triggered: to free the GPU resources on a node (e.g., to perform maintenance) and to rebalance the virtual-to-physical mapping of GPUs. To free the GPU resources on a given node, we provide a tool, *voclForcedMigration*, to send messages to the proxy. When a proxy receives the forced migration message, it will move all its tasks to other nodes. The second scenario is for load balance. To enable this, we provide a function called `voclRebalance()` to check the loads on the physical GPUs. If the load difference across physical GPUs is larger than a threshold value, migration will happen to rebalance the loads on different GPUs. Currently, we use a threshold of half of the internal queue depth N described in Section 5.3.3, which means if the difference of the number of function calls that are issued to the native OpenCL library but not completed on two physical GPUs is larger than $N/2$, migration will be triggered. Many other strategies are possible and VOCL provides an interface that allows users to define new load balancing modules.

Migration of a virtual GPU image across physical GPUs is performed using the following algorithm:

1. **Lock the VGPU:** The VGPU is locked to prevent commands from being issued during migration.

2. **Drain command queue:** Before starting the migration procedure, the source proxy must wait for completion of all issued OpenCL function calls by invoking the OpenCL function `clFinish()`.

3. **Select physical GPU:** Select the physical GPU to which the virtual GPU will be mapped. Many criteria are possible; we select the physical GPU with the least

computation load. In this step, the source proxy queries the load on each available physical GPU.

4. **Transfer OpenCL VGPU:** The source proxy marshalls the source OpenCL VGPU and transmits it to the destination proxy. In the destination proxy, an OpenCL VGPU is created using this information.

5. **Create VOCL VGPU:** The new VOCL VGPU is created, connected to the selected physical GPU, and mapped to its corresponding OpenCL VGPU.

6. **Transfer contents of device memory:** Data in device memory of the source VGPU is sent to the new destination VGPU. Here, the data transfer is pipelined to reduce the data transfer overhead.

7. **Release source GPU:** After data transfer is completed, the source VGPU is released.

8. **Unlock the VGPU:** Release the migration lock on the VGPU and allow the client to resume issuing OpenCL commands to the destination VGPU.

5.3.3 Queueing Virtual GPU Operations

One of the first steps in migration is to wait for completion of all issued OpenCL commands. In this step, if a large number of function calls are issued and migration is necessary, we may need to wait a long time before migration can start, which impacts the delay until a fault can be migrated around, maintenance can be performed, or the load can be rebalanced.

To reduce the waiting time, instead of issuing all received OpenCL function calls to the GPU, the proxy creates an internal command queue to queue up the received functions as

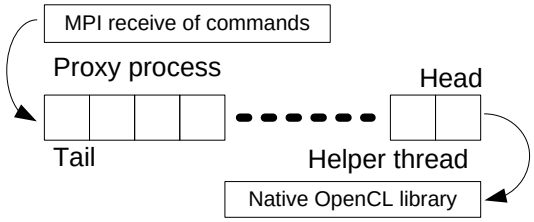


Figure 5.4: Internal Queue in Proxy

shown in Figure 5.4. When an OpenCL function call is received, the proxy enqueues it into the command queue. The proxy also creates a helper thread to issue function calls to the GPU. Each time, the helper thread issues a fixed number, N, of OpenCL function calls to the GPU and calls `clFinish()` to wait for their completion. After that, the helper thread issues another N functions and calls the `clFinish()`, so on and so forth. In this way, when migration is necessary, the proxy process only needs to wait for the completion of at most N function calls. As such, the wait for completion time can be significantly reduced compared to issuing all OpenCL function calls to the GPU. However, this approach will cause overhead to program execution since kernel execution becomes more synchronous by the calling of `clFinish()`. Use $N = 1$ as an example, it means that `clFinish()` is called after every kernel launch. As a consequence, a kernel cannot be launched until its previous kernel finishes the computation, which can cause significant overhead since kernel launch is overlapped with previous kernel's execution in general. Note that by tuning the *queue depth*, or N value, we can control the overhead of this approach at a low level, as we will show in our experimental results.

After the OpenCL VGPU is migrated to the destination proxy, the source proxy will send the unissued function calls to the destination proxy. After receiving the function calls, the destination proxy will first update the OpenCL handles to those in the destination

OpenCL VGPU, then enqueue them to its internal queue.

5.3.4 Atomic Enqueueing Commands in the Presence of Migration

When migration starts, the source proxy stops issuing OpenCL function calls to its GPU. Instead, it waits for completion of the issued function calls. At this time, any OpenCL function calls received from the VOCL library are stored in its internal command queue. Unissued function calls will be sent to the destination proxy and additional overhead is caused. On the other hand, if the VOCL library stops issuing function calls to the source proxy, the number of function calls to be sent from the source proxy to the destination proxy can be reduced and migration overhead is reduced as a result. To achieve this, we use a *migration lock* to prevent the VOCL library from issuing function calls to the source proxy when migration is in progress.

We utilize the MPI one-sided mutex algorithm of Ross et al. [69] to establish a migration lock between the application and the VOCL proxy. When migration starts, the source proxy acquires the mutex and holds it until migration is complete. In the VOCL library, the application must acquire the mutex before it can issue OpenCL function calls to the device. As such, if there is no migration, the application can always acquire the mutex immediately. On the other hand, if migration is in progress in the proxy, the application must wait for the completion of the migration, when the mutex will be released before it can issue a function call to the proxy. In this way, function calls are restricted in the application when migration is in-progress.

Since the application is expected to issue OpenCL function calls much more frequently

than migration will occur, the mutex structure is located in the VOCL library on the application's node to reduce the overhead of locking.

5.4 Experimental Evaluation

We use the four application kernels in Chapter 3 to evaluate the task migration within the VOCL framework. As mentioned before, matrix multiplication and n-body are compute intensive, while matrix transpose and Smith-Waterman need more data movement between host memory and device memory. Using these kernels, we measure the cost of migration; demonstrate the performance impact of rebalancing the mapping of VGPUs to physical GPUs; and explore the tradeoff between a shallow queue depth, which decreases the time-to-migration, and a deeper queue depth which can improve the efficiency of kernel execution.

Experiments were conducted on four QDR InfiniBand-connected compute nodes. Each node contains with two AMD Magny-Cours CPUs (Each has eight cores.), 64 GB of memory, and two NVIDIA Tesla M2070 GPUs, each with 6GB of global memory. The two GPUs are connected to different PCI express links and one GPU shares its PCI express link with the InfiniBand network interface card (NIC). In our experiments, we use two of the nodes as the remote GPU nodes and the other two as the local nodes, on which only the CPU is used. Each node runs the CentOS 5.5 Linux operating system and the CUDA 3.2 toolkit is installed to provide OpenCL support. In addition, we use the MVAPICH2 [60] MPI implementation, which supports the QDR InfiniBand interconnect. Each of our experiments is conducted three times and the average value is reported.

5.4.1 Overhead Caused by Migration Locker

In this section, we measure the overhead of the mutex lock to restrict the number of function calls when migration starts. We ran the four applications in the VOCL framework without migration. Then the program execution time was measured with and without the mutex lock used in the OpenCL function call. Table 5.1 shows the program execution time with and without the mutex lock. As we can expect, the performance difference observed by acquiring and releasing the mutex lock is effectively negligible. In some cases, the program execution time using the lock is actually less than the case without the mutex lock, meaning there are factors other than the mutex lock that impact the application performance more. To characterize the overhead of acquiring and releasing the mutex lock, we write a microbenchmark that acquires and releases the mutex lock 10,000 times. We measure the overhead in two scenarios. In one case, the lock is local to a process while in the other case, the lock is remote. From the experimental results, the execution times of acquiring/releasing the local lock and the remote lock 10,000 times are 15.01 ms and 157.72 ms, respectively. In other words, the time consumption of each mutex lock acquire/release is 1.5 us for the local lock and 15.7 us for the remote lock. Compared to the kernel execution time, which are in the order of milliseconds, this overhead is modest and thus can be ignored in practice.

5.4.2 Impact of Command Queue Depth

As described in Section 5.3.3, the proxy issues batches of N kernels to the OpenCL library and then blocks on their completion by calling `clFinish()`. The value of N determines

Table 5.1: Program Execution Time with and without MPI Mutex Locker. (Program size used in this experiment is as follows: matrix size is 1024 x 1024 in matrix multiplication and matrix transpose, sequence size is 1024 characters in Smith-Waterman, and the number of bodies is 15360 in n-body.)

Application kernels	Without MPI mutex lock (seconds)	With MPI mutex lock (seconds)
Matrix multiplication	7.499	7.498
N-body	36.788	36.786
Matrix transpose	9.069	8.768
Smith-Waterman	16.848	17.531

the depth of the OpenCL command queue and high values of N can improve kernel execution efficiency while low values of N reduce the delay between issuing a migration event and when migration can be performed.

Figure 5.5 shows the total execution time with no migration over a range of queue depths. `Infini` indicates that the queue has an infinite depth and that the proxy does not periodically invoke `clFinish()`. As can be seen, for matrix multiplication, matrix transpose, and n-body, program execution time with different queue depths shows little variation. Specifically, with $N = 2$, which means `clFinish()` is called after every two kernel launches, program execution time increases by 4.6%, 2.7%, and 0.8% respectively. The reason is that these three applications have long-running kernels that mitigate performance degradation from synchronous kernel execution. Smith-Waterman, on the other hand, launches a large number of short kernels, resulting in a slowdown of 256% when $N = 2$. Increasing the value of N reduces overhead for all four applications.

While increasing queue depth improves device utilization, it also increases the waiting time before migration can be initiated. Figure 5.6 shows the wait for completion time with N from 2 to 20. As can be seen, with the increase of the N value, the wait for completion

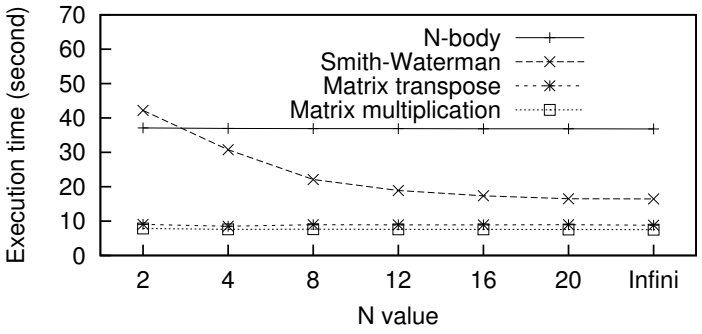


Figure 5.5: Overhead Caused by Internal Queue in Proxy

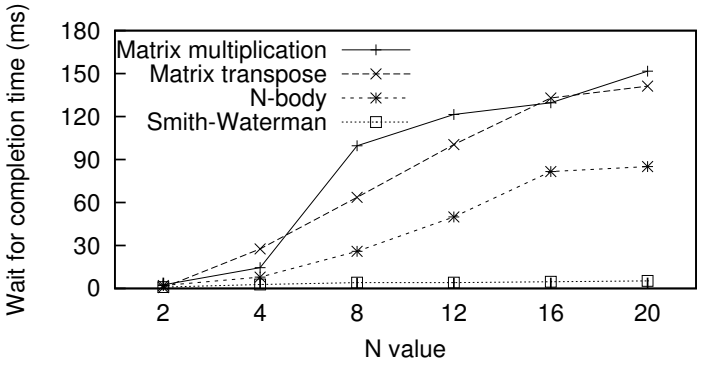


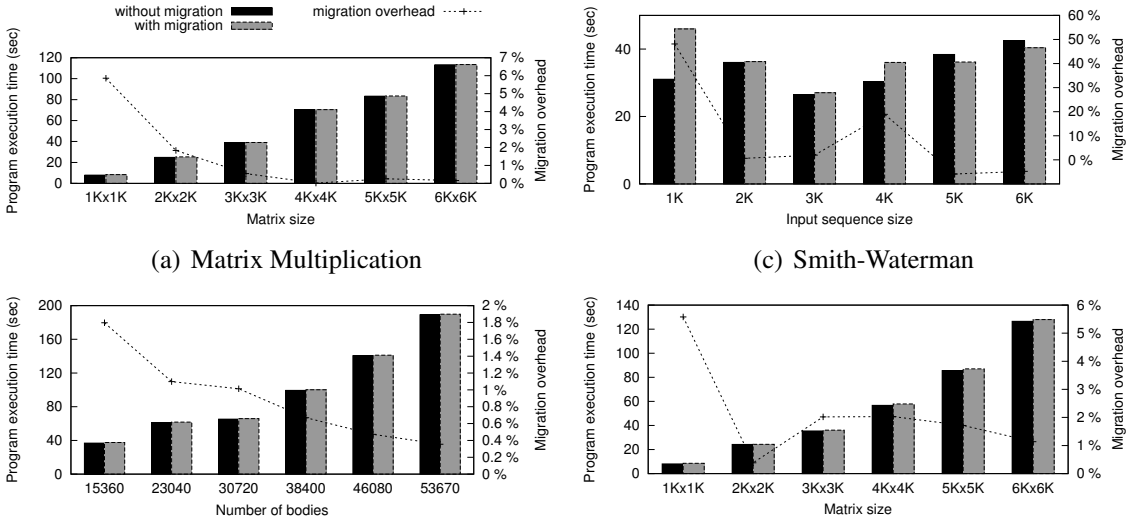
Figure 5.6: Wait for Completion Time with Different N Values

time will increase for all four applications. Note that the wait for completion time only affects the time interval between a migration event and when migration can be performed; it does not correspond to overhead.

Using the data from these two experiments, we choose a value of $N = 20$ as the default queue depth in VOCL (N is an user-adjustable parameter). When $N = 20$, the overhead caused by queueing is less than 2% and the wait for completion time is also relatively low, only a few hundred milliseconds for our applications.

5.4.3 Analysis of Migration Overhead

In Figure 5.7, we show the total execution time for each kernel with no migration and when a single migration is performed. From Figure 5.7, we see that, overall, as the problem size increases, the relative overhead decreases. The reason is that the execution time increases faster than the migration overhead with regard to the problem size. Thus, less relative overhead is caused in programs running a larger problem size. In addition, migration overhead is a few hundred milliseconds. For programs that run long enough, other factors such as network congestion and system noise affect the total execution time more than migration does. From these results, we conclude performance degradation caused by migration can be negligible for programs running a reasonably long time (e.g., a few tens of seconds).



(a) Matrix Multiplication

(c) Smith-Waterman

(b) N-body

(d) Matrix Transpose

Figure 5.7: Total Execution Time for each Kernel over a Range of Input Sizes with and without Migration.

Figure 5.8 and Table 5.2 show a detailed breakdown of the migration overhead for the

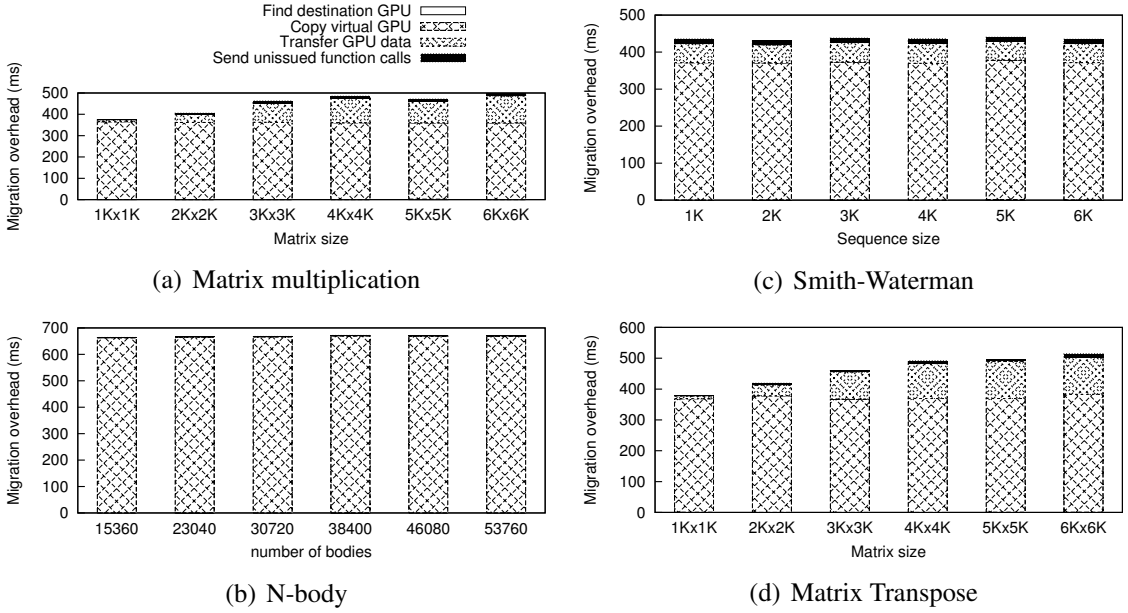
Table 5.2: Breakdown of Migration Overheads (in msec) for each Benchmark on the Smallest Input Problem.

	Find destination GPU	Copy VGPU	Data transfer	Send Un-issued function calls
Matrix multiply	0.265	364.799	8.636	1.317
N-body	0.106	662.485	0.953	0.408
Matrix transpose	0.217	368.485	8.527	1.400
Smith-Waterman	0.142	371.162	51.505	12.465

four benchmarks across different input sizes. The time for virtual GPU creation includes the latency of draining the device's command queue as well as the program build time for the destination VGPU, which dominate the overhead in all the four applications. For Smith-Waterman, we see that the large number of queued function calls generated by the application also increases the cost of migrating the queue of unissued functions. As for n-body, it contains two kernel programs, each of which needs to be built separately. As such, the copy VGPU time is about twice of the other three applications.

5.4.4 Performance Impact of Load Balancing

Migration also adds the capability to balance the VGPU workload across physical GPUs. In Figure 5.9 we show the performance improvement from VOCL's load balancing. For this experiment, we run two instances of the same benchmark and map both VGPUs to the same physical device. This represents a scenario where one device is initially occupied when the new VGPUs are created. In the baseline case, both instances share the physical GPU for their full execution. In the migration case, one of the applications triggers the VOCL load balancer which performs migration. This corresponds to a scenario in which resources become available while the application is running. After migration, each VGPU



(a) Matrix multiplication

(c) Smith-Waterman

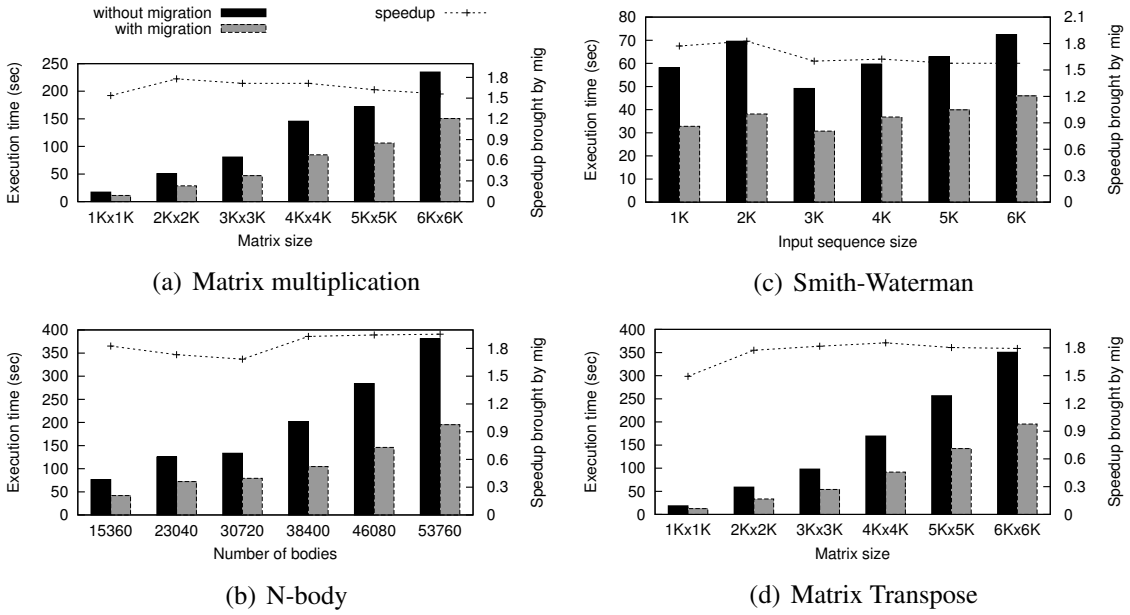
(b) N-body

(d) Matrix Transpose

Figure 5.8: Breakdown of Migration Overheads for each Benchmark across all Input Sizes.

is mapped to a separate physical GPU.

From this data, we see that, with task migration enabled in the framework, application performance is improved for all four applications. Specifically, the total time to complete both matrix multiplication instances reduces by a factor of 1.7; n-body is 1.9 times faster; matrix transpose is 1.7 times faster; and Smith-Waterman is 1.4 times faster. Among the four applications, speedup of n-body is the highest and Smith-Waterman is the lowest. This is consistent with the varying degree of overhead incurred by migration in all four applications, as shown in Figure 5.7; the least amount of data is transferred in the migration of n-body and Smith-Waterman requires the largest amount of data transfer. In addition, total execution time of n-body is much larger than that of the Smith-Waterman. As such, migration overhead has far less impact on its performance than Smith-Waterman.



(a) Matrix multiplication

(c) Smith-Waterman

(b) N-body

(d) Matrix Transpose

Figure 5.9: Total Execution Time for each Benchmark over a Range of Input Sizes without and with VOCL Load Balancing.

5.5 Summary

In this chapter, we extend our VOCL framework by supporting live task migration to achieve load balance and quick system maintenance. Migration in the VOCL framework is based on the virtual GPU, which consists of all the OpenCL resources used by an application program on a physical GPU. When migration happens, the source proxy collects the OpenCL objects and the information needed to create the objects in the source virtual GPU and sends them to the destination proxy. Then the destination proxy creates a new virtual GPU based on the received information. Also, data in the GPU memory and OpenCL function calls that are not issued to the source physical GPU will be sent to the destination proxy.

To reduce the migration overhead, we modify the VOCL framework in two aspects. First, we queue up received OpenCL functions and issued a few kernels each time instead of issuing all received OpenCL function calls. In this way, the time to wait for the completion of issued OpenCL function calls can be decreased. Second, we propose the atomic transaction approach to restrict issuing function calls in the VOCL library during the migration period. For performance evaluation, we measure the overhead of the migration lock and that of changing the VOCL framework to reduce the time for completion. Moreover, we profile the migration overhead and show the benefits of migration when load imbalance exists in a system.

Chapter 6

Application Verification

6.1 Overview

In previous chapters, we present the VOCL framework as well as its optimizations and extensions. VOCL enables applications to use GPUs in a more flexible way in large-scale heterogeneous systems. As a result, each application can transparently use all GPUs in the system and each GPU can be used by multiple applications simultaneously. In this chapter, we demonstrate the benefits of VOCL and the optimizations using two bioinformatics applications—Basic Local Alignment Search Tool (BLAST) [7, 89] and Smith-Waterman [73, 87]. Specifically, we parallelize the two applications in CUDA and show the performance improvement of the parallel implementations compared to their corresponding baseline sequential ones. To show the benefits of VOCL, we translate the CUDA parallel implementations to OpenCL and run them on VOCL. For BLAST, we show its performance improvement by transparently using multiple (both local and remote) GPUs in the system. For Smith-Waterman, as shown in Section 3.5.2, using remote GPUs causes

significant overhead, which is due to the large number of kernel launches used for inter-block data communication during its execution. As such, we show the performance improvement by using GPU synchronization in the use of remote GPUs.

In the following, we first give a brief description of the algorithm used in each application, followed by the parallelization on the GPU and the optimization techniques proposed to improve its performance. With the various parallel implementations in CUDA, we show the performance improvement compared to a baseline sequential implementation. After translating the CUDA parallel implementations into the OpenCL ones, we show the benefits of using the VOCL framework. For BLAST, we show the speedup achieved by using multiple GPUs (both remote and local). For Smith-Waterman, we demonstrate the overhead decrease by using the GPU synchronization strategy for inter-block data communication.

6.2 Parallelization of Basic Local Alignment Search Tool for Protein Sequence Search

6.2.1 Algorithm Description

BLAST is actually a family of algorithms, with variants used for searching alignments of different types (i.e., protein and nucleotide) of sequences. Among them, BLASTP is used to compare protein sequences against a database of protein sequences. There are four stages in the BLASTP algorithm:

1. *Hit detection.* Hit detection identifies high-scoring matches (i.e., *hits*) of a fixed

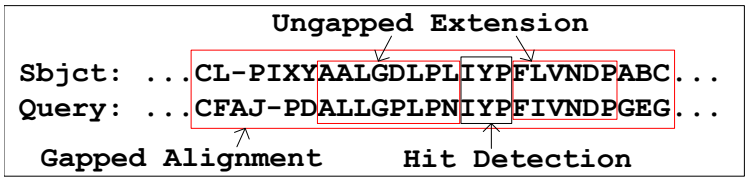


Figure 6.1: First Three Stages of BLAST Execution

length between a query sequence and a subject sequence (i.e., a database sequence).

2. *Ungapped extension*. Ungapped extension determines whether one or more hits obtained from the first stage can form the basis of a local alignment that does *not* include insertions or deletions of residues. The alignments with scores higher than a certain threshold will be passed to the next stage.

3. *Gapped alignment*. This stage performs further extension on the previously obtained alignments with gaps allowed. The result alignments will be filtered with another threshold.

4. *Gapped alignment with traceback.* In this stage, the final alignments to be displayed to users are re-scored, and the alignments are generated using a traceback algorithm.

Figure 6.1 gives an example of the first three stages of alignment computation. The fourth stage repeats the third one with traceback information recorded. BLAST reports alignment scores calculated based on a *scoring matrix* and *gap penalty factors*. In addition, statistical information such as the *"expect" value* that measures the significance of each alignment is reported.

6.2.2 Related Work

Since the BLAST tool is both compute- and data-intensive, many approaches have been investigated to parallelize BLAST in the past. On multi-core platforms, the BLAST implementation from National Center for Biotechnology Information (NCBI) has been parallelized with `pthreads`. On cluster platforms, there are parallel implementations such as TurboBLAST [18], ScalaBLAST [66], and mpiBLAST [25, 46, 47] available. Among them, mpiBLAST is a widely used parallelization of NCBI BLAST. Combining efficient task scheduling and scalable I/O design, mpiBLAST can effectively leverage tens of thousands of processors to speedup the BLAST search [45].

Parallel BLAST has also been implemented on accelerators such as FPGAs [39, 44, 53, 58, 74, 86]. In a recent study, Mahram et al. [53] introduced a co-processing approach that leverages both the CPU and FPGA to accelerate BLAST. Specifically, their implementation parallelizes the first two stages of BLAST on the FPGA to pre-filter dissimilar subject sequences. Then, the original NCBI BLAST is called on the CPU to search the filtered database. Their implementation can generate the same results as NCBI BLAST and achieve as much as 25-fold performance improvement.

Our work is mostly related to BLAST parallelization on GPUs. Liu et al. [48,49] developed CUDA-BLASTP and reported a 10-fold speedup over NCBI BLASTP on a desktop machine with two Tesla C1060 GPUs. CUDA-BLASTP uses a pre-filtering design similar to the FPGA study by Mahram et al. [53], and it does not parallelize all the compute stages of the BLASTP algorithm on the GPU. The filtering approach may suffer high overhead when searching BLAST jobs with a large number of subject sequences similar to the query.

Vouzis et al. [84] introduced another implementation of BLASTP on the GPU. In their implementation, databases are partitioned and processed on both the GPU and CPU, so that the system resources can be better utilized. Their approach also parallelizes only the first two stages on GPUs. With one CPU-helper thread, Vouzis's GPU BLAST implementation achieves between a three- and four-fold speedup for various query sequences.

6.2.3 Mapping BLAST on CUDA

In this section, we describe how we map the BLASTP algorithm onto the GPU.

6.2.3.1 Profiling of Serial BLASTP

We first profile the execution of BLASTP by searching two sequences (`query1` and `query2`) against the NCBI `NR` database, which contains 9,874,397 sequences with a total size over 5 GB. The sizes of `query1` and `query2` are 1K and 2K, respectively. Table 6.1 shows the time consumed by the four stages for searching the two query sequences. Note that the execution time of the first two stages cannot be separated because these two stages are executed together (details will be described in Section 6.2.3.2). Clearly, the first three stages, i.e., hit detection, ungapped extension, and gapped alignment consume more than 99% of the total execution time, regardless the query sequence length. Thus, our implementations focus on parallelizing the first three stages.

Table 6.1: Profiling of Serial BLASTP (Unit: Second. Note: Numbers in the bracket are percentages of the total execution time.)

Query sequence	Hit detection + ungapped extension	Gapped alignment	Gapped alignment w/ traceback	Total time
Query 1	144.28 (76.09%)	44.87 (23.67%)	0.46 (0.24%)	189.61 (100.00%)
Query 2	260.05 (76.56%)	78.92 (23.23%)	0.70 (0.21%)	339.67 (100.00%)

6.2.3.2 Hit Detection and Ungapped Extension Parallelization

As mentioned earlier, the first two stages of BLASTP, i.e., *hit detection* and *ungapped extension* are actually combined together. The algorithm first picks up a word from the beginning of the subject sequence and scans it against the query sequence to find hits. Once a hit is found, the extension is performed immediately on the hit, in both directions. After the extension is done, the algorithm moves on to scan for the next hit in the query sequence that matches the current word of the subject sequence so far and so on. After the current word of the subject sequence is compared against the entire query sequence, the algorithm moves on to the next word in the subject sequence. Since the scanning of a subject word depends on the hit detection and extension results of previous subject words (for more details see [8]), only limited parallelism can be exploited in aligning a pair of sequences in the current BLASTP implementation. Consequently, we parallelize the BLASTP algorithm by having each thread align a pair of sequences (i.e., a query sequence and a subject sequence). In this way, multiple pairs of sequences are aligned concurrently, as shown in Figure 6.2.

Before the kernel is launched to the GPU, query sequences, subject sequences, and a few other data structures are transferred from the host memory to the device memory. Each time, there is one query sequence on the device, which is shared by all threads in the kernel. Also, the database is divided into different chunks, which are searched one after another on the GPU card (one kernel launch per chunk). The chunk size is limited by the global memory size as well as the size of on-chip memory (as described in Section 6.2.4.1) on a GPU card. By searching one chunk at a time, our GPU implementation can process a database of arbitrary size.

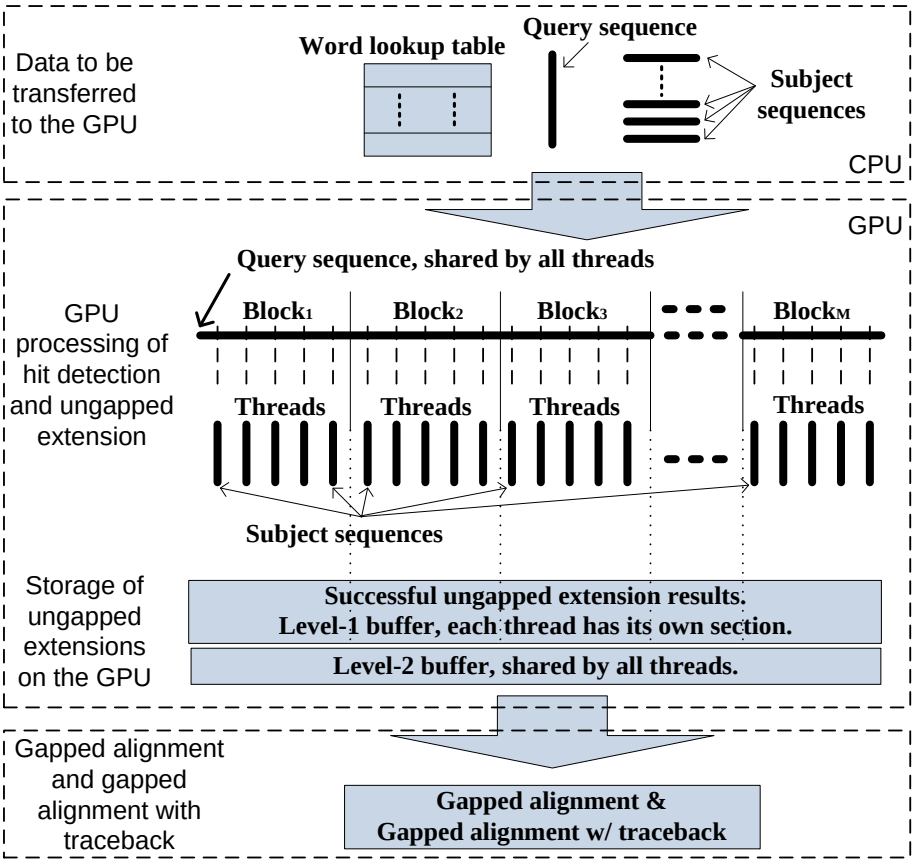


Figure 6.2: Hit Detection and Ungapped Extension Parallelization

Within a kernel launch, different threads align different subject sequences against the query sequence. When a thread finds successful ungapped extensions, it stores them in the global memory. Since all threads can find ungapped extensions in parallel, care must be taken to avoid write conflicts between different threads. One design option is to have all threads share a global memory buffer, and each thread calls an atomic operation to find a write location for each ungapped extension. Such a design can incur high synchronization overhead because atomic operations are expensive on some GPUs (e.g., the GTX 280). Another design option is to maintain a fixed-size buffer for each thread to store ungapped

extensions.[1] This design can avoid the synchronization overhead of atomic operations. However, such a design is space-inefficient because the number of ungapped extensions generated by each thread can differ significantly, and that number cannot be known beforehand.

We propose a *two-level hierarchical* buffer for storing the ungapped extension, as shown in Figure 6.3. In the level-1 buffer, each thread is allocated a fixed-size segment which can store N ungapped extensions. The level-2 buffer, which can store M ($M >> N$) ungapped extensions, is shared by all threads and guarded with atomic operations. The writing procedure of ungapped extensions is given by Algorithm 1. As can be seen, a thread first writes an ungapped extension to its allocation in the level-1 buffer. When a thread uses up its allocation in the level-1 buffer, it writes the rest of the ungapped extensions to the level-2 buffer. Such a hierarchical buffering design can efficiently utilize the global memory space as well as avoid unnecessary synchronization overhead.

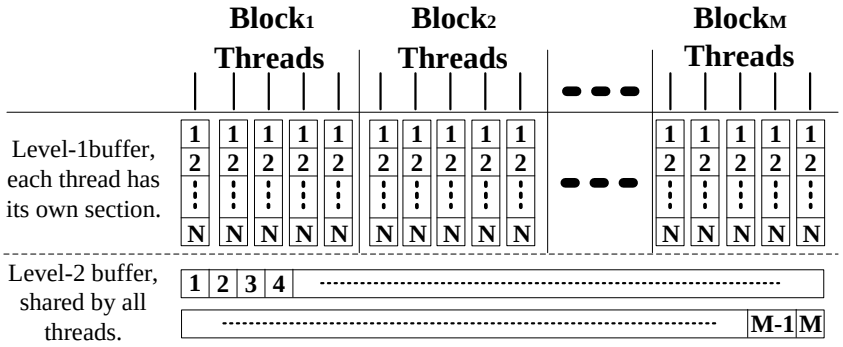


Figure 6.3: Ungapped Extension Storage in Global Memory

In the first two stages, the BLASTP algorithm also needs to maintain several global

[1]Note that dynamic memory allocation was not supported on NVIDIA GPUs when we were working on this project.

counters, such as the number of hits detected. One way to implement a global counter is to have each thread update a shared variable with atomic add operations. Again, to avoid the overhead of atomic operations, we implement per-thread counters on GPU, and all the per-thread counters will be added up together on the host side to produce the correct value for a global counter.

Algorithm 1 Ungapped Extension Storage in Global Memory

```
1: shrdIndex ← 0
2: privtIndex ← 0
3: B ← privtBufSize
4: ..., an ungapped extension is found ...
5: if privtIndex < B then                          ▶ level-1 private buffer
6:     bufPtr ← privtBuf + privtIndex
7:     privtIndex ← privtIndex + 1
8: else                                            ▶ level-2 shared buffer
9:     bufPtr ← shrdBuf + atomicAdd(shrdIndex, 1)
10: end if
11: bufPtr ← unExtPtr
12: ...
```

6.2.3.3 Gapped Alignment Parallelization

Gapped alignment uses seeds created by the ungapped extension stage as its inputs and creates alignments with even higher alignment scores. Typically only a small percentage of database sequences will need to be processed with gapped alignment. As such, we launch a separate kernel for this stage to re-map tasks to individual threads. To minimize data transferring overhead, the gapped alignment kernel reuses the subject sequence data stored on the GPU during the first two stages.

During the gapped alignment, the best alignment score corresponding to each subject sequence is recorded, which will be copied to the host memory to filter out dissimilar

subject sequences. In addition, the status of each extension will be recorded and copied back to the host memory for further processing in the final stage – gapped alignment with traceback.

6.2.4 Performance Optimization

Because of its heuristic nature, the BLASTP algorithm is highly irregular with respect to the memory access and execution path. As such, the basic algorithmic mapping described in Section 6.2.3 does not map well onto the GPU architecture. In this section, we present several optimization techniques to address some of the performance hurdles of accelerating BLASTP on GPUs.

6.2.4.1 Memory Access

The BLAST search needs to access a number of different data structures. To improve memory access efficacy, we explore different data placement strategies in the GPU memory hierarchy.

1. *Constant Memory to Store the Query Sequence and Scoring Matrix*: When calculating alignment scores, the BLASTP algorithm needs to frequently compute a matching score between two individual letters from the query and the subject sequences, which can be done by looking up the corresponding element in a scoring matrix. FSA-BLAST optimizes the lookup performance by pre-computing a *query profile* for each query sequence. Specifically, a query profile is a two-dimensional matrix, where each column corresponds to one letter in the query sequence and consists of matching scores between the query letter to all other letters. With the query profile,

the matching score between two letters can be obtained by first finding the column corresponding to the letter in the query sequence and then reading the score from the column according to the letter in the subject sequence. Using query profiles is more efficient because it saves one memory access used to read the current letter of a query sequence.

One common optimization technique in GPU programming is to leverage the cached constant memory to speedup the access of frequently used data. Compared to global memory, constant cache has about two orders of magnitude lower access latency for *cache hits*, but its size is small (64KB). In the FSA-BLAST implementation, each column (corresponding to a letter in the query sequence) in the query profile consumes 64 bytes (32 elements with 2 bytes each). As such, the constant memory is not sufficient to store the query profile for a query sequence larger than 1K letters.

To take advantage of the constant memory, our implementation instead puts the scoring matrix there because it has a fixed size and is accessed by all threads. In fact, the scoring matrix used in BLASTP consists of $32 \times 32 = 1024$ elements and has a total size of only 2KB (2 bytes per element). However, the query sequence needs to be available when using the scoring matrix. In our implementation, a 60K-byte buffer is allocated in the constant memory for its storage. Since one byte is needed for each letter, the maximum query sequence that can be supported is 60K letters. By counting the sequence size in the most recent NCBI `NR` database, we found that more than 99.95% of the sequences are smaller than 4K letters and the largest sequence contains 36,805 letters, suggesting that the 60K-byte buffer is sufficient for storing individual protein sequences in the most recent NCBI `NR` database.

2. *Texture Memory to Store Subject Sequences and the Word Lookup Table:* Texture memory is another type of cached memory on the GPU but with a much larger size than constant memory. For example, with the one-dimensional texture memory, the number of elements that the texture memory can bind to is 2^{27} or 128M elements. In order to take advantage of the texture cache, which has low access latency for cache hits, we partition the database into different chunks of 128MB each. By loading the database chunks on the GPU one after another, our design can search database of an arbitrary size, which is important for solving real-world BLAST search problems in practice.

   Storing database sequences in texture memory may also help exploit data locality in alignment computation. For instance, as shown in Figure 6.4, when a hit is found, ungapped extension will be performed in both directions. With subject sequences stored in texture memory, some portions of the subject sequence may have been cached, thus improving the memory access efficiency.

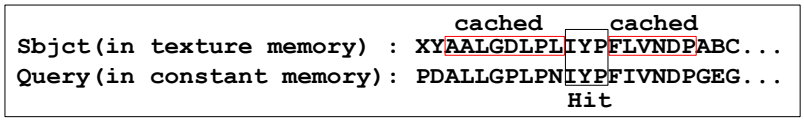


Figure 6.4: Texture Memory Usage for Subject Sequences

   Besides subject sequences, we also store the *word lookup table* in the texture memory. A word lookup table stores precomputed words that can result in hits to each word in the query profile. Again, the size of the word lookup table varies depending on the query size, and it cannot fit into the constant memory for reasonably long query sequences.

6.2.4.2 Load Balancing across Different Threads

When scheduling alignment tasks to different threads in a kernel, a straightforward implementation can be statically assigning a set of sequences to each thread according to the thread ID number. This approach is easy to implement. However, the overall kernel execution time may suffer when there is load imbalance between different threads.

Algorithm 2 Dynamic Subject Sequence Assignment Algorithm

1: $n \leftarrow totalThreadNum$
2: $mutexIndex \leftarrow n$
3: $seqIndex \leftarrow threadID$
4: **while** $seqIndex < numSequences$ **do**
5: $\quad AlignSeq(SubSeq[seqIndex], querySeq)$
6: $\quad ...$
7: $\quad seqIndex \leftarrow atomicAdd(mutexIndex, 1)$
8: **end while**

To alleviate the load-imbalance issue, our implementation adopts a greedy algorithm (as shown in Algorithm 2) that dynamically assigns sequences to different threads. Specifically, the first sequence is assigned to each thread according to the thread ID. Whenever a thread finishes its current assignment, it retrieves the next subject sequence using an atomic operation. In addition, the database sequences will be presorted in the descending order of the sequence lengths. Assuming the BLAST search time is proportional to the length of a subject sequence, the database sorting can alleviate the impact of load imbalance occurred toward the end of the kernel execution. Note that this approach is only applicable to threads in different warps, because threads within a warp always execute the same instructions.

6.2.5 Performance Evaluation and Characterization

In this section, we evaluate the performance of our parallel BLASTP implementations on the GPU. Our experiments focus on five versions of BLASTP with different optimization techniques applied. Version 1 is the basic parallel version as described in Section 6.2.3. Each of the other four versions is corresponding to an optimization technique discussed in Section 6.2.4. The five versions are listed in Table 6.2.

Table 6.2: Versions of GPU BLASTP

Versions	Description
Version 1	It is a straightforward mapping as described in Section 6.2.3.
Version 2	Constant memory is used as described in Section 6.2.4.1.
Version 3	Based on Version 2; atomicAdd is called for load balancing as described in Section 6.2.4.2.
Version 4	Based on Version 2; texture memory is used as described in Section 6.2.4.1.
Version 5	Based on Version 4; atomicAdd is called for load balancing as described in Section 6.2.4.2.

Our experiments are executed on both the NVIDIA Tesla C1060 and C2050 GPU cards. The Tesla C1060 GPU consists of 30 SMs, each containing 8 scalar processors. On each SM, there are 16K registers and 16KB shared memory. There is 4GB of global memory on the Tesla C1060 with an aggregate bandwidth of 102.4GB/s. The Tesla C2050 is a newer-generation GPU from NVIDIA. Compared to the Tesla C1060, the Tesla C2050 has more scalar processors (i.e., 32) per SM and larger register files (i.e., 32K registers). The Tesla C2050 has a L1 cache for each SM and a L2 cache shared by all SMs. Both L1 cache and shared memory use the same on-chip memory, which can be configured as 16

KB L1 cache and 48KB shared memory or as 48KB shared memory and 16 KB L1 cache. The L2 cache has a larger size of 768KB. There is 3GB global memory with an aggregate memory bandwidth of 153.6GB/s on the Tesla C2050. We will refer the Tesla C1060 and C2050 as *Tesla* and *Fermi*, respectively, hereafter.

On the host side, the system has an Intel Core 2 Duo CPU with 2.2GHz clock speed and 4GB DDR2 SDRAM memory. The operating system is the Ubuntu 8.04 GNU/Linux distribution. Our code is developed with the CUDA 3.1 toolkit.

According to experimental results using databases with different sizes, the speedup achieved is quite stable regardless of the database size. Thus, we use a subset of the NCBI `NR` database with one sequence selected out of every 5 sequences. In this way, we can save a lot of time to run our experiments without hurting the correctness of our experimental results. Also, a sequence with 1K letters is used as the query sequence.

For the BLASTP program, all default values are used for the program execution as shown in Table 6.3. In addition, the score matrix `BLOSUM62` is used. The serial CPU version is compiled with `gcc` with the `-O3` optimization option. We report the average of three runs for each experiment.

Table 6.3: Default Parameter Values in BLASTP

Parameter description	Value
Word size	3
Dropoff value for ungapped extension	7
Dropoff value for gapped extension	15
Dropoff value for triggering gaps	22
Dropoff value for final gapped alignment	25
Open gap penalty	-7
Extension gap penalty	-1

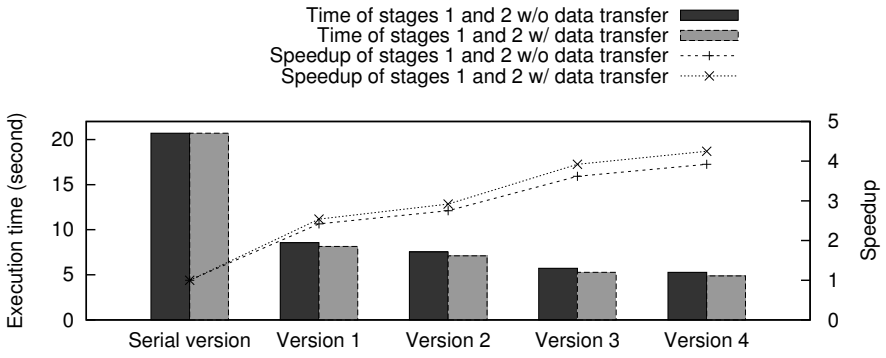
6.2.5.1 Evaluation of Individual Optimization Techniques

In this section, we evaluate the performance impact of the individual optimization techniques described in Section 6.2.4 with respect to the execution time spent on various compute stages.

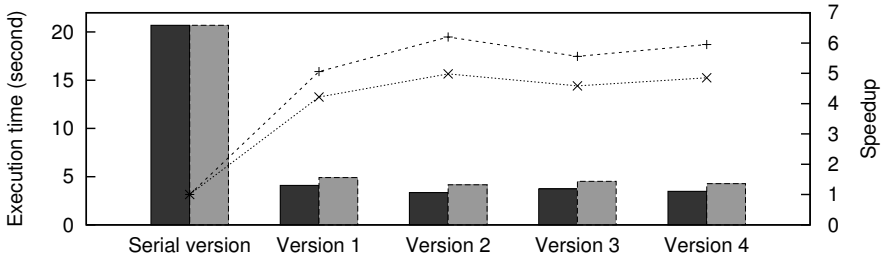
1. *Hit Detection + Ungapped Extension:* Figure 6.5 shows the execution time of the first two stages for the five versions as described in Table 6.2 as well as the baseline serial CPU version. We measure the kernel execution time and total execution time of the first two stages, where the total execution time includes data transfer time between host memory and device memory, pre/post-processing time, and kernel execution time.[2] We also calculate the speedups of various GPU versions against the CPU version.

   Figure 6.5(a) shows the results on the Tesla card. Clearly, performance improvement is achieved when each optimization technique is applied. Specifically, the kernel execution time is 8.148s in Version 1, while that of Version 2 is 7.098s, which means a performance improvement of 12.89% is achieved when the query sequence and scoring matrix are stored in the constant memory. With the load-balancing optimization added, i.e., Version 3, the kernel execution time is further reduced to 6.422s, a 9.51% improvement compared to Version 2. Version 4 extends Version 2 by placing the subject sequences and the word lookup table in the texture memory, resulting a 25.55% performance improvement. Finally, the best performance is achieved in Version 5, where load balancing is added as compared to Version 4. For the total

[2] Since there is no data transfer, pre-processing, and post-processing in the serial CPU version, the kernel execution time and the total execution time is the same for the serial CPU version.



(a) Tesla Results



(b) Fermi Results

Figure 6.5: Performance Improvement Brought by each Optimization Technique and the Corresponding Speedup for the First Two Stages

execution time, the absolute differences between different versions are the same as those for the kernel execution time. With each optimization technique applied, the relative performance improvement is 12.18% (Versions 1 to 2), 24.04% (Versions 2 to 3), 9.29% (Versions 2 to 4), and 7.56% (Versions 4 to 5), respectively.

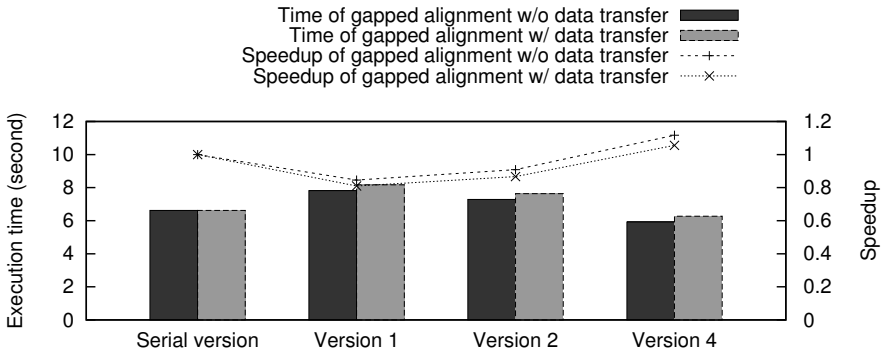
On Fermi, similar trends are observed for Versions 1, 2 and 3 as compared to the Tesla results. However, storing the subject sequences and the word lookup table in the texture memory has an adverse performance impact on Fermi. Specifically, Version 4 is 20.04% slower than Version 2. This can be caused by the L1/L2 cache

introduced in the Fermi architecture. Nonetheless, with load balancing added, Version 5 outperforms Version 4 by 6.80%, similar to what can be observed on Tesla.

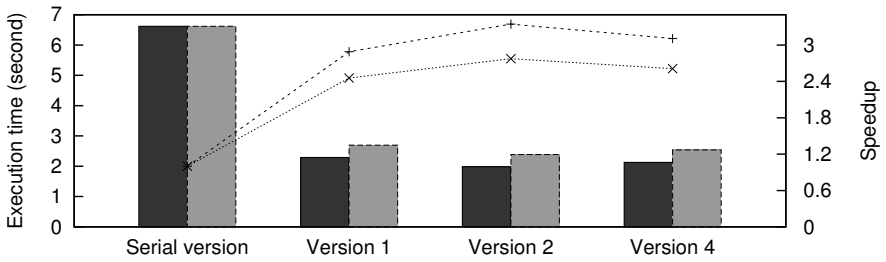
Relative to the CPU serial version, the best GPU versions achieve speedups of 4.25 and 7.28 on Tesla and Fermi, respectively, in kernel execution time. For the total execution time, the speedups are 3.92 and 5.69 on Tesla and Fermi, respectively. As expected, the program performance on Fermi is better than that on Tesla. One reason is that the first two stages of BLAST are memory-bound, and Fermi has more global memory bandwidth than Tesla. Moreover, there are L1 and L2 caches on the Fermi card, which can improve memory performance further.

2. *Gapped Alignment:* This section shows the performance improvement of the gapped alignment stage. There are three different GPU implementations for the parallelization of the gapped alignment, which are Versions 1, 2, and 4 as described in Table 6.2.

   As shown in Figure 6.6(a), for Tesla, performance improvement is observed with each optimization technique applied, similar to what is observed for the first two stages. When constant memory is used for the query sequence and the scoring matrix, the performance of the gapped alignment improves by 6.87% and 6.60% for the kernel and total execution time, respectively. Storing the subject sequences in the texture memory helps to reduce the kernel and total execution time by 18.68% and 17.87%, respectively. Unfortunately, there is no performance improvement for the gapped alignment on the Tesla GPU, when compared to the CPU serial implementation. One reason can be that the irregular memory access is poorly supported on Tesla. Another reason can be the large number of divergent branches in the gapped



(a) Tesla Results



(b) Fermi Results

Figure 6.6: Performance Improvement Brought by each Optimization Technique and the Corresponding Speedup for the Gapped Alignment Stage

alignment code.

Figure 6.6(b) shows the performance results on Fermi. With constant memory used to store the query sequence and the scoring matrix, the kernel execution time decreases from 2.291s (Version 1) to 1.979s (Version 2), corresponding to a 13.61% improvement. However, if the subject sequences are stored in the texture memory, the kernel execution time increases by 7.63% to 2.130s (Version 4). Same as the first two stages, the reason is that global memory access is more efficient on Fermi because of the L1/L2 cache provided. The best GPU version achieves 3.34-fold and 2.77-fold speedup for kernel and total execution time, respectively, as compared to

the serial version on the CPU.

On both Tesla and Fermi, the kernel execution time occupies a majority (more than 80%) of the total execution time in each stage, as shown in Table 6.4.

Table 6.4: Percentage of the Kernel Execution Time

Stage(s)	Tesla	Fermi
Hit detection + ungapped extension	92.08%	80.36%
Gapped alignment	94.53%	84.15%

3. *Overall Execution Time:* In this experiment, we compare the overall execution time for the following five different implementations.

   - CPU serial implementation.
   - *Version G1:* All three stages are executed on the GPU.
   - *Version G2:* The first two stages are executed on the GPU, and the third stage is serially executed on the CPU. There is no overlap between the CPU and GPU processing.
   - *Version G3:* The first two stages are executed on the GPU, and the third stage is executed on the CPU in parallel with two threads. No overlap exists between the CPU and GPU processing.
   - *Version G4:* The first two stages are executed on the GPU, and the third stage is executed on the CPU in parallel with two threads. The CPU and GPU processing is overlapped.

Figure 6.7 shows the overall execution time of the above five implementations. There are several observations we can make from Figure 6.7. First, if all three stages

are executed on the GPU, the overall performance on Fermi is much better than that on Tesla (by 1.93 times). Second, with GPUs used for only the first two stages, on Fermi, if no overlap is used (Versions G2 and G3), the overall performance (9.92s and 7.00s for Versions G2 and G3, respectively) will be worse than that of Version G1 (6.10s). On the other hand, if we overlap the computation on the CPU and the GPU, the overall performance (4.98s) will become better than Version G1 (6.10s) by 18.29%. Third, on Tesla, since there is almost no performance improvement by parallelizing the gapped alignment on the GPU, the overall performance can be improved by either using multiple threads to accelerate the gapped alignment on the CPU (Version G3) or overlapping its execution on the CPU with the execution of other stages on the GPU (Version G4). For instance, if the `pthread` is used for the parallelization, we can reduce the execution time by 27.79% (from Version G2 to Version G3). Furthermore, if the CPU and GPU execution is overlapped, this performance improvement can almost be doubled, i.e., the execution time decreases from the 11.89s of Version G1 to 5.95s of Version G4.

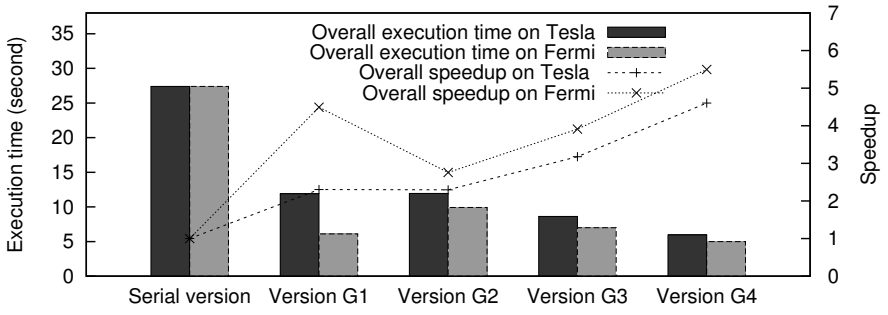


Figure 6.7: Overall Execution Time

6.2.6 Multiple GPUs for Hit Detection and Ungapped Extension

In this section, we show the speedup brought by using multiple GPUs based on the VOCL framework. Since VOCL is based on the OpenCL programming model, we first translate the CUDA implementations to OpenCL, then we run the OpenCL implementations by using up to 4 GPUs. Figure 6.8 shows the speedup of 1, 2, and 4 GPUs, where time and speedup of the first two stages as well as that of the total execution time are shown.

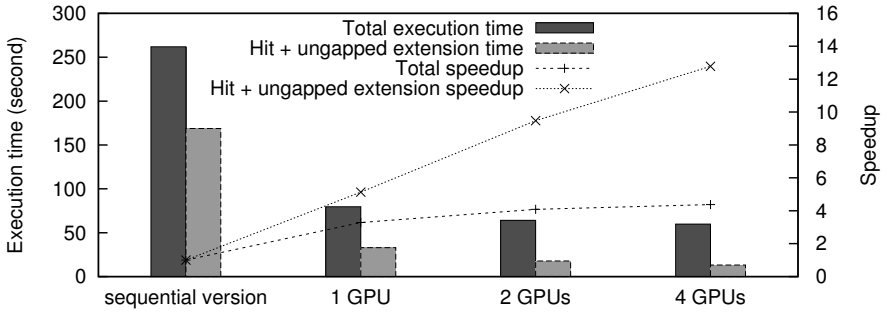


Figure 6.8: Execution Time and Speedup with Multiple GPUs used.

As shown in Figure 6.8, with more GPUs used, less execution time is needed to search the same sequence database. Specifically, speedup of the first two stages is 5.13 times if one local GPU is used. When 2 and 4 GPUs are used, speedups of the first two stages are 9.48-fold and 12.78-fold, respectively. From these results, the performance does not scale linearly when more GPUs are used concurrently. The reason is the serialization of data copy across different GPUs, which takes the same amount of time no matter how many GPUs are used to search the database. The more GPUs that are used, the less percentage of time that will be used by the kernel execution, because the data transfer will consume a larger percentage of time.

As for the overall speedup, it is 3.29-fold, 4.09-fold, and 4.38-fold with 1, 2, and 4 GPUs used, respectively, which is much less than that of the first two stages. The reason is that the third stage is executed on the CPU (parallelized with `pthread` using 2 threads), which takes the same amount time no matter how many GPUs are used. With more GPUs used to accelerate the first two stages, total execution time is dominated by the third stage and the overall speedup does not increase very much.

6.2.7 Summary

In this work, as a complement of the existing parallel BLAST implementations in multi-core and distributed systems, we parallelize BLAST on GPUs to accelerate its execution. We found that there are many irregularities in both the computation and memory accesses for the execution of BLAST on GPUs, which should be overcome as much as possible to achieve good performance. To address the irregularities and improve performance, we propose techniques including storing query sequences and the scoring matrix in the constant memory, using texture memory to cache subject sequences and the word lookup table, and dynamically assigning sequences to threads to achieve good load balance. Moreover, we overlap the first two stages on the GPU and the third stage on the CPU, which is parallelized with `pthread`, and better performance is achieved than by executing all three stages on the GPU. Compared to the serial CPU implementation, our parallel implementation achieves more than a 5-fold speedup for overall performance if one GPU is used. Finally, we translate the CUDA implementation to OpenCL and show the performance improvement brought by concurrently using GPUs based on the VOCL framework. We achieved more than a 12-fold speedup for the first two stages when using 4 GPUs to search

a sequence database concurrently.

6.3 Parallelization of Smith-Waterman

6.3.1 Algorithm Description

The Smith-Waterman application [73] is an *optimal local sequence alignment* methodology that follows the dynamic programming (DP) paradigm, where intermediate alignment scores are stored in a matrix. After alignment scores are calculated in the DP matrix, the matrix entries are inspected, and the highest-scoring local alignment is generated. Thus, the Smith-Waterman algorithm can be broadly classified into two phases: (1) matrix filling and (2) backtracing.

To fill out the DP matrix, Smith-Waterman utilizes a scoring system that includes a scoring matrix and a *gap-penalty* scheme. The scoring matrix, M, is a two-dimensional matrix containing scores for aligning individual amino acids or nucleotide residues. The *gap-penalty* scheme, on the other hand, provides the option of gaps being introduced within the alignments to generate a better alignment score. With the scoring scheme and the affine-gap penalty, the DP matrix is filled in a *wavefront* pattern, which starts from the upper left corner of the DP matrix and moves toward the bottom right corner, as shown in Figure 6.9. As can be seen, each element in the DP matrix depends on its north, west and northwest neighbors, and can be expressed as Equations (6.1), (6.2), and (6.3):

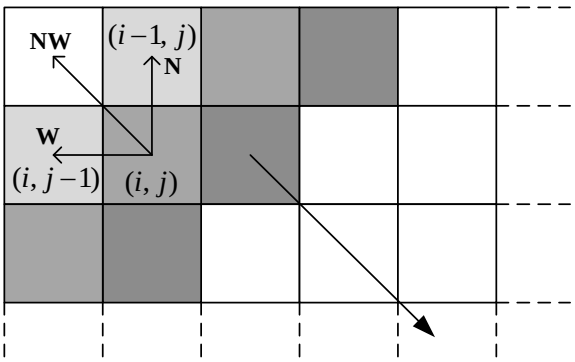


Figure 6.9: Wavefront Pattern and Dependency in the Matrix Filling Process.

$$N_{i,j} = \max \begin{cases} 0 \\ N_{i-1,j} - e \\ NW_{i-1,j} - o \end{cases} \tag{6.1}$$

$$W_{i,j} = \max \begin{cases} 0 \\ W_{i,j-1} - e \\ NW_{i,j-1} - o \end{cases} \tag{6.2}$$

$$NW_{i,j} = \max \begin{cases} 0 \\ W_{i,j} \\ N_{i,j} \\ NW_{i-1,j-1} + d(i,j) \end{cases} \tag{6.3}$$

where $d(i, j)$ is the alignment score of individual characters i and j, $N_{i,j}$ stores a score depending only on its north neighbor, $W_{i,j}$ stores a score depending only on its west neighbor, and $NW_{i,j}$ stores the maximum score of element (i, j). In the affine-gap penalty, the open-gap penalty o is usually larger than the extension-gap penalty e, which is to introduce as fewer gaps as possible for the alignment.

From Figure 6.9, elements on the same anti-diagonal can be calculated in parallel; but their calculation depends on elements of its previous anti-diagonals and barrier synchronization is needed across the computation of different anti-diagonals.

The backtracing phase of the algorithm is essentially a sequential operation that generates the highest scoring local alignment. In this work, we mainly focus on accelerating the matrix filling because it consumes more than 99% of the execution time and it is the object to be parallelized.

6.3.2 Related Work

The Smith-Waterman algorithm has previously been implemented on the GPU by using graphics primitives [50,51], and more recently, using CUDA [54,57,75]. Though the most recent CUDA implementations of Smith-Waterman [54,57,70] report speedups as high as 30-fold, they all suffer from a myriad of limitations. First, each of their approaches only follows a coarse-grained, embarrassingly parallel approach that assigns a single problem instance to *each* thread on the device, thereby sharing the available GPU resources among multiple concurrent problem instances. This approach *severely* restricts the maximum problem size that can be solved by the GPU to sequences of length 2048 letters or less. In contrast, we propose and implement a fine-grained parallelization of Smith-Waterman by distributing the task of processing a single problem instance *across all the threads* on the GPU, thereby supporting realistic problem size, as large as 8K letters in lengths on the GTX 280. The above limitation is due to the physical size of global memory on the GPU.

Though the global memory can support the alignment of large sequences, the coarse-grained parallel approach forces the global memory to be shared amongst multiple instances of Smith-Waterman whereas our fine-grained parallel approach leaves the entire global memory resource available to a single instance of Smith-Waterman, allowing larger sequences to be processed. Striemer et al. [75] primarily use shared memory and constant cache for the coarse-grained Smith-Waterman parallelism on the GPU. Thus, their implementation is limited to query sequences of length 1024 or less because of the limited shared memory and cache size. Furthermore, all previous implementations only compute the optimal alignment score, as it is the data-parallel portion of Smith-Waterman, and ignore the actual generation of the sequence alignment. In contrast, we provide a *complete* solution, i.e., compute the optimal alignment score *and* output the actual sequence alignment.

6.3.3 Analysis of Smith-Waterman Execution on the GPU

In this work, we parallelize Smith-Waterman with a *fine-grained* parallel approach to support the alignment of large sequences, which can cover nearly all sequences within the NIH's GenBank. This approach utilizes the parallelism that exists across elements on the same anti-diagonal. As shown in Figure 6.9, each element depends on its north, west, and northwest neighbors, which are in its immediate previous two anti-diagonals. Thus, there is no data dependency across elements on the same anti-diagonal and these elements can be calculated in parallel. However, data dependency exists between different anti-diagonals, so computation of each anti-diagonal should be performed after its previous anti-diagonals are completed.

The GPU has a multi-level memory hierarchy, in which global memory has a large size,

but its access latency is high. In our fine-grained parallel approach, the DP matrix is stored in global memory due to the $O(n^2)$ (n is the size of the sequences to be aligned.) space requirement and the support of large sequence alignment. When the DP matrix is filled, according to Equations (6.1)–(6.3), three alignment scores ($N(i,j), NW(i,j), W(i,j)$) are calculated for each element (i,j). To calculate these three scores, four scores, $N_{i-1,j}$, $NW_{i-1,j}$, $W_{i,j-1}$, and $NW_{i,j-1}$ are loaded from its immediate previous anti-diagonal and one score $V_{i-1,j-1}$ is loaded from the next previous anti-diagonal, which are 20 bytes in total. Also, three scores are stored with 12 bytes. Finally, to get the individual letter score $d(i,j)$, two letters (one byte for each) are loaded from the input sequences and the corresponding score is read from the individual score matrix (4 bytes). Thus, there are 38 bytes of memory access for the computation of each element. As for the arithmetic operations, according to Equations (6.1)–(6.3), there are 12 operations in total. On a Tesla C2050 GPU card, the instruction issue rate is 515.2 Giga instructions per second (Ginst/s), and the global memory bandwidth is 153.6 GB/s. Thus, as a rule of thumb, a kernel will likely be memory-bound if the ratio of instructions-to-bytes loaded from global memory is smaller than $515.2/153.6 = 3.4$. Otherwise, it is computation-bound. As mentioned above, 38 bytes are accessed and 12 arithmetic operations are executed for the calculation of each element. Thus, the ratio of arithmetic instructions-to-bytes is $12/38 = 0.315$, which is much less than the threshold. By disassembling the kernel with an internal tool from NVIDIA, there are 106 instructions for the calculation of each element. So, with all instructions considered, the ratio of instructions-to-bytes is 2.78. It is still less than the threshold 3.4 and the matrix filling is memory-bound as a result.

In the following, we will mainly focus on improving the efficacy of memory access

to accelerate the matrix filling. On the other hand, since different anti-diagonals should be filled sequentially and data communication is required across different threads, barrier synchronization is needed after the computation of each anti-diagonal. Currently, barrier synchronization is implemented via multiple kernel launches, which are expensive operations and synchronization using multiple kernel launches can occupy up to 50% of the total matrix filling time as shown in Table 4.1. This problem becomes even more serious when remote GPUs are used as shown in Figure 3.16(c). To reduce the synchronization overhead, we integrate the *GPU synchronization* approaches into the parallel implementation on the GPU to reduce the time of switching back and forth between the host and the device. Finally, we move the trace back to the GPU to reduce the data transfer time from device memory to host memory. In the following, each of the above optimizations is described in detail.

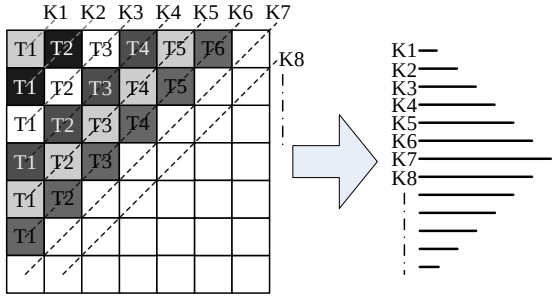
6.3.4 Techniques for Efficient Memory Access and Data Copy

In this section, techniques to improve memory access and to reduce data copy time between host memory and device memory are described. Efficient memory access includes access to the DP matrices, input sequences, and the scoring matrix. Specifically, we improve the efficacy of the DP matrix access via coalesced memory access. As for the input sequences and the scoring matrix, we store them in the constant memory to take the advantage of constant cache. When the matrix filling is parallelized on the GPU, data copy from host memory to device memory is needed. For the trace back, the amount of data that is transferred from device memory to host memory depends on where the trace back is executed (host or device). In the following, each optimization is described in detail.

6.3.4.1 Efficient DP Matrix Access

DP matrices are stored in the global memory in our fine-grained parallel implementations. From the CUDA programming guide [63], *coalesced memory access* is an important approach to improve the efficiency of global memory access. For example, on a Tesla C2050, if threads within a warp access a *memory segment* that is aligned to its size (e.g., we use the `float` data type in the DP matrix, then each memory segment is of 128 bytes, and the first address should be a multiple of 128.) [63], data transfer for threads within the warp can be combined into one data transaction, which can significantly improve the data access performance considering the large bandwidth and high access latency of global memory. For the DP matrices, we first store them in the same way as that in the host memory, i.e., *row-major* data format. Then, we re-align the matrix layout to make elements of the same anti-diagonal stored in adjacent locations. In other words, the DP matrices are stored in a *diagonal-major* data format. Next, based on the diagonal-major data format, some buffers are padded at the end of each anti-diagonal (if necessary) to make all global memory stores coalesced. Finally, we use the shared memory to cache the DP matrices based on a *tiling* approach.

1. *Direct Parallelization of Smith-Waterman on a GPU*: One direct way to parallelize the matrix filling is to move the DP matrices from host memory to device memory, and then launch one kernel for each anti-diagonal from the top left to bottom right. In this way, elements on the same anti-diagonal are calculated in parallel, and different kernels are computed sequentially. In this implementation, the DP matrices are stored in the same way as that in the host memory, i.e., a *row-major* data format as shown in Figure 6.10(a). To load balance on each SM, computation of elements



(a) Mapping of Threads (T_i) to Matrix Elements

(b) Computational load for each kernel

Figure 6.10: Naïve (Direct) Mapping of the DP Matrix and Computational Load Imposed on Successive Kernels.

on each anti-diagonal is distributed uniformly across all the threads in a kernel. To simplify our work, every kernel contains a one-dimensional grid of blocks, in which threads are organized in one dimension, too, as in Figure 6.10(b). Note that this implementation is referred to as the *naïve implementation*, hereafter.

This approach is straightforward and easy to implement. However, as shown in Figure 6.10(a), since each element on an anti-diagonal is in a different memory segment if the sequence size is larger than 32 bytes,[3] all global memory accesses are uncoalesced. Thus, the performance of the naïve implementation is expected to be poor since matrix filling on the GPU is memory-bound. To improve the DP matrix access efficacy, we propose the techniques of *matrix layout re-alignment*, *coalesced matrix access*, and *tiling matrix access via shared memory*, which are described in the following three subsections, respectively.

2. *Matrix Layout Re-Alignment*: In the naïve implementation, access to the DP matrices is totally uncoalesced. One approach to improve the access efficiency is making

[3]For sequences containing more 32 letters, each row in Figure 6.10(a) has more than 33 floats (i.e., 132 bytes), then elements on the same anti-diagonal are in different memory segments.

elements on the same anti-diagonal into the fewest memory segments as possible. This can be achieved by storing the DP matrices in a *diagonal-major* data format, as shown in Figure 6.11. As can be seen, the row-major data format in Figure 6.11(a) is transformed to the diagonal-major data layout, as in Figure 6.11(b), where elements on the same anti-diagonal are stored in the same "row" in adjacent locations. With the *diagonal-major* data format, elements that are calculated by threads in a warp are in at most two memory segments. Thus, fewer data transactions are needed for filling the same DP matrix, which is verified later via the CUDA profiler [61]. This implementation is called the *simple implementation*.

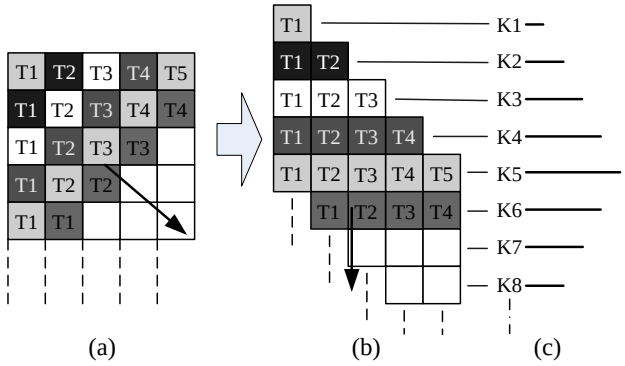


Figure 6.11: Matrix Re-Alignment and Computational Load Imposed on Successive Kernels.

With the diagonal-major data format, the DP matrix is filled in the same way as the naïve implementation. In other words, each kernel launch is responsible for elements on one anti-diagonal (i.e., one "row" in Figure 6.11(b)). Barriers across different anti-diagonals are implemented via different kernel launches. Similarly, every kernel contains a one-dimensional grid of blocks, and threads in a block are organized in one dimension as shown in Figure 6.11(c).

3. *Coalesced Matrix Access*: Compared to the row-major format, matrix access is much more efficient with the diagonal-major format because elements accessed by threads in a warp are in adjacent locations. However, it is possible that elements accessed by threads in a warp are still in different memory segments. The eighth anti-diagonal in a DP matrix (assume the matrix has more than eight anti-diagonals) is such an example, as shown in Figure 6.12. In our implementations, we use the `float` type to store alignment scores, and the memory segment size is 128 bytes, i.e., 32 elements. For the eighth anti-diagonal, index of the first element is 28, and there are 8 elements on this anti-diagonal, so elements with index less than 32 are in a different memory segment from the others.

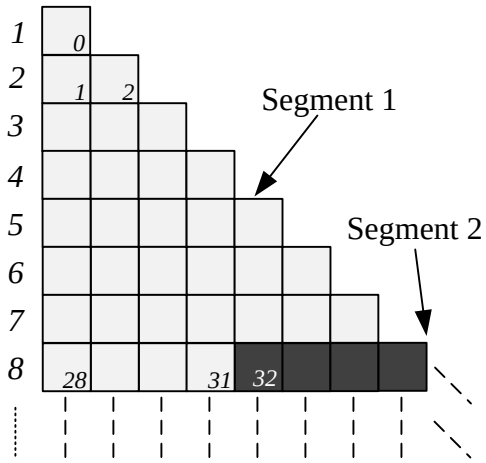


Figure 6.12: An Example that Threads in a Wrap Access Two Memory Segments.

To remove such cases mentioned above, we propose the *coalesced matrix access* approach (referred to as *coalesced implementation* later), which is implemented by adding buffers at the end of each anti-diagonal (if necessary). Specifically, we store the DP matrices with the diagonal-major format and make the starting address of each anti-diagonal 128-byte aligned by adding padding, as shown in Figure 6.13.

(This figure shows a 16-byte aligned case because of the figure size limit. A 128-byte aligned case can be done in the same way, except more padding is needed.) In this way, all data stores are coalesced. Unfortunately, because of the skewed dependence between the elements across adjacent anti-diagonals, as shown in Figure 6.13, loads from the global memory cannot be totally coalesced.

As can be seen in Figure 6.13, a larger memory space is needed to store the same DP matrix in the coalesced implementation. Consider a DP matrix with the same number of rows and columns n, since the number of elements on each anti-diagonal should be the multiple of 32, the number of padding elements is 16 per anti-diagonal on average. So the relative increase of the DP matrix size is

$$\frac{16\,(2n-1)}{n^2} \tag{6.4}$$

From Equation (6.4), the larger the input sequences are, the less the relative increase will be. For example, for input sequences of 1K letters, the DP matrix size is increased by only 3.2%. Thus, the coalesced implementation does not cause very much overhead in global memory usage.

To demonstrate the benefits of each of the proposed techniques, we run the CUDA profiler and show the number of data transactions for filling the DP matrices of the three implementations in Figure 6.14, where 7K-letter sequences are used as the inputs. From Figure 6.14, the number of data transactions of the simple implementation is greatly decreased compared to the naïve implementation; while the decrease from the simple implementation to the coalesced implementation is negligible. This is because most of the global memory accesses in the simple implementation are already coalesced, leaving little room for additional improvement.

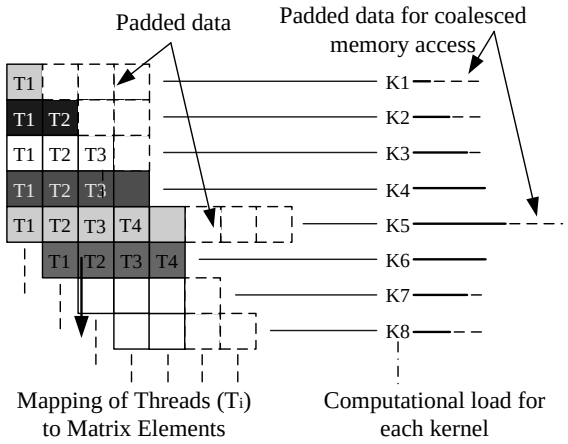


Figure 6.13: Incorporating *Coalesced* Data Representation of Successive Anti-Diagonals in Memory.

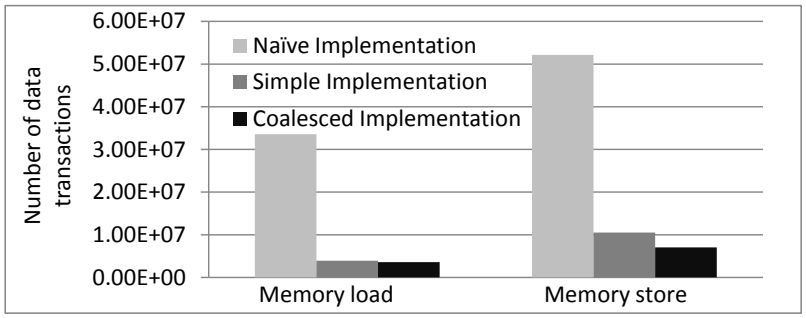


Figure 6.14: Number of Data Transactions to Global Memory.

4. *Tiling Matrix Access via Shared Memory*: In the previous subsections, we propose techniques for efficient global memory access. Here we apply the tiled-wavefront design pattern, which was used to efficiently map Smith-Waterman to the IBM Cell/BE [5], to the GPU architecture. This approach amortizes the cost of kernel launches by grouping the matrix elements into computationally independent *tiles*. In addition, shared memory is used as a bridge to improve the performance of global memory access by avoiding uncoalesced memory access.

   Our tile-scheduling scheme assigns a thread block to compute a tile, and a grid of

blocks (kernel) is mapped to process a single *tile-diagonal*, thus decreasing the number of inter-block barriers needed to fill the DP matrix. CPU implicit synchronization via new kernel launches or our proposed GPU synchronization serve as barriers to threads from the previous tile-diagonal. Consecutive tile-diagonals are computed one after another from the top left corner to the bottom right corner of the matrix in the design pattern of a *tiled wavefront*, as shown in Figure 6.15.

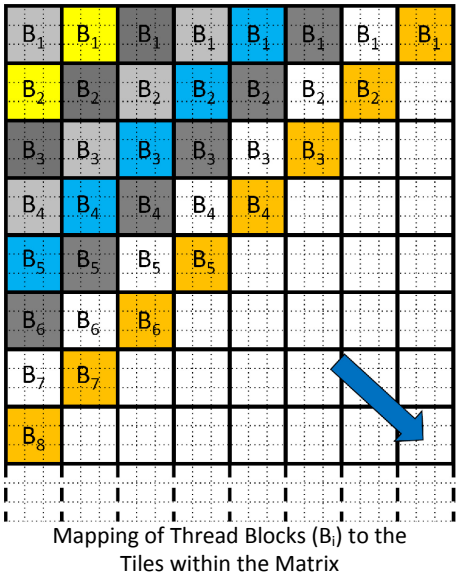


Figure 6.15: Tiled Wavefront.

The elements *within* a tile are computed by a thread block by following the simple wavefront pattern, starting from the top left element of the tile. The threads within each block are synchronized after computing every anti-diagonal by explicitly calling CUDA's local synchronization function `__syncthreads()`.

Each thread block computes its allocated tiles within shared memory. The processed

tile is then transferred back to the designated location in global memory. This memory transfer will be coalesced because we handcraft the allocation of each tile to follow the rules for coalesced memory accesses.

6.3.4.2 Efficient Access to Scoring Matrix and Input Sequences

Constant memory is used to improve the efficacy of scoring matrix and input sequence accesses. There are three reasons for doing this. First, compared to the DP matrices, the scoring matrix and input sequences are much smaller in size. In our implementation, the maximum sequence size that can be supported on a Tesla C2050 is 13K letters and the largest space for input sequences is 26K bytes (one byte per letter). On the other hand, the scoring matrix has a fixed size. It contains $23 \times 23 = 529$ elements for the total size 2116 bytes. Thus less than 30KB are needed for the scoring matrix and input sequences in total, which is far less than the size of constant memory. Second, both the input sequences and the scoring matrix are read-only and accessed by all the threads in the kernel, which matches well with what the constant memory is designed for. Third, with some data accessed from the constant memory, requirements of global memory bandwidth can be reduced.

6.3.4.3 GPU Synchronization

Efficient global memory accesses can accelerate the computation, but it cannot accelerate the synchronization on the GPU. While the tiled-wavefront technique reduces the number of kernel launches, it explicitly and implicitly serializes the computation both within and across tiles, respectively. One solution to this problem is the *GPU synchronization* strategy proposed in Chapter 4. GPU synchronization can efficiently synchronize the execution

across thread blocks. In this way, we avoid both the large number of kernel launches and the serialization in the tiled wavefront.

6.3.4.4 Trace Back: Via CPU or GPU?

In Sections 6.3.4.1 and 6.3.4.2, techniques for efficient access to DP matrices, the scoring matrix, and input sequences are presented. These techniques can accelerate the matrix filling phase. However, as described in Section 6.3.1, Smith-Waterman includes another phase—trace back. Trace back itself is serial and cannot be parallelized in a fine-grained parallel approach as the matrix filling. Thus, the execution time for trace back on the GPU is longer than on the CPU. For trace back, its execution needs two flag matrices (for affine-gap penalty) generated in the matrix filling phase. If the trace back is executed on the CPU, the two flag matrices should be copied back, then the total time for the trace back is the sum of time for transferring the matrices and executing the trace back on the CPU, which can be expressed as

$$T_{bc} = t_m + t_c \tag{6.5}$$

where T_{bc} is the total time to do trace back on the CPU, t_m is the data transfer time for the two flag matrices, and t_c is the time needed to execute the trace back on the CPU side. On the other hand, if we execute trace back on the GPU and then copy the alignments back to the host memory, the total time needed for the trace back is

$$T_{bg} = t_g + t_o \tag{6.6}$$

where T_{bg} is the total time to perform trace back on the GPU, t_g is the time to run the trace back on the GPU, and t_o is the time to copy the alignments from device memory to host memory.

Comparing the above two approaches, if the trace back is executed on the CPU, the execution time is less than that on the GPU, but the two flag matrices need to be copied from device memory to host memory. On the other hand, if trace back is executed on the GPU, the trace back itself may need more time to execute, but the data to be copied from device memory to host memory is much less, thus reducing data transfer time. Hence, it is possible to save some time in total if we run the trace back on the GPU, and this is verified in our experiments later.

6.3.5 Performance Evaluation

In this section, we evaluate the performance of our fine-grained parallel implementations. Specifically, the following three aspects are evaluated: (1) the performance improvement corresponding to each optimization technique in Section 6.3.4 for the fine-grained parallel implementations, where execution time and speedup with regard to the CPU sequential implementation are measured; (2) the kernel and total speedup achieved for different sequence sizes; (3) demonstrating the benefits brought by the GPU synchronization, in both the local and remote GPU scenarios.

In this chapter, the CUDA version 3.2 is used as the programming interface to various GPU cards (Tesla C2050 and Tesla C1060). Our host machine is configured with two Magny-Cours AMD CPUs (Each has eight cores.). In addition, there is 32GB host memory equipped on the node. The host machine is installed with the 64-bit CentOS 5.5 Linux distribution. The CUDA toolkit-3.2 is used for executing all the programs. For the results presented here, we choose sequence pairs of size 7K and report their alignment results. In this work, each result is the average of three runs. For the CPU sequential implementation,

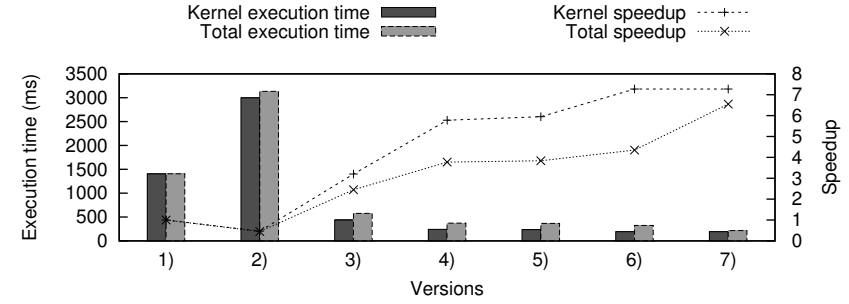
it is compiled with the -O3 optimization option.

6.3.5.1 Performance Improvement Corresponding to each Optimization Technique

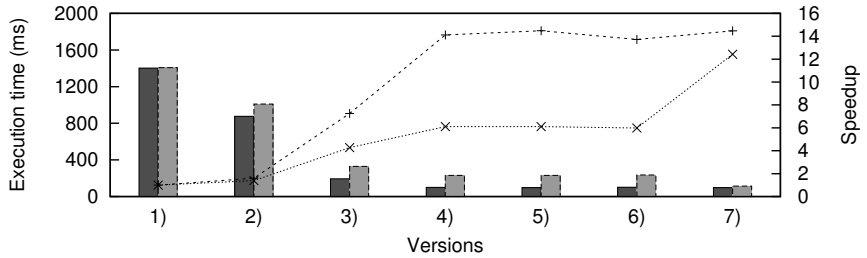
In this section, we present the performance improvement corresponding to each optimization technique. The optimization techniques include three aspects as explained in Section 6.3.4: 1) efficient DP matrix access, which includes four implementations—naïve implementation, tiled implementation, simple implementation, and coalesced implementation; 2) efficient accesses to input sequences and scoring matrix; and 3) trace back on the GPU to save data transfer time from device memory to host memory. In addition, performance of the CPU sequential implementation is presented in the figure as the baseline. Thus, we have seven versions in total, as shown in Figure 6.16, where Figure 6.16(a) shows the results on the Tesla C1060 (referred to as *Tesla* later) and Figure 6.16(b) is for the Tesla C2050, i.e. Fermi.

From Figure 6.16, we can observe four trends. First, the naïve implementation has comparable performance to the CPU sequential implementation. Specifically, for the sequential implementation, the matrix filling time is 1403.1 ms. On Tesla, the naïve implementation needs 2998.2ms, which is about 2 times slower. On Fermi, this time is 876.6 ms, about 2 times faster than the sequential implementation.

Second, with each optimization applied, some performance improvement is obtained. For instance, on Tesla, with the matrix re-alignment technique, the matrix filling time decreases to 242.7 ms, which is 6 times faster than the CPU sequential implementation. With the coalesced matrix access technique, the matrix filling time further decreases to 235.8 ms. Finally, with the input sequences and scoring matrix stored in the constant memory, the matrix filling time is reduced to 193.0 ms, more than 7 times faster than



(a) Tesla C1060 (Tesla)



(b) Tesla C2050 (Fermi)

Figure 6.16: Performance Improvement for Matrix Filling Corresponding to each Optimization Technique. (Note: The above versions are described as: (1) Serial implementation (2) Tiled implementation (3) Naïve implementation (4) Simple implementation (5) Coalesced implementation (6) Coalesced implementation + constant memory (7) Coalesced implementation + constant memory + trace back on the GPU.)

the CPU sequential implementation. On Fermi, the similar performance improvement is observed except the usage of the constant memory. With the constant memory used on Fermi, matrix filling time increases slightly from 96.9 ms to 102.3 ms. This is due to the L1/L2 caches on Fermi, which can accelerate the access to the input sequences and make the sequence access even faster than that in the constant memory.

Third, the tiled implementation is the second slowest across all GPU implementations. This is true on both the Tesla and the Fermi. With the tiled approach, the matrix filling time is 438.5 ms and 193.6 ms on Tesla and Fermi, respectively, which is about 2-fold

slower than the simple implementation, respectively.

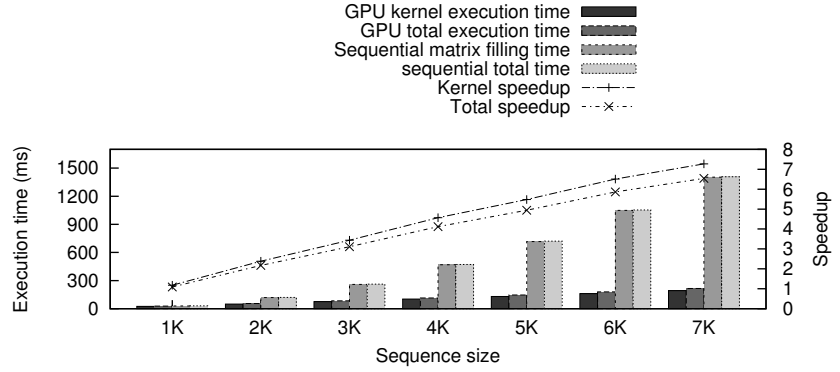
Fourth, as for the trace back, we show the data transfer time and the execution time of trace back in Table 6.5. From Table 6.5, the trace back itself is much faster on the host than on the device. It takes about 0.58 ms on the CPU, while this time is 13.90 ms and 10.65 ms on Tesla and Fermi, respectively. However, the overall time is different if data copy time is considered. With trace back executed on the CPU, two DP matrices need to be transferred from device memory to host memory. If the trace back is executed on the GPU, only two output sequences are copied back to the host memory. As such, with trace back executed on the CPU, data copy time is 121.23 ms and 126.16 ms on Tesla and Fermi, respectively. On the other hand, the data copy time is only 0.09 ms and 0.13 ms on Tesla and Fermi, respectively, with trace back executed on the device. Overall, executing trace back on the GPU takes less time than on the CPU and better overall performance can be achieved.

Table 6.5: Comparison of Total Time Needed by Trace Back (Unit: ms)

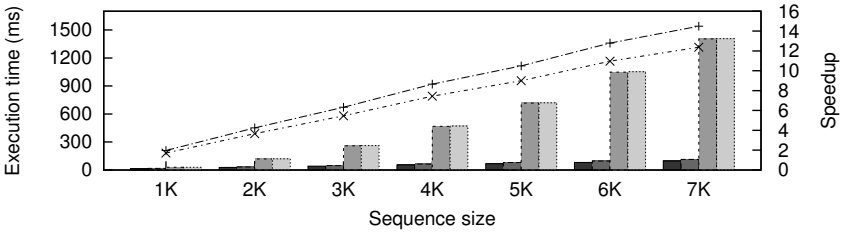
GPU type	Tesla C1060 (Tesla)		Tesla C2050 (Fermi)	
Trace back on	CPU	GPU	CPU	GPU
Data copy time	121.23	0.09	126.16	0.13
Trace back time	0.58	13.90	0.59	10.65
Total time	121.81	13.99	126.75	10.78

Using the implementations with the best performance, Figure 6.17 shows the kernel speedup and total speedup with regard to different sequence sizes. As above, Figure 6.17(a) is the result on Tesla and Figure 6.17(b) is for Fermi.

First, as can be seen, higher speedup can be achieved for larger sequences. For 1K-letter sequences, the speedup for the matrix filling is 1.18-fold on Tesla and 1.97-fold on Fermi; while this number is 7.27-fold and 14.49-fold on Tesla and Fermi, respectively,



(a) Tesla C1060 (Tesla)



(b) Tesla C2050 (Fermi)

Figure 6.17: Execution Time and Speedup with regard to Different Sequence Sizes for the Best Version. (On Tesla, it is the version with coalesced matrix access + constant memory with trace back on GPU. On Fermi, the version is the coalesced matrix access *without* constant memory and trace back is performed on GPU.)

for 7K-letter sequences. One of the reasons is that GPU resources are underutilized for small sequences. Second, overall speedup is less than the matrix filling speedup. The reason is that data need to be copied between host memory and device memory in the GPU implementations, but there is no such data copy in the CPU sequential implementation. Third, higher speedup is achieved on Fermi than on Tesla. This is due to the higher device memory bandwidth on Fermi. As described in Chapter 2, global memory bandwidth is 102.4GB/s on Tesla and it is 153.6GB/s on Fermi. Since matrix filling is memory bound, the higher the bandwidth is, the faster the DP matrix can be filled.

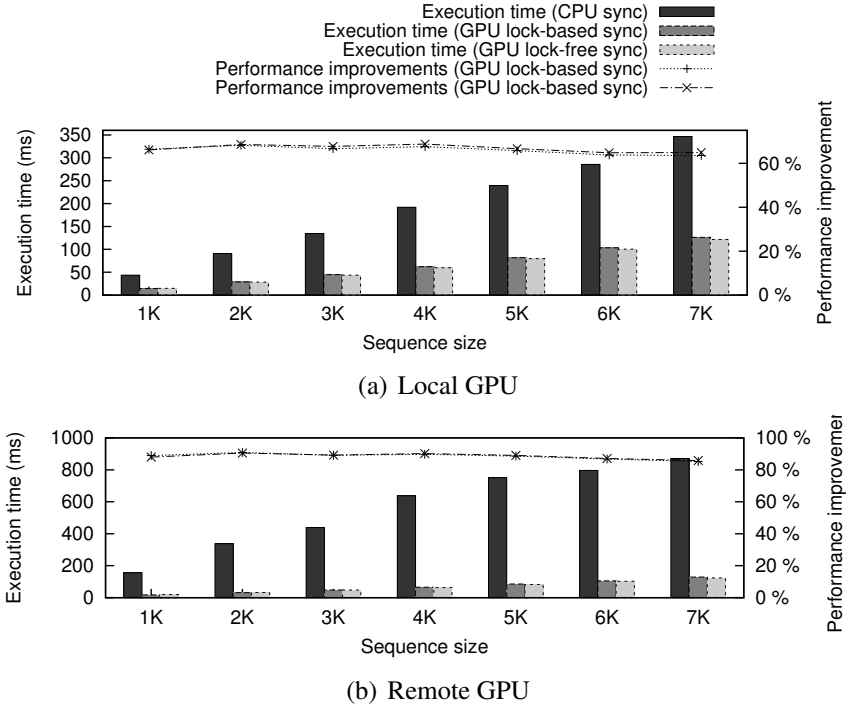
6.3.5.2 Performance Improvement Brought by GPU Synchronization in the Use of Remote GPUs

Data communication is needed across different blocks in our fine-grained parallel implementations. On the GPU, inter-block data communication occurs via the global memory and barrier synchronization is needed. Currently, barrier synchronization is achieved by multiple kernel launches, which can cause significant overhead that becomes even larger with the use of remote GPUs, as shown in Figure 3.16(c).

Using our OpenCL version of GPU synchronization strategies proposed in Section 4.8, we evaluate the benefits of the GPU synchronization. Our evaluation includes two aspects: 1) performance improvement brought by the GPU synchronization and 2) overhead caused by the use of remote GPUs based on the VOCL framework with the GPU synchronization. Experiments are performed on the Tesla C2050, i.e., Fermi GPU card.

Figure 6.18 shows the decrease in matrix filling time with our GPU synchronization applied. Figure 6.18(a) shows the local GPU case while Figure 6.18(b) shows the remote GPU case. As we can see, GPU synchronization significantly reduces the matrix filling time in both the local and the remote GPU cases. In the local GPU scenario, matrix filling time decreases by 66.4% when GPU lock-based synchronization is used. A similar decrease of the matrix filling time is observed with the GPU lock-free synchronization. For the remote GPU case, even more performance improvement is observed with the GPU synchronization. Specifically, the matrix filling time decreases by more than 85%, as shown in Figure 6.18(b).

Figure 6.19 shows the overhead involved in using remote GPUs with the GPU synchronization strategies. Figure 6.19(a) shows the overhead when using GPU lock-based

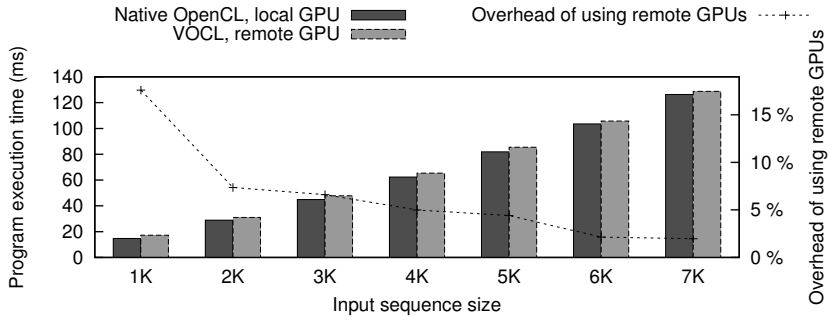


(a) Local GPU

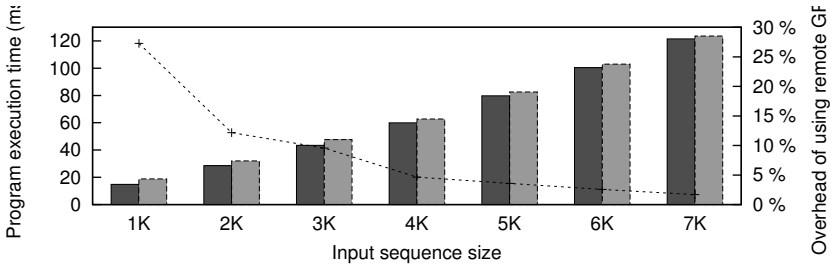
(b) Remote GPU

Figure 6.18: Performance Improvement Brought by GPU Synchronization

synchronization, and Figure 6.19(b) shows it when using GPU lock-free synchronization. Compared to the large overhead with the CPU synchronization when using the remote GPU, as shown in Figure 3.16(c), significantly less overhead is introduced when using GPU synchronization. Specifically, for the 1K-letter sequences, the relative overhead is 17.6% and 27.3% for the lock-based and lock-free synchronization, respectively. As for the 7K-letter sequences, the overhead is less than 2% for both GPU synchronization approaches. When using GPU synchronization, the overhead decreases with larger sequences because the larger number of messages transferred for multiple kernel launches with CPU synchronization is eliminated. In addition, because the kernel execution time of



(a) GPU Lock-Based Synchronization



(b) GPU Lock-Free Synchronization

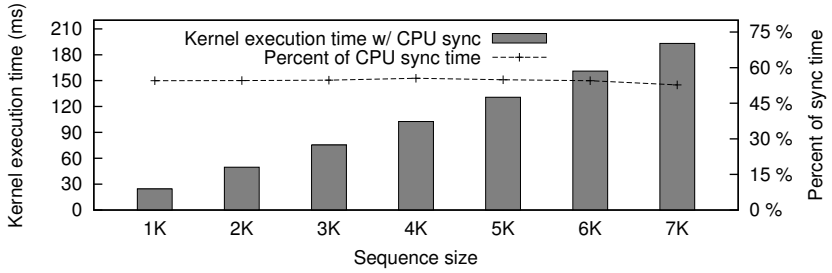
Figure 6.19: Overhead in the Utilization of Remote GPUs

Smith-Waterman occupies more than 80% of the total matrix filling time, the overhead in utilizing remote GPUs is quite small when using GPU synchronization.

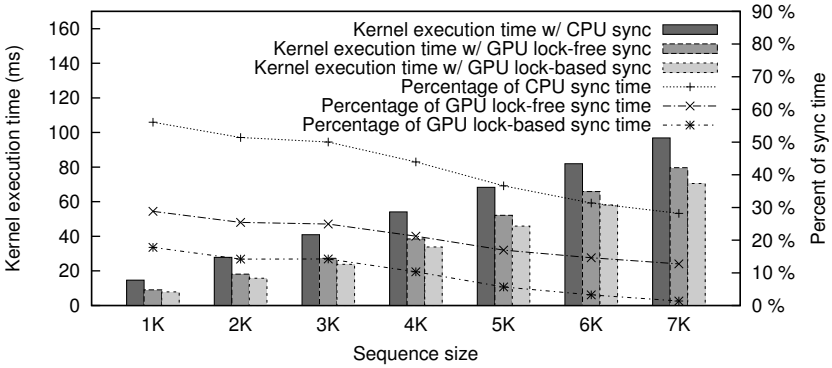
6.3.5.3 Time Spent in Inter-Block Data Communication

Figure 6.20 shows the kernel execution times and the percentage of time spent in inter-block data communication with different synchronization approaches for different sequence sizes. Note that the synchronization time is obtained using an indirect approach. Specifically, we measured the *computation time* by running an implementation using the GPU synchronization but without the barrier function called, and then we calculate the

synchronization time as the difference between the kernel execution time and the computation time. Figure 6.20(a) is for the Tesla,[4] and Figure 6.20(b) shows the results on the Fermi.



(a) Tesla C1060 (Tesla)



(b) Tesla C2050 (Fermi)

Figure 6.20: Percentage of Time Spent in Inter-Block Data Communication

From Figure 6.20, we have the following observations. First, when the CPU synchronization is used, time spent in inter-block data communication occupies a large percentage of the kernel execution time. Specifically, on Tesla, time for inter-block data communication is more than 50% for all the sequence sizes. This percentage is a little smaller on

[4]We did not show the results of the GPU synchronization on Tesla since guaranteeing the correctness of inter-block data communication using the memory fence function makes performance with the GPU synchronization worse than that with the CPU synchronization.

Fermi, but it is still about 30%. Second, the GPU synchronization significantly reduces the time for inter-block data communication. As we can see, the percentage of time spent in inter-block data communication reduces to about 15% and 5% with the GPU lock-based and lock-free synchronization, respectively, on Fermi. From the above results, we can see that the GPU synchronization can effectively improve the performance of inter-block data communication on Fermi.

6.3.6 Summary

In this chapter, we explained the parallelization of Smith-Waterman onto GPUs. We proposed a fine-grained parallel approach, which effectively covers the alignment of all the protein sequences in the NCBI GenBank. To improve its performance, we proposed several optimization techniques to improve the performance of memory access. With the parallel implementations, we showed the performance improvement brought by each optimization technique and compared its performance to that of the CPU sequential implementation. Finally, we used the Smith-Waterman algorithm as a case study to demonstrate the benefits of our proposed GPU synchronization strategies in the use of local GPUs and of remote GPUs via the VOCL framework. From our experimental results, using a traditional CPU synchronization approach in Smith-Waterman introduces significant overhead because a large number of small MPI messages need to be transferred to support multiple kernel launches.

Using our GPU synchronization eliminates the multiple kernel launches and, in turn, eliminates the large number of small MPI messages. As a result, the overhead that is introduced is quite small and decreases further for the alignment of larger sequences.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This dissertation proposes the VOCL (Virtual OpenCL) framework to generalize the utility of GPUs in large-scale heterogeneous computing systems. This framework, based on the OpenCL programming model, exposes physical GPUs as decoupled virtual resources that can be transparently managed, independent of the application execution. The proposed framework requires no source code modification for applications to take advantage of remote GPUs. In other words, all GPUs installed in a system can be used in the same way no matter whether they are local or remote. For local GPUs, VOCL internally calls the native OpenCL functions for GPU computing. As for remote GPUs, VOCL translates OpenCL function calls to MPI data communication between the local node and the remote node, thus introducing additional overhead as compared to local GPU usage. To reduce the overhead, we propose optimization techniques, including kernel argument caching and data transfer pipelining. With the optimizations, we achieve about 85% of

the data write bandwidth and 90% of the data read bandwidth when compared to writing and reading in a native nonvirtualized environment, respectively. We evaluate the performance of VOCL using four real-world applications (matrix multiplication, matrix transpose, Smith-Waterman, and n-body) with various computation and memory access intensities and demonstrate that compute-intensive applications can execute with negligible overhead in the VOCL environment even when remote GPUs are used.

The VOCL framework enables applications to transparently utilize remote GPUs. However, if an application program needs data communication across different work-groups during its execution and launches multiple kernels for the data communication, large synchronization overhead is introduced, particularly in the scenario of remote GPU utilization. This is due to the large number of small messages transferred for the multiple kernel launches. To reduce the overhead caused by multiple kernel launches, we propose two GPU synchronization strategies, which can synchronize the program execution by using our proposed barrier function `__gpu_sync()` on the GPU instead of terminating the kernel's execution and re-launching the kernel from the CPU. As a result, the large number of small messages are removed, and overhead can be reduced. With GPU synchronization, the overhead involved in the use of a remote GPU decreases from about 150% to about 15% for the Smith-Waterman algorithm. Note that the GPU synchronization strategy can reduce the overhead in using a remote GPU, but it is an application-level optimization and application programs need to be optimized accordingly.

Finally, we extend the VOCL framework to support the live migration of virtual GPUs across physical GPUs. With the migration capabilities supported, the load across physical GPUs can be balanced, and system maintenance can be performed without waiting for

completion of all computation on a node.

7.2 Future Work

The current VOCL framework generalizes the GPU usage. However, VOCL just provides the basic functionalities for virtualization—viewing all GPUs in a system in the same way and supporting live migration of virtual GPUs across remote GPUs. Obviously, it can be extended and enhanced in the following aspects:

1. Load metrics on physical GPUs. Currently, we use the number of function calls that are issued but not finished as the metric. This does not consider the properties of different kernels and the problem size. Future work would entail alternative metrics to measure the load on each physical GPU.

2. Further generalization of the migration of virtual GPUs. The current VOCL framework only supports migration of virtual GPUs across remote physical GPUs. Next step would be to extend the VOCL framework to support migration across all the GPUs (both the local and remote) in a system.

3. Redundant computation. Previous generations of GPUs provide no protection against software errors that may be caused by cosmic radiation [71]. Thus, it is possible that computing results are incorrect in practice. Though the latest Fermi architecture provides error-correcting code (ECC) memory, it is sometimes not used in order to achieve better performance. As such, one extension to the VOCL framework is to perform redundant computation. With redundant computation, each VOCL VGPU can be mapped to multiple OpenCL VGPUs. When an OpenCL function is called,

VOCL sends the function to multiple physical GPUs simultaneously and waits for results from the multiple GPUs. Then it chooses the result with values from the majority of the physical GPUs.

4. Performance modeling and task scheduling. With the VOCL framework, multiple programs can be executed in a flexible way. One problem is the efficient utilization of computational resources. To this end, it is necessary that a performance model is used to model the overall efficiency of the computational resources. Based on the performance model, we can design various task scheduling strategies to achieve the overall best performance.

5. Super-GPU in a system. Our VOCL framework can support the utility of all GPUs in a system, but programmers still need to specify which GPU a program is assigned to. For programmers without a good knowledge of the overall system configuration and resource usage, it is hard for them to choose appropriate GPUs for their program execution. As such, one way to extend the VOCL framework is to combine all GPUs together to form one *super-GPU* in the whole system, and then programmers do not need to choose the GPU for their program execution, which can further reduce the programming effort to use GPUs to accelerate their programs.

Bibliography

[1] MPICH2: High-performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/index.php.

[2] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[3] OpenCL and the ATI Stream SDK v2.0. http://developer.amd.com/documentation/articles/pages/opencl-and-the-ati-stream-v2.0-beta.aspx.

[4] Top500 Supercomputing Sites. http://www.top500.org/.

[5] A. M. Aji, W. Feng, F. Blagojevic, and D. S. Nikolopoulos. Cell-SWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine. In *Proc. of the ACM International Conference on Computing Frontiers*, May 2008.

[6] J. Alemany and E. W. Felten. Performance Issues in Non-Blocking Synchronization on Shared-Memory Multiprocessors. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J Mol Biol*, 215(3):403–410, October 1990.

[8] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[9] AMD/ATI. Stream Computing User Guide. April 2009. http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf.

[10] W. D. Andrew, A. Chang, A. Chien, S. Fiske, W. Horwat, J. Keen, R. Lethin, M. Noakes, P. Nuth, E. Spertus, D. Wallach, and D. S. Wills. Retrospective: the J-Machine. In *Proc. of the 25th International Symposium on Computer Architecture*, 1998.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. id Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. In *Communications of the ACM*, April 2010.

[12] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Proc. of the 15th EuroPVM/MPI*, September 2008.

[13] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications (IJHPCA)*, 24(1):49–57, 2009.

[14] A. Barak and A. Shiloh. The MOSIX Cluster Operating System for High-Performance Computing on Linux Clusters, Multi-Clusters, GPU Clusters and Clouds, 2011. A white paper.

[15] A. Barak and A. Shiloh. The MOSIX Virtual OpenCL (VCL) Cluster Platform. In *Proc. of the Intel European Research and Innovation Conference*, October 2011.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Wareld. Xen and the Art of Virtualization. In *Proc. of Proceedings of the ACM Symposium on Operating Systems Principles*, December 2003.

[17] G. Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[18] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. TurboBLAST : A Parallel Implementation of Blast Built on the Turbohub. In *Proc. of the Parallel and Distributed Processing Symposium*, April 2002.

[19] M. Cameron, H. E. Williams, and A. Cannane. Improved Gapped Alignment in BLAST. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, volume 1, pages 116–129, July 2004.

[20] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proc. of the 23rd ACM SIGGRAPHEUROGRAPHICS Symp. on Graphics Hardware*, pages 57–64, June 2008.

[21] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its First Implementation. *IBM developerWorks*, Nov 2005.

[22] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a Message-Driven Processor. In *Proc. of the 14th Annual International Symposium on Computer Architecture*, 1987.

[23] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proc. of the ACM/IEEE conference on Supercomputing*, November 2003.

[24] A. Danaliszy, G. Mariny, C. McCurdyy, J. S. Meredithy, P. C. Rothy, K. Spaffordy, V. Tipparajuy, and J. S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite, March 2010. `http://ft.ornl.gov/doku/shoc/start`.

[25] A. E. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *Proc. the 4th International Conference on Linux Clusters*, June 2003.

[26] G. Dozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Proc. of the 15th EuroMPI*, September 2010.

[27] J. Duato, F. D. Igual, R. Mayo, A. J. Pena, E. S. Quintana-Orti, and F. Silla. An Efficient Implementation of GPU Virtualization in High Performance Clusters. In *Lecture Notes in Computer Science*, volume 6043, pages 385–394, 2010.

[28] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. Performance of CUDA Virtualized Remote GPUs in High Performance Clusters. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, September 2011.

[29] W. Feng and S. Xiao. To GPU Synchronize or Not GPU Synchronize? In *Proc. of the IEEE International Symposium on Circuits and Systems*, May 2010.

[30] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. de Supinski, and R. Thakur. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *Proc. of the IEEE International Conference on Cluster Computing*, September 2010.

[31] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006.

[32] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proc. of Supercomputing'2008*, pages 1–12, October 2008.

[33] A. Greβ and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *IPDPS*, April 2006.

[34] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011.

[35] R. Gupta and C. R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

[36] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *Proc. of IPDPS*, May 2011.

[37] P. H. Hargrove and J. C. Duell1. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Proc. of SciDAC*, June 2006.

[38] E. Herruzo, G. Ruiz, J. I. Benavides, and O. Plata. A New Parallel Sorting Algorithm based on Odd-Even Mergesort. In *Proc. of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 18–22, 2007.

[39] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. In *ACM Transactions on Reconfigurable Technology and Systems*, volume 1, June 2008.

[40] I. Jung, J. Hyun, J. Lee, and J. Ma. Two-Phase Barrier: A Synchronization Primitive for Improving the Processor Utilization. *International Journal of Parallel Programming*, 29(6):607–627, 2001.

[41] G. J. Katz and J. T. Kider. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *Proc. of the 23rd ACM SIGGRAPHEUROGRAPHICS Symp. on Graphics Hardware*, pages 47–55, June 2008.

[42] Khronos OpenCL Working Group. The OpenCL Specification (Version 1.1), June 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[43] V. Lakshmi. High Performance Computing for Hydrological Sciences. `http://www.cuahsi.org/his/docs/hpc-writeup.pdf`.

[44] D. Lavenier. G.: Seed-based Genomic Sequence Comparison Using a FPGA/FLASH Accelerator. In *Proc. of the IEEE Conference on Field Programmable Technology*, 2006.

[45] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. Feng. Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008.

[46] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[47] H. Lin, X. Ma, W. Feng, and N. F. Samatova. Coordinating computation and i/o in massively parallel sequence search. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2010.

[48] W. Liu. CUDA-BLASTP on Tesla GPUs. January 2010. `http://www.nvidia.com/object/blastp_on_tesla.html`.

[49] W. Liu, B. Schmidt, and W. Mueller-Wittig. CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2011.

[50] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. *IPDPS*, April 2006.

[51] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU Accelerated Smith-Waterman. In *Proc. of the 2006 International Conference on Computational Science, Lectures Notes in Computer Science Vol. 3994*, pages 188–195, June 2006.

[52] B. D. Lubachevsky. Synchronization Barrier and Related Tools for Shared Memory Parallel Programming. *International Journal of Parallel Programming*, 19(3):225–250, 1990. 10.1007/BF01407956.

[53] A. Mahram and M. C. Herbordt. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering. In *Proc. of the 24th ACM International Conference on Supercomputing*, June 2010.

[54] S. A. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 2008.

[55] G. Martinez, M. Gardner, and W. Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures. In *Proc. of 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, November 2011.

[56] Message Passing Interface Forum. The Message Passing Interface (MPI) Standard. `http://www.mcs.anl.gov/research/projects/mpi/`.

[57] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *Proc. of the 8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, October 2008.

[58] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: Towards a Portable, Cost-effective Open Source Hardware Implementation. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[59] NCSA. Blue Waters — Sustained Petascale Computing, 2012. `http://www.ncsa.illinois.edu/BlueWaters/`.

[60] Network-Based Computing Laboratory. MVAPICH2 (MPI-2 over OpenFabrics-IB, OpenFabrics-iWARP, PSM, uDAPL and TCP/IP). `http://mvapich.cse.ohio-state.edu/overview/mvapich2`.

[61] NVIDIA. CUDA Compute Visual Profiler, June 2010. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/VisualProfiler/computeprof.html`.

[62] NVIDIA. CUDA Occupancy Calculator, June 2010. `http://developer.download.nvidia.com/compute/cuda/3_1/sdk/docs/CUDA_Occupancy_calculator.xls`.

[63] NVIDIA. NVIDIA CUDA Programming Guide-3.1, June 2010. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf`.

[64] NVIDIA. High Performance Computing - Supercomputing with Tesla GPUs. 2011. `http://www.nvidia.com/object/tesla_computing_solutions.html`.

[65] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems*, 3:677–695, 2007.

[66] C. Oehmen and J. Nieplocha. ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Trans. Parallel Distrib. Syst.*, 17(8), 2006.

[67] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. K. Panda. RDMA-Based Job Migration Framework for MPI over InfiniBand. In *Proc. of IEEE International Conference on Cluster Computing*, September 2010.

[68] D. W. Roeh, V. V. Kindratenko, and R. J. Brunner. Accelerating Cosmological Data Analysis with Graphics Processors. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[69] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO Atomic Mode Without File System Support. In *IEEE International Symposium on Cluster Computing and the Grid*, May 2005.

[70] E. F. O. Sandes and A. C. M. de Melo. CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences. In *Proc. of PPoPP*, January 2010.

[71] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On Testing GPU Memory for Hard and Soft Errors. In *Proc. of The first annual Symposium on Application Accelerators in High-Performance Computing (SAAHPC'09)*, 2009.

[72] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 99, 2011.

[73] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. In *Journal of Molecular Biology*, April 1981.

[74] E. Sotiriades and A. Dollas. A General Reconfigurable Architecture for the BLAST Algorithm. In *Journal of VLSI Signal Processing*, 2007.

[75] G. M. Striemer and A. Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *IPDPS*, May 2009.

[76] J. A. Stuart and J. D. Owens. Message Passing on Data-Parallel Architectures. In *IPDPS*, May 2009.

[77] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. In *Dr. Dobb's Journal*, February 2005.

[78] Synergy Lab @ Virginia Tech. `http://synergy.cs.vt.edu`.

[79] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies*, December 2009.

[80] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast Implementation of DGEMM on Fermi GPU. In *Proc. of Supercomputing*, November 2011.

[81] N. Vasudevan and P. Venkatesh. Design and Implementation of a Process Migration System for the Linux Environment. In *Atlas conference abstracts*, March 2006.

[82] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of Supercomputing*, November 2008.

[83] V. Volkov and B. Kazian. Fitting FFT onto the G80 Architecture. pages 25–29, April 2006. `http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6-report.pdf`.

[84] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. In *Bioinformatics*, pages 182–188, 2011.

[85] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Process-Level Live Migration in HPC Environments. In *Proc. of ACM/IEEE conference on Supercomputing*, June 2006.

[86] F. Xia, Y. Dou, and J. Xu. Families of FPGA-Based Accelerators for BLAST Algorithm with Multi-seeds Detection and Parallel Extension. In *Communications in Computer and Information Science*, volume 13, 2008.

[87] S. Xiao, A. Aji, and W. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proc. of the 15th ICPADS*, December 2009.

[88] S. Xiao and W. Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *Proc. of the 24th IPDPS*, April 2010.

[89] S. Xiao, H. Lin, and W. Feng. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *Proc. of the 25th IPDPS*, May 2011.

Vita

Shucai Xiao received his B.S. degree in Electronics and Information Technologies in 2001 from Beijing University of Posts and Telecommunications, and he received his M.S. degree in Electronics Engineering in 2004 from Tsinghua University, Beijing, China. He is currently a Ph.D. student in the Bradley Department of Electrical and Computer Engineering at Virginia Tech, under the supervision of Prof. Wu-chun Feng. Shucai's research interests include generalizing the utility of GPUs as accelerators for general purpose computation in large-scale heterogeneous computing systems, which includes facilitating GPU utilization as well as achieving good performance on them. Shucai's publication list can be found here.

Conference Proceedings

1. Palden Lama, Yan Li, Ashwin M. Aji, Pavan Balaji, James Dinan, Shucai Xiao, Yunquan Zhang, Wu-chun Feng, Rajeev Thakur, and Xiaobo Zhou, pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments, In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, July, 2013

2. Shucai Xiao and Wu-chun Feng, Generalizing the Utility of GPUs in Large-Scale

Heterogeneous Computing Systems, In *the Ph.D. Forum at the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS Ph.D. Forum)*, May, 2012

3. Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng, Transparent Accelerator Migration in a Virtualized GPU Environment, In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May, 2012

4. Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng, VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units, In *Proceedings of the 1st Innovative Parallel Computing (InPar)*, May, 2012

5. Shucai Xiao, Heshan Lin, and Wu-chun Feng, Acceleration of Protein Sequence Search in a Heterogeneous Computing System, In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May, 2011

6. Wu-chun Feng and Shucai Xiao, To GPU Synchronize or Not GPU Synchronize?, In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May, 2010

7. Shucai Xiao and Wu-chun Feng, Inter-Block GPU Communication via Fast Barrier Synchronization, In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April, 2010

8. Shucai Xiao, Ashwin M. Aji, and Wu-chun Feng, On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit, In *Proceedings of the 15th International Conference of Parallel and Distributed Systems (ICPADS)*, December, 2009

9. Shucai Xiao, Jung-Min Park, and Yanzhu Ye, Tamper Resistance for Software Defined Radio Software, In *proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, July, 2009

10. Song Huang, Shucai Xiao, and Wu-chun Feng, On the Energy Efficiency of Graphics Processing Units for Scientific Computing, In *Proceedings of the 5th IEEE Workshop on High-Performance, Power-Aware Computing (HPPAC)*, May, 2009

Journals

1. Shucai Xiao, Zhijian Ou, and Zuoying Wang, A Speaker Clustering Algorithm in Speech Recognition, *Journal of Chinese Information Processing* (In Chinese), P84–P87, July, 2005

2. Shucai Xiao and Zuoying Wang, A New log Energy Feature in Endpoint Detection, *Audio Engineering* (In Chinese), P37–P40, June, 2004