Spring 2012

Scaling CUDA for Distributed Heterogeneous Processors

Siu Kwan Lam
San Jose State University

SCALING CUDA FOR DISTRIBUTED HETEROGENEOUS PROCESSORS

A Thesis

Presented to

The Faculty of Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Siu Kwan Lam

May 2012

The Designated Thesis Committee Approves the Thesis Titled

SCALING CUDA FOR DISTRIBUTED HETEROGENEOUS PROCESSORS

by

Siu Kwan Lam

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2012

ABSTRACT

SCALING CUDA FOR DISTRIBUTED HETEROGENEOUS PROCESSORS

by Siu Kwan Lam

The mainstream acceptance of heterogeneous computing and cloud computing is prompting a future of distributed heterogeneous systems. With current software development tools, programming such complex systems is difficult and requires an extensive knowledge of network and processor architectures. Providing an abstraction of the underlying network, message-passing interface (MPI) has been the standard tool for developing distributed applications in the high performance community. The problem of MPI lies with its message-passing model, which is less expressive than the shared-memory model. Development of heterogeneous programming tools, such as OpenCL, has only begun recently. This thesis presents *Phalanx*, a framework that extends the virtual architecture of CUDA for distributed heterogeneous systems. Using MPI, Phalanx transparently handles intercommunication among distributed nodes. By using the shared-memory model, Phalanx simplifies the development of distributed applications without sacrificing the advantages of MPI. In one of the case studies, Phalanx achieves $28\times$ speedup compared with serial execution on a Core-i7 processor.

ACKNOWLEDGEMENTS

I would like to offer my gratitude to my advisor Dr. Donald Hung, who has provided enormous support to my thesis. I would also like to thank Dr. Xiao Su, who has encouraged my interest in research. In addition, this thesis is supported in part by the SJSU Award for Research, Scholarship, or Creative Activity (SJSU RSCA grant).

Contents

List of Figures

Chapter 1

Introduction

This thesis presents a new approach to programming a distributed heterogeneous system. Traditionally, high performance distributed applications use message-passing interface (MPI) for intercommunication among nodes. The difficulty to program MPI applications lies with its use of the message-passing model, which is less expressive than the shared-memory model. The new approach does not aim to replace MPI, but to hide all MPI specifics from the programmers. Therefore, programmers can program efficiently using the shared-memory model with the advantages of MPI.

This thesis focuses on the development of the *Phalanx* framework, which compiles compute unified device architecture (CUDA) kernels for parallel execution on multicore processors and distributed heterogeneous systems. Phalanx extends the virtual architecture of CUDA for MPI-based distributed applications. Together with the on-demand and dynamic allocation of compute resources provided by cloud computing, Phalanx allows CUDA kernels to scale from a manycore GPU to a massive distributed system consisting of hundreds of processors.

This chapter provides a general background for the discussion of Phalanx. In the rest of this chapter, Sections 1.1 and 1.2 provide a brief discussion of different levels of parallelism and parallel architectures, respectively. Section 1.3 discusses the challenges faced by distributed application programmers. Section 1.4 presents several related projects that are leading the research in heterogeneous computing. Finally, Section 1.5 states the goals of this project and explains the organization of this thesis.

1.1 Levels of Parallelism

Traditionally, software developers have relied on the advances in hardware for higher performance. For years, the processor industry merely pushed for a higher clock rate to increase computational performance until the *power wall* was reached. The power wall represents a physical limitation of the CMOS technology. Power consumption increases exponentially with an increment in clock rate [1]. Since power is dissipated as heat, the temperature of a processor would rise to an unbearable level. To further improve the performance without increasing clock rate, processor designers began to exploit various forms of parallelism in computer programs.

The following subsections describe three levels of parallelism. The order of presentation indicates a growing reliance on programmer effort. The first level of parallelism can be automatically discovered by compilers and processors. The last level of parallelism requires explicit control by the programmer.

1.1.1 Instruction-Level Parallelism

With *instruction-level parallelism* (ILP), a processor executes data-independent instructions in a concurrent fashion. Patterson and Hennessy [2] described the following techniques. *Pipelining* allows data-independent instructions to overlap in different stages of a processor datapath. S*uperscalar* implements multiple datapaths for data-independent instructions to execute in parallel. A more complex implementation uses *out-of-order* execution, which reorders the sequence of instructions to reduce data dependency among consecutive instructions.

ILP is limited and further exploitation requires overly complicated logic in the processor. With an exponential growth of transistor count in processors predicted by Moore's Law, a more efficient use of transistors is needed [2].

1.1.2 Data-Level Parallelism

With *data-level parallelism* (DLP), a processor exploits the scenario where the same instruction operates on a wide data set [3]. Such a scenario occurs frequently in loops where the same operation repeats for every element in an array. Processors usually implement vector instructions for DLP. Some compilers can automatically generate vector instructions from high-level source code, but explicit vectorization by a programmer will typically yield higher performance.

1.1.3 Task-Level Parallelism

Task-level parallelism (TLP) is a coarsened version of ILP. Independent computing tasks can be executed in parallel. A task is usually represented by a thread. Each thread has an independent instruction stream. A multicore processor executes multiple threads in different processor cores. Multithreaded processors, such as Oracle UltraSparc and Intel Hyperthread processors, overlap instructions from multiple threads within a single processor by sharing processor resources [2]. When a thread engages in a long memory operation using the load-store units, other threads can execute in parallel using the arithmetic units.

1.2 Parallel Architectures

Consisting of a combination of ILP, DLP and TLP, a modern parallel architecture is both massive and complex. As noted by Gebali [3], it is difficult to precisely categorize parallel architectures using Flynn's Taxonomy, which divides computer systems according to the organization of instruction and data streams. Some parallel architectures exist on a single silicon die. Some exist as a group of networked machines. The following categorization is based on that of Gebali [3].

1.2.1 SIMD Processor

single-instruction multiple-data (SIMD) processors are common in commodity computers because adding support for SIMD instructions to scalar processors is relatively easy [4, 3]. Almost all modern Intel processors contain the MMX and SSE multimedia extensions, supporting up to 128-bits of vector data. Intel recently released the SandyBridge architecture with *advanced vector execution* (AVX), extending the vector width to 256-bits.

Pattern and Hennessy [4] explained how SIMD instructions can reduce the time to fetch, decode, and execute. A SIMD instruction performs multiple parallel operations in one cycle, replacing 2-8 scalar instructions. With 256-bit vectors, a single SIMD instruction can execute 8 parallel single precision floating-point operations.

1.2.2 Multicore

Multicore processors execute multiple threads in different execution cores. Explicit task division is required to use the capability of multicore processors. Determining the granularity of tasks is not easy. If a program has a fine-grain TLP, the overhead due to communication and synchronization could affect the performance of the program significantly. Some on-going projects aim to perform automatic scheduling of tasks. For instance, *StreamIT* [5] is a domain-specific language that allows programmers to describe the data-flow of a program. Its compiler converts data-flow descriptions into parallel tasks. However, the difference in the programming model creates a steep learning curve for programmers.

1.2.3 Manycore

Manycore architectures, such as NVIDIA CUDA, combine DLP and TLP in a massive manner. Using thousands of parallel threads, a NVIDIA CUDA-enabled *graphical processing unit* (GPU) acts as an accelerator for the CPU. In the past, the design of GPUs exploited the high degree of DLP in computer vision applications. With the introduction of CUDA, GPUs can now support general-purpose applications. More applications can benefit from the high computation and memory throughput of these general-purpose GPUs (GPGPUs).

GPUs execute specially compiled programs called *kernels*. To invoke a CUDA kernel, the CPU transfers the kernel and all depending data to the GPU. During the kernel execution, the CPU is free to compute other tasks. When the kernel computation has been completed, the CPU retrieves the results from the memory on the GPU.

1.2.4 Distributed Systems

Gebali [3] described two kinds of distributed systems. A *cluster* system consists of interconnected computers over a *local-area network* (LAN). These computers are often identical. A compute *grid* consists of interconnected computers over a *wide-area network* (WAN). Computers in a grid are often heterogeneous, consisting of different processor architectures, operating systems, and data models. Distributed systems are suitable for coarse grain parallelism because the communication overhead can be significant.

Communication among nodes in a distributed system usually uses message-passing interface (MPI). MPI [6] is a standardized *application programming interface* (API) that provides a high performance messaging facility for high level languages. It is scalable. It supports parallel computing in shared-memory multiprocessors, clusters and massive compute grids. It is also portable. OpenMPI [7, 8] is a notable open-source effort that combines technologies from multiple MPI implementations and strongly support for heterogeneous processors, operating systems (OSs), and networks.

Cloud computing is a form of grid computing [3]. *Amazon Elastic Compute Cloud* (EC2)[1] supports both CPU and GPU instances. Users can instantly deploy a heterogeneous compute grid with no installation cost. Cloud services supply compute

[1]http://aws.amazon.com/ec2/

resources as utilities, so that users pay only for their usage. Cloud computing provides a new possibility for affordable supercomputing.

1.3 Challenges in Developing Distributed Applications

Computing resources are available for building distributed heterogeneous systems at low costs, but challenges remain in the software development for such systems. A major problem lies in the programming model. Programmers are familiar with the shared-memory model, which is used by most of the popular programming languages, including C/C++, JAVA, and FORTRAN. MPI uses the message-passing model, which expresses data movements in messaging events, such as send and receive. The following presents a comparison between the shared-memory model and the message-passing model.

1.3.1 Shared-Memory Model Versus Message-Passing Model

The shared-memory model and the message-passing model each have advantages for different parallel programming patterns. For complex and dynamic data movements, the shared-memory model is more expressive, whereas the message-passing model requires custom message composition to describe each data movement. For synchronization, the shared-memory model requires explicit control through the use of barriers and memory fences, whereas synchronization in the message-passing model is implicitly defined by simple send and receive events.

Listing 1.1: A C-like pseudocode demonstrating a parallel counter increment using the shared-memory model.

```
int sharedCounter=0; //shared by all threads
int localVar;        //local to thread
//obtain mutex lock on sharedCounter
lock(sharedCounter);
localVar = sharedCounter; //read sharedCounter
localVar += 1;            //increment local data
sharedCounter = localVar;  //modify shared data
//release mutex lock on sharedCounter
```

```
unlock(sharedCounter);
//ensure all threads see the new value
memoryfence();
```

Listing 1.2: A C-like pseudocode demonstrating a parallel counter increment using the message-passing model.

```
//assume a 10 nodes ring
//the host node is node-0
//the last node is node-9
int counter;
if(nodeID==0){//the host node
    counter=1;
    send(counter, 1); //sends to node-1
    recv(counter, 9); //receive from node-9
}else{//other worker nodes
    recv(counter, (nodeID-1)%10); //receive from previous node
    count+=1;
    send(count, (nodeID+1)%10); //send to the next node
}
```

Listing 1.1 shows a pseudocode for performing counter increments in parallel fashion using the shared-memory model. Explicit use of locks and memory fences is necessary to ensure data consistency. Listing 1.2 shows a similar pseudocode using the memory-passing model. Unlike the shared-memory model, each node has a separate address-space. The nodes form a ring topology. Each node adds to the *counter* variable and passes the variable to the next node.

Kumar *et al.* [9] claimed that the divide-and-conqueur pattern is easier to map onto the shared-memory model. The message-passing model offers a better task isolation and easier validation. Despite the shared-memory model being more expressive, the error-prone nature of explicit synchronization and unclear task isolation can cause difficulty in software verification and maintenance. The message-passing model is more suitable for the following reasons. First, tasks are naturally isolated in different processes. Second, race conditions are impossible with the separation of address-space.

However, the message-passing model adds additional complexity in performance-critical situations. How long a synchronous receive event must wait for its corresponding send event is unclear. The processor remains idle when waiting for message events. In MPI, asynchronous messaging can reduce idle time by overlapping computations, but it requires explicit synchronization and management. Asynchronous messaging increases the overall complexity of a program.

While caching in the shared-memory model is often automatic, the message-passing model requires programmers to adjust program design and message composition to account for data locality. Due to the overhead of each message, coalescing data transfer can reduce the number of messages, thereby improving network utilization. The network bandwidth is often lower than the computing throughput. Without efficient use of network resources, a message-passing application can easily become I/O bounded.

1.4 Related Works

This section briefly introduces three related projects that are leading the research in heterogeneous computing.

1.4.1 MCUDA

Stratton et al. [10] described a source-to-source compilation framework called MCUDA for translating CUDA-C source code into multithreaded ANSI-C programs. MCUDA decomposes CUDA kernels at synchronization points and encloses the resulting subkernels with a loop that iterates over all threads. Each subkernel loop can be compiled into a SIMD loop for efficient execution on CPUs.

In contrast, Phalanx compiles at the PTX-level. Any high level language that can be lowered to PTX can be used in Phalanx.

1.4.2 Ocelot

Ocelot[2] is a large research project from the School of Electrical and Computer Engineering, Georgia Institute of Technology. Diamos *et al.* [11] described Ocelot as a dynamic compilation framework for CUDA and an opensource CUDA runtime. It can just-in-time recompile CUDA kernels for execution on NVIDIA GPUs, AMD GPUs and x86 CPUs. Ocelot uses a series of complex analysis to transform PTX for CPU execution. Ocelot facilitates the research of GPU computing by providing debugging, analysis and monitoring frameworks for CUDA kernels. A recently added feature allows remote machines to emulate GPUs. Comparing with Ocelot, Phalanx distributes the computation of a kernel across multiple machines. Ocelot offloads a kernel execution to a remote machine.

1.4.3 OpenCL

OpenCL (open computing language)[3] is a parallel programming language that focuses in portability across heterogeneous devices. With OpenCL, programmers can write portable parallel programs. OpenCL programs execute on handheld devices, personal computers, and servers. Khronos Group maintains an open standard for OpenCL. Hardware support relies on individual vendors to provide implementations for specific devices.

Phalanx can execute program written in OpenCL by lowering OpenCL to PTX. NVIDIA officially supports the compilation of OpenCL to PTX[4].

1.5 Project Goals and Thesis Organization

Heterogeneous computing is an on-going research. Phalanx introduces a new approach by combining CUDA and MPI. Phalanx aims to simplify the application development for distributed heterogeneous systems, so that individuals and businesses can easily leverage

[2]http://gpuocelot.gatech.edu
[3]http://www.khronos.org/opencl/
[4]http://developer.nvidia.com/opencl

the highly available and affordable compute resources provided by cloud services. Phalanx compiles CUDA kernels for executing on distributed systems. Heterogeneous support is currently limited to different CPU architectures. GPU support is possible, but it is left for future works. Phalanx achieves its goal by implementing:

- a shared-memory model for programming distributed systems;
- a unified virtual architecture that scales across heterogeneous systems; and,
- a runtime system that implements the unified virtual architecture on distributed systems.

The rest of this thesis discusses the Phalanx framework in details. Providing an overall description of Phalanx, Chapter 2 introduces its functions, major components and the development tasks. Chapters 3, 4 and 5 explain the details of the PTX-to-LLVM compiler, the runtime system and the memory system, respectively. Chapters 6 and 7 present two application case studies. Finally, Chapter 8 concludes the thesis.

Chapter 2

Phalanx Overview

The Phalanx framework uses CUDA as a unified virtual architecture. It consists of the following components:

1. a compiler for translating CUDA kernel definitions for CPU backends;

2. a runtime system that emulates CUDA for distributed systems; and,

3. a memory system that handles intercommunication among nodes in the distributed systems.

These components are discussed in details in the following chapters. Chapter 3 discusses the PTX-to-LLVM compiler. Chapter 4 discusses the runtime system. The memory system, which is part of the runtime system, is discussed separately in Chapter 5.

2.1 CUDA as a Unified Virtual Architecture

Phalanx extends CUDA for distributed heterogeneous systems. CUDA is a proprietary virtual architecture and programming model for NVIDIA GPGPUs (general-purpose graphical processing units). CUDA combines the expressiveness of the shared-memory model and the clear isolation of the message-passing model through its thread and memory hierarchies.

2.1.1 Thread Hierarchy

CUDA has a thread hierarchy that facilitates the decomposition of an application into a set of parallel tasks. Figure *2.1.2* illustrates the thread hierarchy. When a CUDA *kernel* launches, a *grid* is allocated in the GPU context. The relation between a kernel and a grid is similar to the relation between a program and a process. A grid is an executing instance

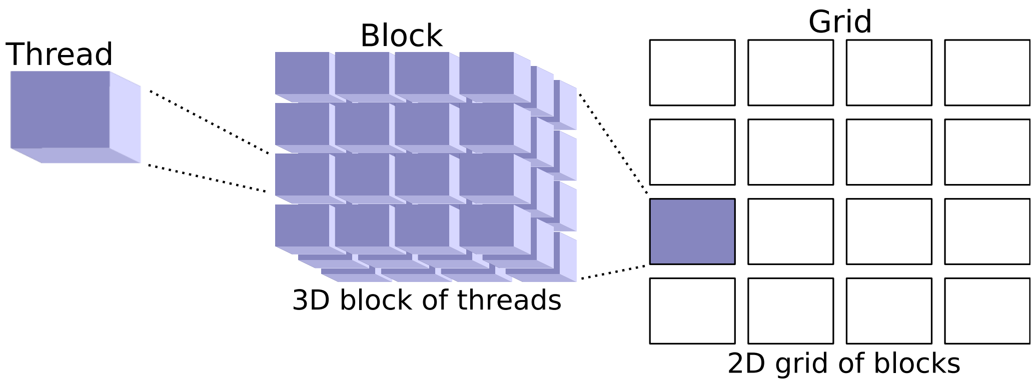


Figure 2.1.1: CUDA thread hierarchy [12].

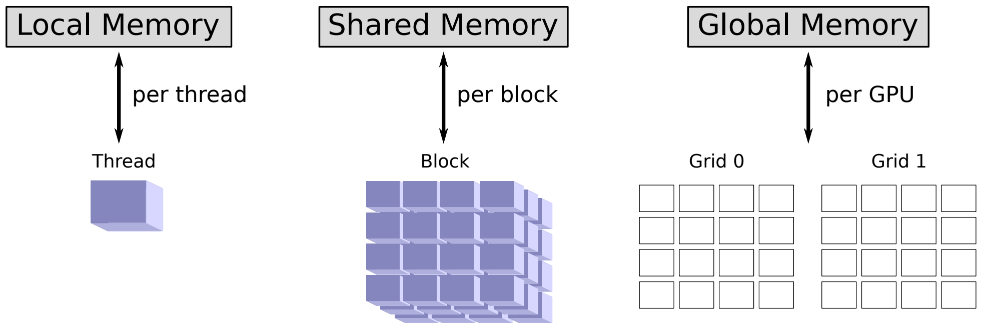


Figure 2.1.2: CUDA memory hierarchy and its relation with the thread hierarchy [12].

of a kernel. A grid consists a set of *blocks*, also known as *Cooperative Threads Arrays* (CTAs). Each block defines an independent task that executes in TLP fashion. Intercommunication among blocks is not possible because the execution order for blocks is not strictly defined. *Threads* in a block can cooperate in DLP fashion.

Each thread executes the kernel once. Two new keywords, *blockIdx* and *threadIdx*, uniquely identify each block in a grid and each thread in a block. These identifications are generally used for task division, allowing the kernel to assign different tasks for different threads.

2.1.2 Memory Hierarchy

The CUDA memory hierarchy corresponds to the thread hierarchy as depicted in Figure 2.1.2. Each thread has access to a private *local memory* for storing large data that does not

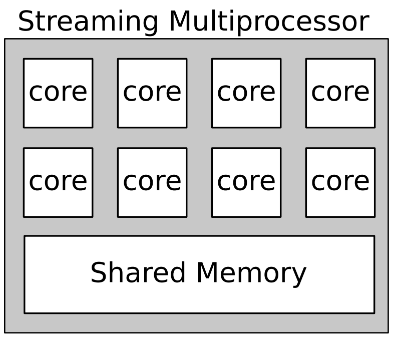


Figure 2.1.3: Logical structure of NVIDIA streaming multiprocessors of compute capability 1.x [13].

fit into its *registers*. All threads in a block have access to a *shared memory*, which is private to the block, to cooperate on a computation. *Global memory* is visible to all threads. The CPU host can only access the global memory. Therefore, parameters and data used by a kernel are initially located in the global memory.

2.1.3 Streaming Multiprocessors

The thread and memory hierarchies are mapped onto *streaming multiprocessors* (SMs) on a GPU. A SM basically contains a set of processor cores and a shared memory (see Figure 2.1.3). Each SM can concurrently execute multiple blocks if registers and shared memory are sufficient. Logically, all threads are executed in parallel. Physically, threads are executed in *warps*, which are groups of 32 threads. A *warp scheduler* on the SM selects and executes a warp at every issuing cycle. Multiple cores execute the same instruction for different threads in a warp. This execution model is called *single-instruction multiple-threads* (SIMT). Whenever a warp engages in a high latency operation, the warp scheduler can switch to another warp for efficient use of issuing cycles [13].

2.1.4 Mapping CUDA to Distributed Systems

The thread and memory hierarchy can be easily mapped onto a distributed system. Figure 2.1.4 compares the system diagram of a GPU to a distributed system to show the similarities between them. The *host* in the distributed system represents a machine that

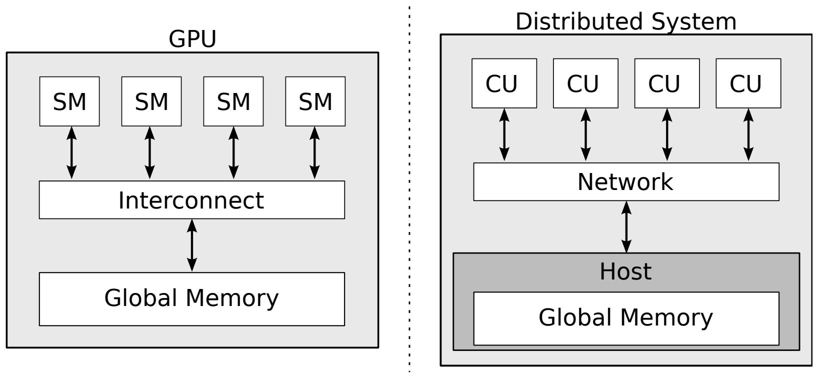


Figure 2.1.4: Comparison of a GPU (left) and a distributed system (right).

initiates a kernel execution. Each *compute unit* (CU) can represent a machine or a processor. The network can be the interconnection on a motherboard or an Ethernet network.

In Phalanx, each CU performs the work of a SM in a smaller scale. A CU computes a block at a time. Messaging between the CUs and the host uses MPI. The support for different types of network depends on the installed MPI implementation. For instance, OpenMPI supports shared memory multiprocessors, Ethernet, InfiniBand, and Myrinet [7].

2.2 Compiling CUDA Kernels for Distributed Systems

To execute CUDA kernels in a distributed system, a *PTX-to-LLVM compiler* translates CUDA kernels at the PTX level. PTX (parallel thread execution) is a virtual instruction set for CUDA. PTX is portable across current and future generations of GPU [12]. Compiling at the PTX level allows programmers to select any high-level programming languages that can be reduced to PTX. NVIDIA officially supports C/C++, FORTRAN, and OpenCL. Chapter 3 discusses the PTX-to-LLVM compiler in detail.

2.3 Emulation of CUDA

The virtual architecture of CUDA cannot be directly mapped onto a distributed system. Also, not all PTX instructions can be mapped to CPU instructions. The Phalanx runtime system emulates the scheduling of blocks onto CUs and the operation of some PTX instructions. Chapter 4 discusses the runtime system. Although the memory system is part of the runtime system, it is discussed separately in Chapter 5 due to its complexity.

2.4 Design for Heterogeneity

Phalanx has a portable design to ensure support for heterogeneity. Phalanx supports heterogeneity for different CPU architectures, operating systems and data-models. The PTX-to-LLVM compiler and the runtime system are written using Python and C++, respectively.

MPI provides messaging functionality in an abstract API. Different MPI implementations support a different set of system configuration. For instance, OpenMPI support nodes of mix architectures. The case studies in Chapters 6 and 7 use a cluster of x86-32 and x86-64 machines.

Phalanx relies on LLVM (low-level virtual machine)[1] for heterogeneous code generation. First described by Lattner [14], LLVM is a compiler infrastructure that uses multi-stage optimization. LLVM accepts a low-level intermediate representation (LLVM-IR) that uses a virtual instruction set in *static single assignment* (SSA) form. Now, LLVM is an umbrella project consisting of many advanced optimization algorithms, language frontends and code generation backends. The code generation backends can transform LLVM-IR into the corresponding instruction set for a wide range of CPU architectures, including x86, ARM, MIPS, Sparc, and PowerPC. An experimental PTX backend has been added recently. In the future, Phalanx can use this new feature to implement CPU and GPU heterogeneity.

[1]http://llvm.org

2.5 The Workflow of Phalanx

The following proposes a flow for CUDA-based programming for distributed heterogeneous systems. Given a CUDA kernel source file written in CUDA-C, NVCC, from the NVIDIA CUDA Toolkit[2], is used to compile the kernel source file to a PTX assembly file. The PTX-to-LLVM compiler, a component of the Phalanx framework, translates the PTX assembly file to LLVM-IR. LLVM generates assembly code for the target architecture from the LLVM-IR. At this point, the assembly code is usually transferred to the target machine. To generate the *worker executable*, the assembly code is linked against the Phalanx runtime system, which is in the form of a shared library on the target machine. On the host machine, a *host executable* remotely executes worker executables. It is a simple C++ program that uses the Phalanx API for scheduling tasks onto the worker executables and does not perform the actual computation of the kernel. Figure 2.5.1 illustrates the proposed workflow for preparing a CUDA kernel for Phalanx-based distributed computing.

[2]http://developer.nvidia.com/cuda-downloads

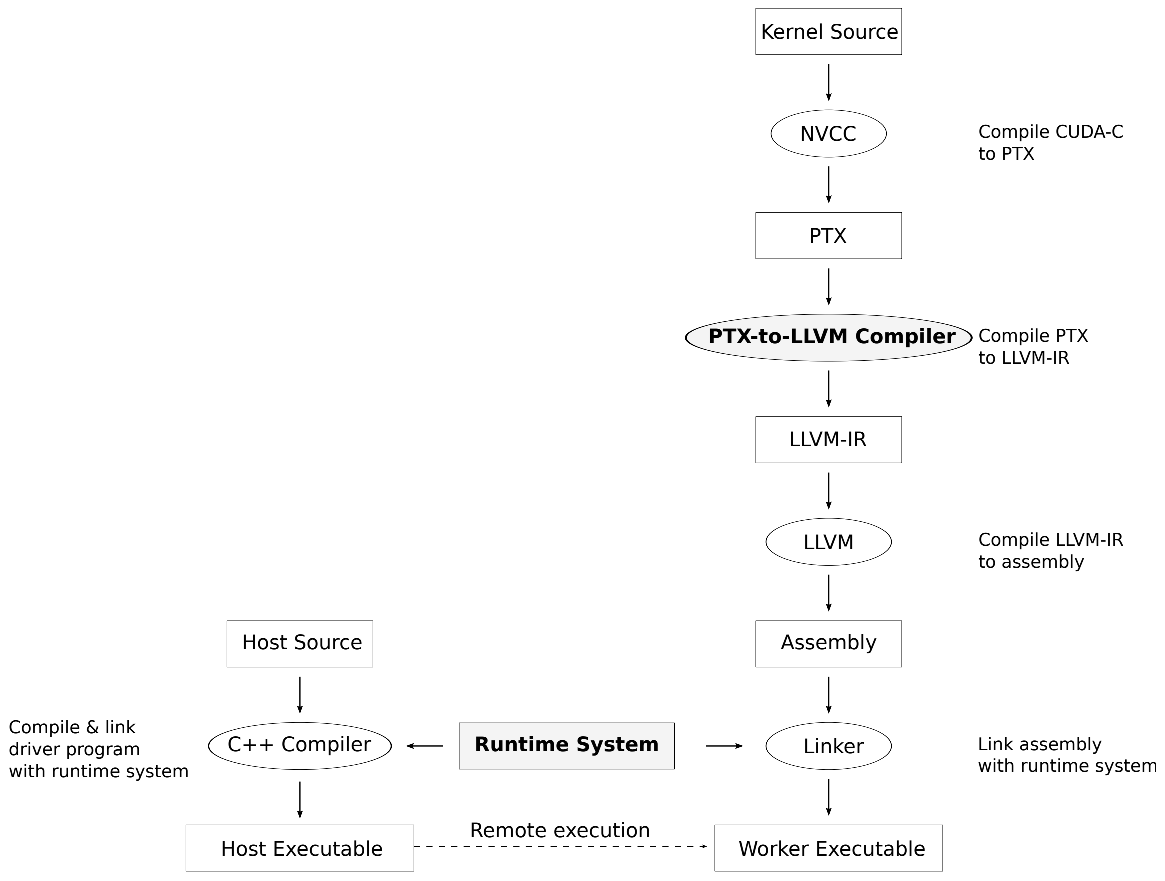


Figure 2.5.1: The flow of CUDA-based programming for distributed heterogeneous system enabled by Phalanx.

Chapter 3

The PTX-to-LLVM Compiler

The PTX-to-LLVM compiler transforms PTX into LLVM-IR for heterogeneous code generation. The compiler restructures PTX instructions into code that suits the execution model of CPUs. The manycore architecture of a SM and the multicore architecture of a CPU is very different. The PTX execution model cannot be directly mapped to the execution model of the CPU. Section 3.1 discusses the characteristics of the PTX execution model. Section 3.2 discusses a refined execution model to allow CPU execution of CUDA kernels. Section 3.3 discusses the handling of register and memory states.

3.1 Execution Model of PTX

The PTX defines an execution model for running a massive number of parallel threads. At each issuing cycle, a warp of 32-threads is scheduled for execution. Figure 3.1.1 illustrates the logical execution of a warp. In the figure, a warp is executing a kernel of 30 instructions. Initially, all threads in the warp execute the first ten instructions in basic-block *A*. Each basic-block has only one execution path. Branching instructions can occur only at the last instruction of a basic-block. Logically, all threads execute in parallel. Instruction 10 is a branch that splits the warp into two halves. The first half-warp executes basic-block *B*. The second half-warp executes basic-block *C*. The diverged execution path forces serial execution of half-warps instead of parallel execution of all threads. Finally, a branch instruction at the end of basic-block *B* and *C* reconverges the execution path to basic-block *D*.

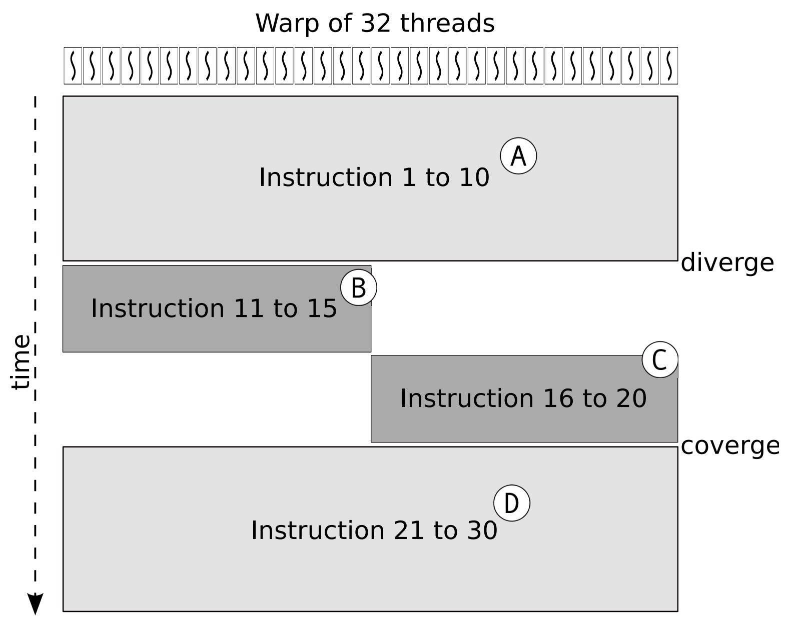


Figure 3.1.1: Execution of a warp.

3.1.1 Branching and Warp Divergence

When threads in a warp are at different execution path due to branching, the warp is divergent. At each issuing cycle, the warp scheduler can execute only a subset of threads that have the same instruction pointer. Threads that have a different instruction pointer are disabled. The warp scheduler iterates over each execution path sequentially, using multiple issuing cycles to complete the execution for a diverged warp. Since each path uses one issuing cycle, the worst case, in which all 32-threads have distinct instruction pointers, requires 32 issuing cycles for executing one warp. Therefore, warp divergence imposes a significant penalty for performance [12].

3.1.2 Thread Barrier

In PTX, barriers allow threads in a block to synchronize and cooperate. Threads reaching the barrier must wait until all threads in the block have reached the same barrier before resuming the execution. The barrier also guarantees all previous memory modifications are visible by all threads.

3.2 Refined Execution Model

Straiten *et al.*[10] discovered a simple transformation of CUDA kernels that allows efficient execution on CPUs. Their method converts CUDA kernels at source level by wrapping code segments between synchronization points–entry, exit and barriers–with a loop that iterates over every threads of the block. Deimos *et al.*[11] use a similar method, but the transformation is applied at PTX level and uses complex control-flow analysis for optimization.

```
.entry _Z21matrixMultipliyKernelPfS_S_i (
    .param .u32 __cudaparm__Z21matrixMultipliyKernelPfS_S_i_A,
    .param .u32 __cudaparm__Z21matrixMultipliyKernelPfS_S_i_B,
    .param .u32 __cudaparm__Z21matrixMultipliyKernelPfS_S_i_C,
    .param .s32 __cudaparm__Z21matrixMultipliyKernelPfS_S_i_N)
{
    .reg .u16 %rh<6>;
    .reg .u32 %r<33>;
    .reg .f32 %f<5>;
    .reg .pred %p<5>;
$LBB1__Z21matrixMultipliyKernelPfS_S_i:
    mov.u16     %rh1, %ctaid.x;
    ...
    ld.param.s32    %r7, [__cudaparm__Z21matrixMultipliyKernelPfS_S_i_N];
    set.le.u32.s32  %r8, %r7, %r6;
    ...
    @%p1 bra    $Lt_0_2306;
    bra.uni     $LBB10__Z21matrixMultipliyKernelPfS_S_i;
$Lt_0_2306:
    mov.u32     %r14, 0;
    ...
    @%p2 bra    $Lt_0_3842;
    mov.s32     %r15, %r7;
    ...
    ld.param.u32    %r20, [__cudaparm__Z21matrixMultipliyKernelPfS_S_i_B];
    add.u32     %r21, %r20, %r17;
    ...
    ld.param.u32    %r24, [__cudaparm__Z21matrixMultipliyKernelPfS_S_i_A];
    add.u32     %r25, %r22, %r24;
    ...
$Lt_0_3330:
    ld.global.f32   %f2, [%r25+0];
    ld.global.f32   %f3, [%r21+0];
    mad.f32     %f1, %f2, %f3, %f1;
    ...
    @%p3 bra    $Lt_0_3330;
    bra.uni     $Lt_0_2818;
$Lt_0_3842:
    mul.lo.s32  %r16, %r7, %r6;
$Lt_0_2818:
    ld.param.u32    %r28, [__cudaparm__Z21matrixMultipliyKernelPfS_S_i_C];
    add.s32     %r29, %r16, %r4;
    ...
    st.global.f32   [%r31+0], %f1;
$LBB10__Z21matrixMultipliyKernelPfS_S_i:
    exit;
$LDWend__Z21matrixMultipliyKernelPfS_S_i:
}
```

Figure 3.2.1: Sample separation of PTX kernel entry into subkernels.

Comparing with the method of Deimos *et al.*, Phalanx uses a relatively naive method. First, the compiler decomposes a kernel into basic-blocks by separating at each label.

Then, the compiler further decomposes these basic-blocks at every branch, memory, and control instructions. Control instructions include exit and barrier instructions. At this point, a kernel is partitioned into a sequence of *subkernels*. Subkernels are basic-blocks that can have only branch, memory, and control instructions as the last instruction. Figure 3.2.1 shows the PTX corresponding to the kernel defined in Listing 3.1. In the figure, the dotted lines denote the boundary of subkernels. The boldfaced lines highlight the memory, branch, and other control instructions.

Unlike the method of Deimos *et al.*, Phalanx does not join the partitioned subkernels. Instead, each subkernel is generated as an individual function in the LLVM-IR. Each PTX instruction in the subkernels is translated into a vector instruction that represents the execution of a warp. Thus, the vectors are 32-elements wide. Phalanx relies on LLVM to split or join vectors to match the supported vector size of the target architecture. Two loops are inserted for the vectorized subkernels. The first loop encloses all arithmetic instructions. Only the last instruction in a subkernel is non-arithmetic. The second loop encloses the last instruction. The loops iterate over a range of warps. The range is supplied as parameters to the subkernel function, indicating the start and the end of the range. Figure 3.2.2 is a control-flow graph of a sample subkernel function. In the figure, the first shaded box represents the body of the first loop. The second shaded box represents the body of the second loop.

The resulting subkernel function does not account for warp divergence. Each vector instruction consists of the operations of 32 threads. PTX uses *single-instruction multiple threads (*SIMT*)* execution. The SM automatically handles warp divergence. The CPU uses SIMD execution. The programmer must explicitly control divergence in vector operations. To handle warp divergence, the PTX-to-LLVM compiler generates two special functions for saving register contents and restoring the saved contents to the registers. The compiler does not generate code for scheduling subkernel functions or for applying the register saving and restoring functions. The runtime system is responsible for these features. The discussion of the scheduler is in Chapter 4.

Listing 3.1: Sample matrix multiplication CUDA-C code.

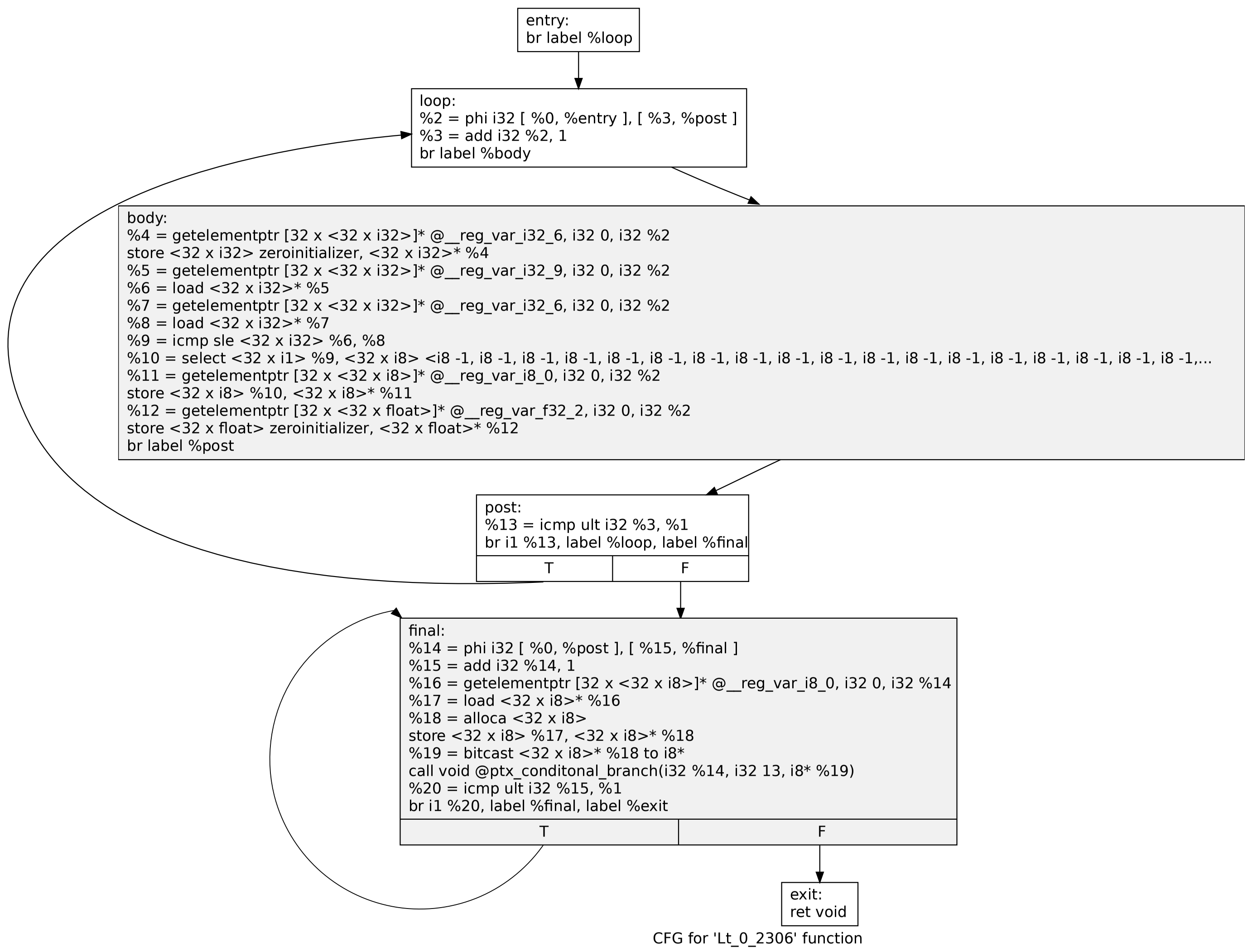
```
__global__
void matrixMultipliyKernel(float A[], float B[], float C[],
   int N){
    const int i = threadIdx.x + blockIdx.x * blockDim.x;
    const int j = threadIdx.y + blockIdx.y * blockDim.y;

    if( i>=N || j>=N ) return;

    float result = 0;
    for( int k=0; k<N; ++k ){
        result += A[k+j*N] * B[i+k*N];
    }
    C[i+j*N] = result;
}
```

3.3 Handling Register and Memory States

The definitions of register and memory in PTX is different from those of CPU architectures. The compiler must adjust for the difference to allocate register and memory accordingly.

3.3.1 Registers

The PTX does not define the upper limit of register use. Its instruction set uses a loose SSA form, in which register can be assigned multiple times as long as each assignment is located at a different basic-block. The Phalanx PTX-to-LLVM compiler must perform register allocation for each kernel to determine the minimum register count. The algorithm for register allocation is simple. Since registers are typed in PTX, the following steps are repeated for each type. First, the compiler scans for registers that are used by multiple subkernels. These registers are added to the *final set* without further processing. Second, it searches for the live ranges of the remaining registers. A live range contains the first and the last occurrences of a register. Then, it continues with the algorithm in



```
entry:
br label %loop

loop:
%2 = phi i32 [ %0, %entry ], [ %3, %post ]
%3 = add i32 %2, 1
br label %body

body:
%4 = getelementptr [32 x <32 x i32>]* @__reg_var_i32_6, i32 0, i32 %2
store <32 x i32> zeroinitializer, <32 x i32>* %4
%5 = getelementptr [32 x <32 x i32>]* @__reg_var_i32_9, i32 0, i32 %2
%6 = load <32 x i32>* %5
%7 = getelementptr [32 x <32 x i32>]* @__reg_var_i32_6, i32 0, i32 %2
%8 = load <32 x i32>* %7
%9 = icmp sle <32 x i32> %6, %8
%10 = select <32 x i1> %9, <32 x i8> <i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1, i8 -1,...
%11 = getelementptr [32 x <32 x i8>]* @__reg_var_i8_0, i32 0, i32 %2
store <32 x i8> %10, <32 x i8>* %11
%12 = getelementptr [32 x <32 x float>]* @__reg_var_f32_2, i32 0, i32 %2
store <32 x float> zeroinitializer, <32 x float>* %12
br label %post

post:
%13 = icmp ult i32 %3, %1
br i1 %13, label %loop, label %final
T | F

final:
%14 = phi i32 [ %0, %post ], [ %15, %final ]
%15 = add i32 %14, 1
%16 = getelementptr [32 x <32 x i8>]* @__reg_var_i8_0, i32 0, i32 %14
%17 = load <32 x i8>* %16
%18 = alloca <32 x i8>
store <32 x i8> %17, <32 x i8>* %18
%19 = bitcast <32 x i8>* %18 to i8*
call void @ptx_conditonal_branch(i32 %14, i32 13, i8* %19)
%20 = icmp ult i32 %15, %1
br i1 %20, label %final, label %exit
T | F

exit:
ret void
```

Figure 3.2.2: Control-flow graph of a sample subkernel function in LLVM-IR.

Listing 3.2 to fill the *final set*. Finally, the *final set* contains the minimum set of registers.

Listing 3.2: Pseudocode for register allocation.

```
foreach instruction {
    foreach register in active set {
        if register leaves its live range {
            remove register from active set
            add register to unused set
        }
    }
    foreach operand used in the current instruction {
        if unused set is not empty {
            remove a register from unused set and add it to
                active set
        } else {
            allocate a new register and add it to active set
        }
        assign the register for the operand
    }
}
foreach register in unused set {
    add to final set
}
```

Registers are statically allocated in the data segment of the worker executable. Each register is allocated as an array of 1024 elements, which is the maximum number of threads per block. Threads belonging to a warp have registers located consecutively in the array, allowing efficient SIMD load/store operations.

3.3.2 Memory

Shared memory is also statically allocated in the data segment. Since the worker executable computes one CUDA block at a time, there is only one copy of shared memory. Unlike CUDA, the capacity of the shared memory is not fixed. A kernel can allocate as much shared memory as permitted by the amount of primary memory on the running machine.

3.4 Future Works

The PTX-to-LLVM compiler is still at an early stage, supporting only a subset of instructions of CUDA compute capability 1.x. It also lacks support for local and constant memory. Aside from completing the support for the full PTX specification, future works should also add various optimization passes for better register allocation and performance. A control-flow analysis that predicts warp divergence and re-convergence would improve execution efficiency.

Chapter 4

The Runtime System

The Phalanx runtime system performs different functions in a *host process* and in a *worker process*. Figure 4.0.1 illustrates different components of the runtime system. A host process uses the *remote kernel manager* in the runtime system for scheduling remote kernel execution on a distributed system. By doing so, worker processes are spawned on each compute unit (CU) in the distributed system. CUs can be cores in a multicore CPU or remote machines connected to an Ethernet network. In a worker process, a *warp scheduler* in the runtime system schedules executions of subkernel functions. Subkernel functions depend on the *PTX emulator* for the implementations of memory, branch, exit and barrier instructions. Memory instructions are redirected to the *memory system*. Global memory requests are translated to MPI messages. Detail discussion of the memory system is in Chapter 5. Other components of the runtime system are discussed in the following sections.

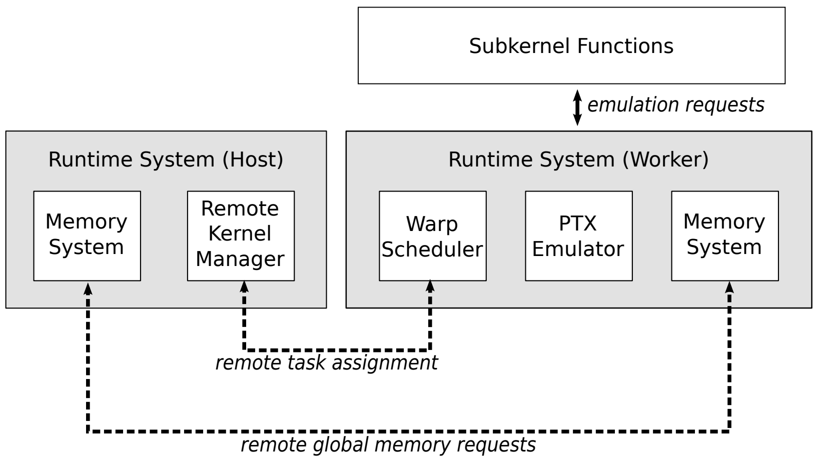


Figure 4.0.1: System diagram of the runtime system.

4.1 Remote Kernel Manager

A host process executes kernel remotely on CUs in the distributed system. For comparison, Listing 4.1 and Listing 4.2 show sample codes of kernel invocations in Phalanx and in CUDA runtime API, respectively. Listing 4.3 shows the declaration of the corresponding kernel being invoked.

Listing 4.1: A kernel invocation example in Phalanx.

```
int main(int argc, char ** argv){
    // initialize phalanx runtime system
    phalanx::communicator.init(argc, argv);
    // set available hosts in the distributed system
    phalanx::communicator.set_hosts("169.254.8.58,169.254.8.95
       ");
    // CUDA specifics: setup block dimension
    dim3 blockDim(3, 4, 5);
    // CUDA specifics: setup grid dimension
    dim3 gridDim(6, 7);
    // allocate data arrays
    int N = 1024;
    int * A = new int[N];
    int * B = new int[N];
    // populate the data arrays
    populateInput(A, B, N);
    // allocate an array for passing parameters
    void * parameter[] = {&A, &B, &N};
    // launch kernel on remote nodes
    phalanx::launchKernel(
        path_to_worker_process, // location of worker
           executable
        gridDim,     // grid dimension of the kernel
        blockDim,    // block dimension of the kernel
        parameters, // list of parameters
        sizeof(parameters)/sizeof(void*), // number of
           parameters
        number_of_process //total number of remote processes
        );
    // clean up
    delete [] A;
    delete [] B;
    // shutdown phalanx runtime system
    phalanx::communicator.finalize();
}
```

Listing 4.2: A kernel invocation example in CUDA runtime API.

```
int main(int argc, char ** argv){                                              1
    // CUDA specifics: setup block dimension                                   2
    dim3 blockDim(3, 4, 5);                                                    3
    // CUDA specifics: setup grid dimension                                    4
    dim3 gridDim(6, 7);                                                        5
    // allocate data arrays                                                    6
    int N=1024;                                                                7
    int * A = new int[N];                                                      8
    int * B = new int[N];                                                      9
    // populate the data arrays                                                10
    populateInput(A, N);                                                       11
    // make space in GPU global memory                                         12
    int * dA;                                                                  13
    int * dB;                                                                  14
    cudaMalloc(&dA, sizeof(int)*N);                                            15
    cudaMalloc(&dB, sizeof(int)*N);                                            16
    // transfer CPU data to GPU global memory                                  17
    cudaMemcpy(dA, A, sizeof(int)*N, cudaMemcpyHostToDevice);                  18
    // invoke kernel for execution in GPU                                      19
    cudaKernelFunction<<<gridDim, blockDim>>>(A, B, N);                        20
    // transfer GPU data back to CPU                                           21
    cudaMemcpy(B, dB, sizeof(int)*N, cudaMemcpyDeviceToHost);                  22
    // clean up GPU memory                                                     23
    cudaFree(dA);                                                              24
    cudaFree(dB);                                                              25
    // clean up CPU memory                                                     26
    delete [] A;                                                               27
    delete [] B;                                                               28
}                                                                              29
```

Listing 4.3: An example of CUDA Kernel declaration.

```
__global__
void cudaKernelFunction(const int A[], int B[], int N);
```

In CUDA runtime API, programmers explicitly state the allocation of global memory on the GPU. The API provides a set of memory transfer functions to copy data between CPU and GPU. Lines 10-12 in Listing 4.2 show the code for allocating global memory in the GPU and copying array *A* to the GPU. Line 13 launches the kernel in a similar fashion to a regular function call in C/C++, but with a special kernel configuration before the parameter list. Line 14 copies the output data array *B* back to the host. Lines 15-16 release the global memory allocated in the GPU.

In Phalanx, the global memory resides in the host process. Unlike CUDA runtime API, Phalanx does not require any additional transfer from CPU memory to global memory. Worker processes can access any memory in the host process through remote global memory requests. However, the current implementation does not protect against out-of-bound memory access for remote global memory requests. So, worker processes may access to any memory location, causing security concerns.

The Phalanx runtime system requires explicit initialization and termination. These procedures correspond to lines 2 and 9, respectively, in Listing 4.1. They are required to properly initialize and terminate the underlying MPI system. Line 3 provides a list of machines on which worker processes are spawned. Here, the machines are specified by IP addresses. A machine may contain multiple CUs. If no machine are provided, worker processes are spawned on the same machine on which the host process is running. MPI assigns worker processes to process slots. Specification of the number of available process slots differs among MPI implementations. For OpenMPI, a *hostfile* is supplied using the MPI launch script (*mpirun* or *mpiexec*) to list the number of slots on each host. User should avoid over-commiting–assigning more than one process to a logical processor core. Over-committing may degrade performance. The best practice is to assign one process slot per logical processor core.

The call to *launchKernel* at lines 11-16 signals the Phalanx runtime to spawn worker processes on the given machines. The first parameter is a path string that tells MPI the location of the worker executable. This path must be the same across all remote machines. The second and third parameters configure the grid and block dimension of the kernel, respectively. The fourth parameter supplies a list of pointers to serve as kernel parameters. The fifth parameter declares the number of kernel parameters. The last parameter specifies the total number of worker processes to spawn.

Kernel invocation in Phalanx is synchronous, unlike the asynchronous kernel invocation in CUDA runtime API. The remote kernel manager takes over until the kernel finishes and all worker processes have terminated. This is necessary to handle incoming task requests from the worker processes. Section 4.4 further discusses the task requests from worker processes.

4.2 Warp Scheduler

A worker process begins execution by sending a task request to the host process. The task request asks the host process to assign a pending CUDA block for the worker process. Each block is computed by a worker process. When the block has been completed, the worker process sends another task request. This procedure repeats until all blocks in the grid have been completed.

Once the worker process have received a block, it starts to schedule warps for subkernel executions. The warp scheduler maintains a set of state variables for every thread in the block.

live A Boolean value that, if True, indicates the thread has not completed its execution.

diverged A Boolean value that, if True, indicates that the thread is disabled and has a different *subkernel ID* than the non-diverged threads of the warp.

subkernel ID The identifier of the next *subkernel* to execute.

restore context A pointer to a structure containing all saved register values of a *diverged* thread. It has a *null* pointer if the thread is not diverged.

Diverged thread maintains a *restore context* for preserving register values. Subkernels execute every thread in the warp without checking their divergence state. As a result, threads that have been disabled due to divergence will also execute in the subkernels. Their registers could be corrupted in the process. When diverged threads re-enable, their registers are restored by the content of their *restore contexts.*

Listing 4.4 shows the pseudocode of the scheduler algorithm. First, the scheduler executes all warps for the first subkernel. As long as there are *live* threads, the scheduler finds ranges of warps that have the same subkernel ID and execute the subkernel for these ranges of warps. After each subkernel execution, the scheduler commits any pending memory transactions and increments the subkernel ID for each executed thread.

Listing 4.4: Algorithm of the runtime scheduler.

```
execute current subkernel of all warps
commit memory transaction
increment subkernel id for all warps
while live threads count is not 0 {
    begin := 0
    end := 0
    while begin < warp count {
        firstWarp := warp with id==begin
        if firstWarp is not alive {
            begin := begin + 1
        }
        currentSubkernel := subkernel of firstWarp
        end := begin + 1
        while end < warp count and currentSubkernel ==
           subkernel of lastWarp {
            end := end + 1
        }
        execute current subkernel of warp range [firstWarp ,
           lastWarp]
        commit memory transaction
        increment subkernel id for warp range [firstWarp ,
           lastWarp]
    }
}
```

4.3 PTX Emulator

Some PTX instructions have no direct translation into assembly of the target architecture. These PTX instructions are converted into function calls that reference handling routines in the runtime system.

4.3.1 Branch Instructions

Branch instructions are handled once per warp. For *conditional branch*, the runtime checks the predicates of all threads in the warp. If the predicates are the same–all True or all False, the warp is not diverging and the branch is treated as a *uniform branch*. If, however, the predicates are different for some threads, the warp diverges. The branching threads are disabled by marking the *diverged* flag and creating a *restore context*.

PTX allows a branch instruction to be marked as *uniform.* A *uniform branch* guarantees that all threads in the warp participate the branch. Therefore, it hints the runtime to create a convergence point. When executing a *uniform branch* in a diverged warp, the runtime swaps the running threads and the diverged threads. This allows the diverged threads to reach the same subkernel so that the warp converges, improving the efficiency of warp execution. For non-diverged warps, simply changing the current *subkernel ID* to the destination *subkernel ID* is sufficient.

4.3.2 Memory Instructions

Instead of servicing each global memory requests immediately, they are serviced after the subkernel execution. This allows the runtime system to coalesce the requests from multiple warps and optimizes them to reduce network usage. Detail discussion of the optimization is in Chapter 5.

Shared memory store requests must be handled by the runtime system because the subkernel execute every thread in the warp, including the disabled threads. Failing to discard store requests from *non-live* or *diverged* threads could corrupt shared memory content.

4.3.3 Exit Instruction

Exit instructions are handled once per warp. The runtime system clears the *live* flag of all running threads in the warp and restores diverged threads, if any.

4.3.4 Barrier Instruction

In PTX, a barrier instruction guarantees that all previous memory transactions have been completed and all threads reaches the same instruction. The runtime system disables all threads reaching that barrier and restores any diverged threads that have not reached the barrier. When all threads have reached the barrier, the runtime restores all threads for execution.

4.4 Remote Requests

Worker processes send three types of remote request.

Task request A worker process actively asks for new task from the host process whenever it is idle.

Parameter request A worker process asks for parameter values from the host process.

Global memory request A worker process loads from or stores to global memory, which resides in the host process.

Figure 4.4.1 demonstrates a sample sequence of remote requests. The host process spawns a new worker process and sends the kernel configuration (block and grid dimensions). Then, the host process switches to passive mode, handling any incoming requests until all blocks of the grid have been completed. After receiving the kernel configuration, the worker process sends a *task request*. In reply, the host process assigns a block index (*blockIdx*) to the worker process, which begins to compute the kernel block indicated by the received block index. During the lifetime of the kernel block, the worker process asks for parameter values using *parameter requests* and it asks for global memory using *global memory requests*. Upon completion of the kernel block, the worker process sends another *task request*. If all blocks have been completed, the host process signals the worker process to terminate.

Since parameters in a CUDA kernel are copied-by-value, they are unchanged throughout the lifetime of a kernel grid. A worker process can safely cache a parameter value after the first request. This reduces the number of parameter requests and consumption of network bandwidth.

Due to the complexity and importance of the *global memory requests*, they are discussed separately in Chapter 5.

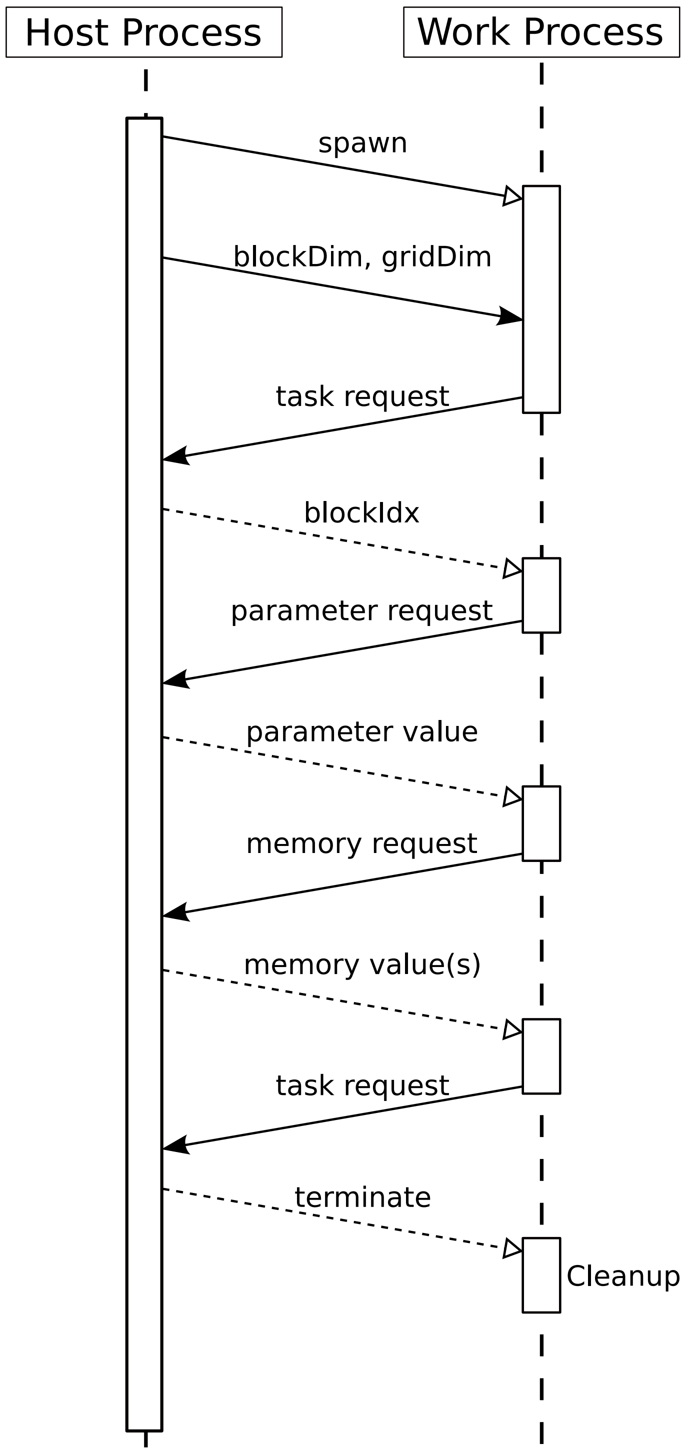


Figure 4.4.1: A sequence diagram illustrating the message-passing of remote requests between a host process and a worker process.

Chapter 5

The Memory System

The memory system is part of the runtime system. It handles all memory requests from the subkernels. The performance of memory requests governs the maximum performance of a Phalanx application. Network bandwidth limits the maximum memory throughput for a Phalanx application. Comparing to the maximum throughput of PCI-express available to a GPU, a typical Ethernet network is significantly slower. With the limited network throughput, the memory system must optimize for maximum network utilization. The rest of the chapter discusses the implementation of memory system.

5.1 Memory Operations

Listing 5.1 shows a simple parallel-prefix-sum kernel. This sample code shows all four types of memory operations. The right-hand-side (RHS) of line 5 is converted to a global memory load. The assignment of the same line translates to a shared memory store. At line 10, the assignment translates to a global memory store. The runtime system handles these three types of memory operations. The shared memory load at the RHS of line 8 is handled in the generated assembly using a simple load instruction of the target architecture.

Listing 5.1: A simple parallel prefix sum kernel

```
__global__                                          1
void prefixSumKernel(const int A[], int B[]){       2
    __shared__ int smem[1024];                      3
    int result=0;                                   4
    // shared memory preloading                     5
    smem[threadIdx.x] = A[threadIdx.x];             6
    // barrier                                      7
    __syncthreads();                                8
    for(int i=0; i<threadIdx.x; ++i){               9
```

```
        result += smem[i]; // load from shared memory
    }
    // store to global memmory
    B[threadIdx.x] = result;
}
```

5.1.1 Global Memory

Global memory requests are not serviced immediately but after the execution of the subkernel. This allows all memory requests to be coalesced and optimized as described in Section 5.4.

Since a subkernel does not isolate disabled threads, which have exited or have diverged to another subkernel, disabled threads also generate memory requests. The runtime system is responsible to filter out these invalid requests.

A worker process forwards any global memory request to the host process through MPI messages. The request contains the data type of the requesting elements, the starting address of the elements and the number of elements to load or to store. For a load request, the host process loads the data at the starting address. The addresses of subsequent elements are computed by incrementing the starting address by the byte length of the data type. The host process replies with a MPI message filled with the loaded data. For a store request, the host process continues to listen for the next message from the worker process. The next message contains the data to be stored. The host process directly stores the received data to the starting address.

5.1.2 Shared Memory

Shared memory requests are serviced immediately. Shared memory loads do not depend on the runtime system. They are converted directly into simple load instructions for the target architecture. The runtime system only handles shared memory stores. Similar to global memory requests, the runtime must filter out invalid shared memory store requests generated by disabled threads. Invalid shared memory loads are not removed. Since diverged threads have kept a copy of the register values, these invalid loads cannot corrupt the registers of these threads.

5.2 Handling Heterogeneous Data Model

Phalanx supports heterogeneous data model in the distributed system. The CUDA and LLVM virtual architectures share the same data types. All integers are in 2's complement representation. CUDA supports only 8, 16, 32 and 64-bit integers. LLVM supports any integer of bit length 1 to $2^{23}-1$. For floating-point numbers, both use IEEE754 compliant types. For memory address, Phalanx forces all addresses to be 64-bit wide. On a 32-bit host machine, 64-bit addresses are simply truncated. Any architecture that implements 2's complement integer representations and IEEE754 single and double precision floating-point representations can be used in Phalanx.

Not all MPI implementations support data model heterogeneity. OpenMPI provides automatic conversion between big and little endians. However, heterogeneity support appears to be incomplete for some features. The *remote-memory access* (RMA) defined in the new MPI standard [6] breaks in OpenMPI when working in a 32/64-bit heterogeneous cluster. Phalanx does not use the RMA feature. Phalanx uses only point-to-point communication of MPI. RMA allows one-sided communication, removing the need to have a passive handler in the host process. The RMA feature can be useful in the future.

5.3 Memory Consistency

The massively parallel execution requires CUDA to adopt a weak memory consistency model [15]. This means modifications in memory are guaranteed to be visible only at barriers. It allows Phalanx to use aggressive optimizations for memory operations and thread scheduling.

5.4 Optimizing Global Memory Requests

Global memory throughput limits the performance of a Phalanx application. Section 5.4.1 discusses the performance impact of memory throughput for any system in general. Sections 5.4.2 and 5.4.3 explain the optimizations used by Phalanx for maximizing global memory throughput.

5.4.1 Principle of Balance

The performance of a system is limited by both computation and I/O throughput. The classical principle of balance states that the best performing system should balance the two throughputs. Given the number of processors is p, their peak compute throughputs are C operations per second and they are connected through a network with bandwidth B bytes per second, the following equation [16, 17] characterizes the balance of such a system:

$$I_{optimal} = \frac{p \cdot C}{B}. \tag{5.4.1}$$

$I_{optimal}$ is the peak arithmetic intensity of a computation. It has units of *operations per byte*. When a computation has an intensity that matches $I_{optimal}$, the system computes as fast as I/O transfer. At this point, the system is the most efficient. Any computation that has an intensity higher than $I_{optimal}$ is compute bounded. If the intensity is lower than $I_{optimal}$, the computation is I/O bounded. A compute bounded computation is preferable because the system is not wasting processor time to wait for I/O.

For distributed systems, the network bandwidth is often significantly slower than the total compute throughput of all processors. This suggests that a suitable computation for distributed systems must have a high arithmetic intensity and the required intensity increases as the number of processors increases. As a result, most computations become I/O bounded.

5.4.2 Caching

Caching can raise the effective I/O bandwidth B_{eff}, thereby reduces $I_{optimal}$. Given a cache with bandwidth B_{cache}, a network with bandwidth B_{net} and the percentage of I/O served from the cache is α, the effective bandwidth B_{eff} is characterized by the following equation:

$$B_{eff} = (1-\alpha)B_{net} + \alpha B_{cache}. \tag{5.4.2}$$

When $B_{cache} > B_{net}$ and $\alpha > 0$, B_{eff} is greater than B_{net}.

Caching exploits data locality and temporarily stores fetched global memory in a worker process. The Phalanx runtime system implements a direct-mapped cache for each worker process. Based on the transfer pattern of the computation and available memory, the cache can be resized accordingly for each machine to optimize α.

The cache is local to each worker process. It could be beneficial for future versions to allow a shared cache for all worker processes on the same machine. It could further reduce the network traffic.

Shared memory is not cached and it is not beneficial to do so. Shared memory is implemented as a static data segment in a worker process. Access to shared memory is direct, whereas access to cache goes through a hash table lookup. Therefore, the cache is slower than the shared memory.

5.4.3 Transaction Size

For any global memory transaction, a worker process sends a request message that contains the data type, address and element count. Whenever the total size of transferring data is not much greater than the size of the request message, memory transaction becomes very expensive. The request message has a constant size of 20-bytes (or 160-bits), without including additional overheads from the MPI implementation or from the network protocol.

When the transaction size is very large comparing to the overheads, network utilization improves. For loading global memory, Phalanx uses a minimum fetch size. By default, each load transaction always fetches at least 4-kB of consecutive data. This over-fetching is beneficial most of the time. The coalesced memory access pattern, which is recommended for CUDA programming [13], guarantees that CUDA kernel loads batches of consecutive data. Moreover, the spatial locality of algorithms increases the chance of consumption of nearby data for each load. Since the cache stores all fetched data, any subsequent load transaction can be reduced into a fast cache load if it refers to cached data.

The runtime system sorts all load requests according to their addresses. It searches for the longest consecutive list of the sorted requests and services them in one transaction. If the transaction size is too small, it will append dummy requests to the list. It repeats until all requests are serviced.

For storing global memory, Phalanx does not mandate a fix transaction size. It only tries to coalesce memory requests to use fewer MPI messages. The runtime system sorts all memory requests according to their addresses. It searches for the longest consecutive list of the sorted requests and services them in one transaction. It repeats until all requests are serviced.

Chapter 6

Case Study: Matrix Multiplication

To demonstrate the feasibility of the Phalanx framework, this chapter and the following chapter present two case studies. In each case study, an application is written for serial and parallel execution. The parallel execution version uses Phalanx. Benchmarks are done to measure the performance gain by parallel execution.

A Core 2 Duo machine and two Core-i7 machines are used in the benchmarks. One of the Core-i7 machines are running in 64-bit mode. Other machines are running in 32-bit mode. All machines are running Ubuntu 10.04 OS.

For the parallel execution benchmark, all three machines are connected to a Gigabit Ethernet switch to form a small distributed system. The setup has a heterogeneous data model that mixes 32-bit and 64-bit processors. The Phalanx host process runs on the Core 2 Duo machine and it schedules worker processes onto the two Core-i7 machines. Each Core-i7 machine has four HyperThread execution cores. Together, the two Core-i7 machines provide 16 logical cores to serve as compute units in the Phalanx setup. The parallel executions are configured to use all 16 compute units with one worker process per unit. The MPI implementation used is OpenMPI 1.4 with TCP/IP based messaging. The serial execution benchmarks are performed on the 32-bit Core 2 Duo machine and the 32-bit Core-i7 machine.

6.1 Application Background

A matrix multiplication application is written using Phalanx. For simplicity, only square matrices are considered. The product P of the multiplication between two $N \times N$ matrices, A and B, is defined as follows:

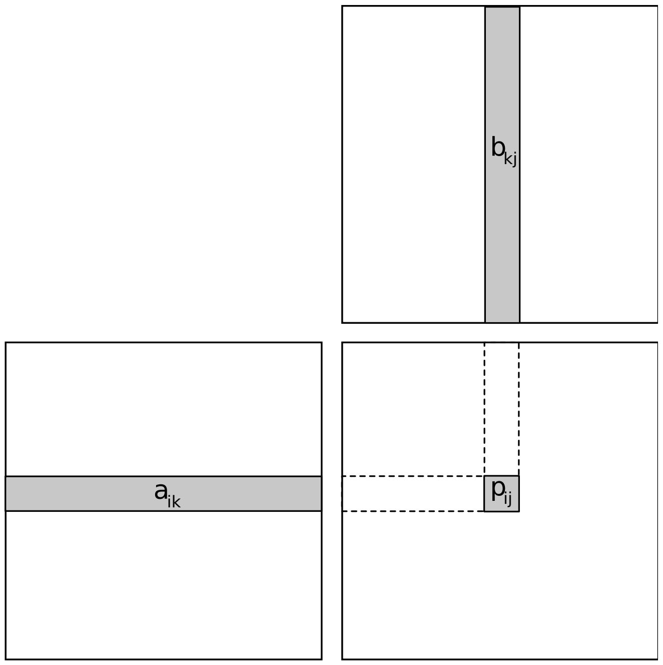


Figure 6.1.1: Illustrating matrix multiplication.

$$p_{ij} = \sum_{k=0}^{N} a_{ik} \cdot b_{kj}. \tag{6.1.1}$$

Each element p_{ij} in the product matrix P at row i and column j is the sum of all products of element-wise multiplication in row i of A and column j of B. Figure 6.1.1 depicts the matrix multiplication algorithm.

6.2 Using Shared Memory

Each output element p_{ij} requires to load $2N$ inputs. The inputs for other elements in the same row or same column overlaps. Reuse of overlapped elements can reduce network traffic for global memory loads. Kirk and Hwu [18] presented a tiled matrix multiplication algorithm. Their algorithm splits the product matrix into tiles and assigns the computation of one tile to each CUDA block. Splitting the product matrix into tiles allows the inputs of each tile to fit into the limited shared memory. For Phalanx, the limitation of shared memory is much higher than for CUDA GPUs. The maximum capacity of shared memory depends on the available size of primary memory for the machine.

6.3 Implementation

The Phalanx implementation of the matrix multiplication supports block size up to 1024 threads, allowing the tile size T to be 32×32. With this configuration, each thread in a block produces a single element in the product matrix. Listing A.2 shows the source code of the matrix multiplication kernel using shared memory for single-precision floats. The kernel fills the shared memory for computing the product of the tile. For each source matrix, the kernel loads $N \times \sqrt{T}$ elements into the shared memory. Afterwards, a barrier ensures all threads of the block have completed the shared memory loading. Each thread of the kernel block computes an element in the corresponding tile.

6.4 Benchmark

Figure 6.4.1 shows the speedups of 16-cores parallel execution using the Phalanx setup with serial execution baselines using the Core 2 Duo and the Core-i7 machines, respectively. It is important to note that the baseline implementation uses a naïve (non-tiled) matrix multiplication algorithm (see Listing A.3), which heavily relies on the processor cache. The x-axis indicates the varying N. The y-axis shows the speedup factors of parallel execution time with respect to the baseline serial execution time.

6.5 Performance Analysis

For single-precision, the speedups saturate at $N = 2048$ because the intensity of the computation is higher than the optimal intensity $I_{optimal}$ (see Section 5.4 for detail discussion of this value). When $N < 2048$, the computation is I/O bounded. When $N > 2048$, the computation is compute bounded. Therefore, the maximum speedups are approximately $12\times$ and $5\times$ comparing to Core 2 Duo and Core-i7 baselines, respectively. For double-precision, the speedups also begin to saturate at $N = 2048$. The maximum speedups are around $9\times$ and $7\times$ comparing to Core 2 Duo and Core-i7 baselines,

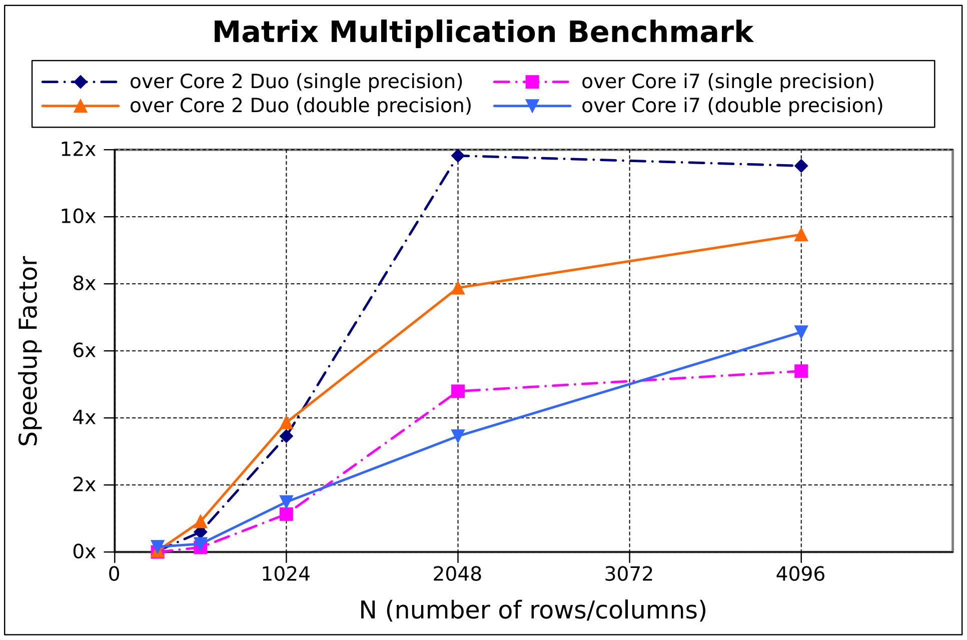


Figure 6.4.1: A benchmark of matrix multiplication showing speedup factor versus matrix size. Speedups are normalized parallel execution time of the Phalanx setup with respect to serial execution baselines using the Core 2 Duo and Core i7 machines.

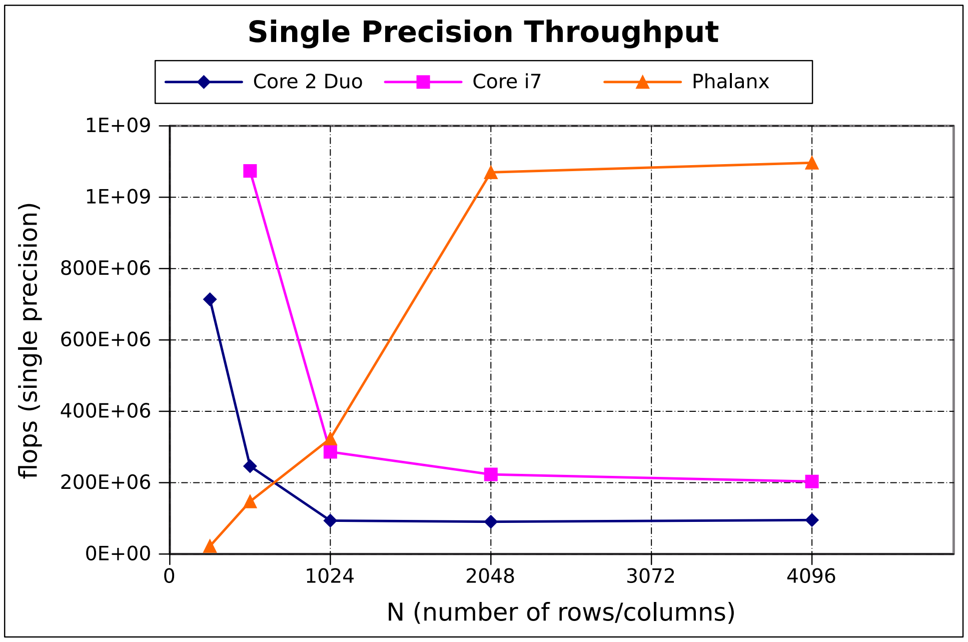
respectively.

Plotting the performance with estimated compute throughputs shows an interesting perspective. Figure 6.5.1 shows the throughputs for serial execution on Core 2 Duo and Core i7, and for parallel execution on the Phalanx-based distributed system. The following equation estimates the number of floating-point operations for the computation:

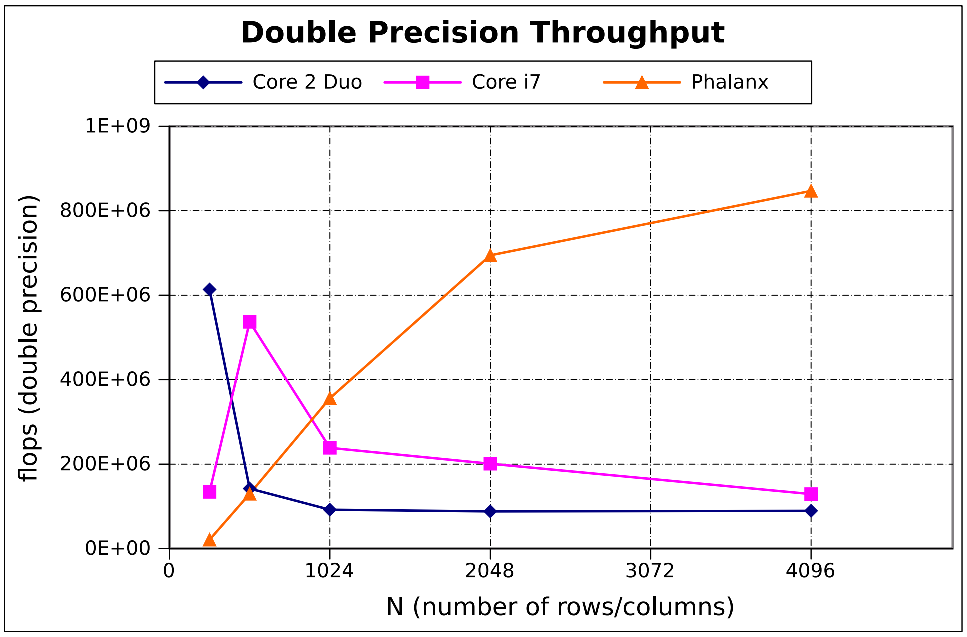
$$(2 \cdot N) \cdot N^2 = 2 \cdot N^3. \tag{6.5.1}$$

The $2 \cdot N$ term estimates the number of multiplication and addition operations per element in the product matrix. There are N multiplications and $N-1$ additions. For big N, the number of additions is very close to N. The remaining N^2 term denotes the number of elements in the product matrix, which is a square matrix with N rows and N columns.

Dividing the estimated number of floating-point operations by the execution time



(a) Achieved single precision compute throughput



(b) Achieved double precision compute throughput

Figure 6.5.1: Achieved compute throughput versus matrix size for single and double precision arithmetic.

yields an estimation of compute throughputs. The unit of the compute throughput is flops (floating point operations per second). Figure 6.5.1 shows the compute throughputs with respect to the size of the matrices for single precision and double precision computations.

An interesting observation is the drop in compute throughputs for both CPUs. A possible reason is the increased rate of cache miss as the matrix size grows exponentially. As the data no longer fit in the processor cache, the effective memory bandwidth reduces significantly. Adapting Equation 5.4.2 for the processor memory hierarchy:

$$B_{eff} = (1-\alpha)B_{RAM} + \alpha B_{cache}. \tag{6.5.2}$$

In the equation, B_{RAM} denotes the bandwidth of the system primary memory. As α tends to zero, the effective bandwidth B_{eff} is dominated by the slow B_{RAM}. As a result, the optimal intensity $I_{optimal}$ increases and the computation becomes I/O bounded.

Chapter 7

Case Study: NBody

This chapter presents a case study for a NBody simulation application. The same setup as in the case study in Chapter 6 is used.

7.1 Application Background

For physics applications, a NBody simulation predicts the motion of particles or celestial bodies due to the interacting forces between them [19]. By assuming all particles have zero volume, the need to account for collisions is removed. This simplifies the problem for the demonstration in this case study.

The particles are constantly attracted by gravitational forces due to their masses. The force f_{ij} acting on a particle i by another particle j is governed by the following equation [19]:

$$f_{ij} = G\frac{m_i m_j(v_j - v_i)}{||v_j - v_i||^3}. \tag{7.1.1}$$

In the equation, v_i and v_j denote the 3D position vectors of particle i and particle j, respectively. The symbols m_i and m_j denote the masses for particle i and particle j, respectively. The term $||v_j - v_i||$ denote the distance between the two particles. The constant G denotes the gravitational constant.

To compute the total force acting on a particle, all distinct pairs of forces f_{ik} are computed for all k except when k equals i. That is:

$$F_i = \sum f_{ik},\ for\, k \neq i. \tag{7.1.2}$$

If there are N particles, the total force acting on each particle is the sum of $N-1$ forces between all distinct pairs of particles. Therefore, the computation for all particles has N^2 complexity.

7.2 Implementation

The kernel source code of the NBody simulation that uses double precision is shown in Listing A.5. In this implementation, each CUDA thread computes the next position of a particle. The work is broken down so that each CUDA block loads a small number of particles into shared memory and uses the pre-loaded data for the computations. Next, it loads the next batch of particles and performs the computation. These steps repeat until all particles have been considered. From the total force acting on a particle, the new position is computed using verlet integration.

The implementation for serial execution is shown in Listing A.6. Comparing the serial version with the parallel version, there are little differences between the two. The parallel version is achieved simply by distributing the computation for each particle to a CUDA threads and adding a shared memory loading phase.

7.3 Benchmark

Figures 7.3.1, 7.3.2 and 7.3.3 show the execution time, speedup and throughput benchmarks of the NBody simulation. Only four measurements were obtained for the serial execution because the execution time becomes too long for large numbers of particles. As the number of particles increases, execution time grows exponentially. In Figure 7.3.2, the speedups compare the parallel execution using the Phalanx setup with the serial execution baselines using Core 2 Duo and Core-i7, respectively. In Figure 7.3.3, an operation represents a computation using Equation 7.1.1. The total number of operations is approximately N^2.

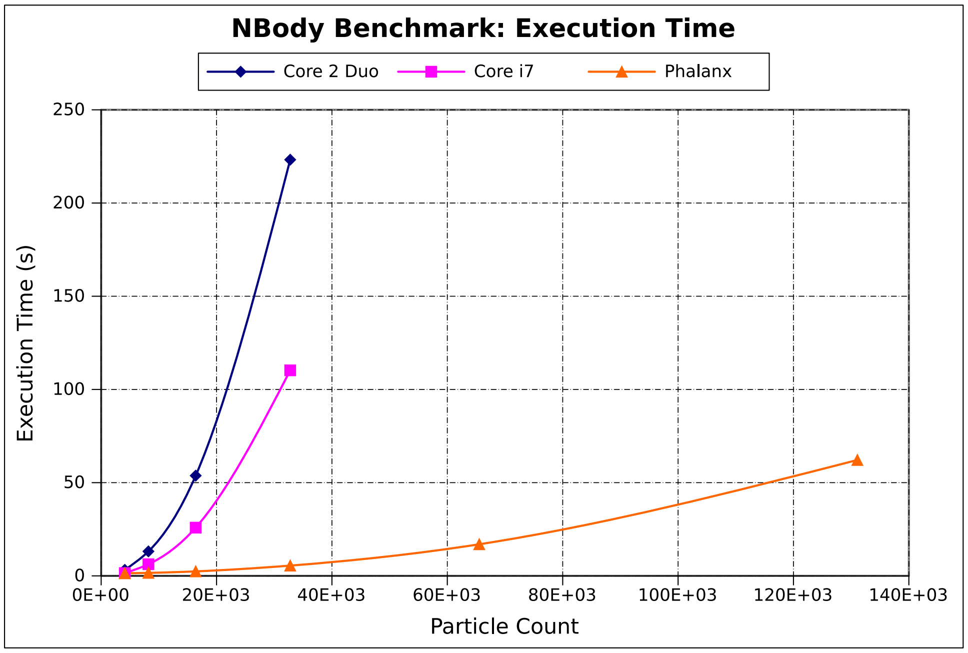


Figure 7.3.1: A benchmark of NBody simulation showing the execution time versus particle count.

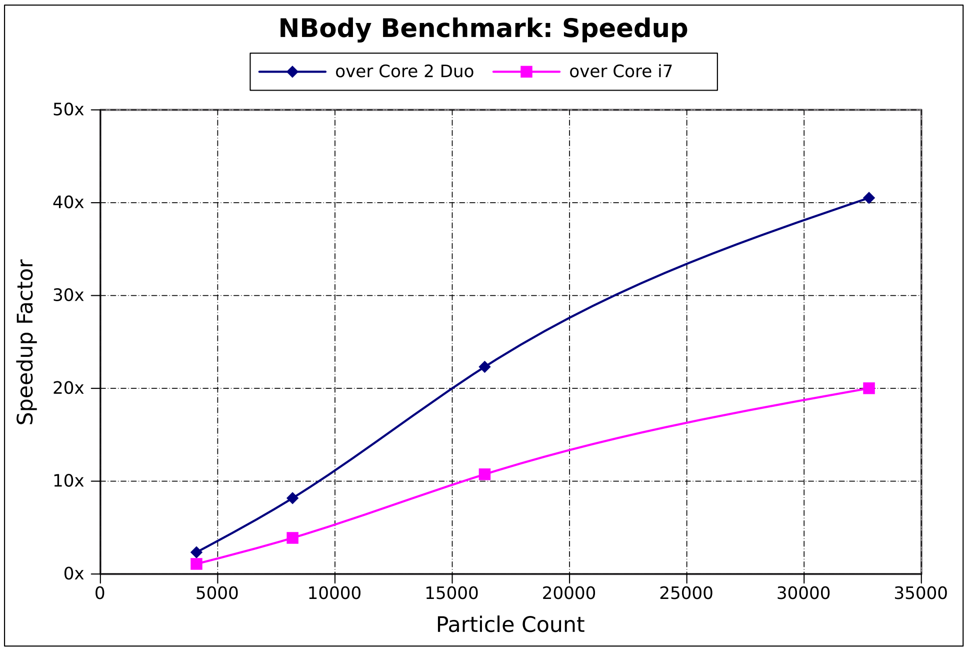


Figure 7.3.2: A benchmark of NBody simulation showing the speedup factor versus particle count. Speedups are normalized parallel execution time of the Phalanx setup with respect to serial execution baselines using the Core 2 Duo and Core i7 machines.

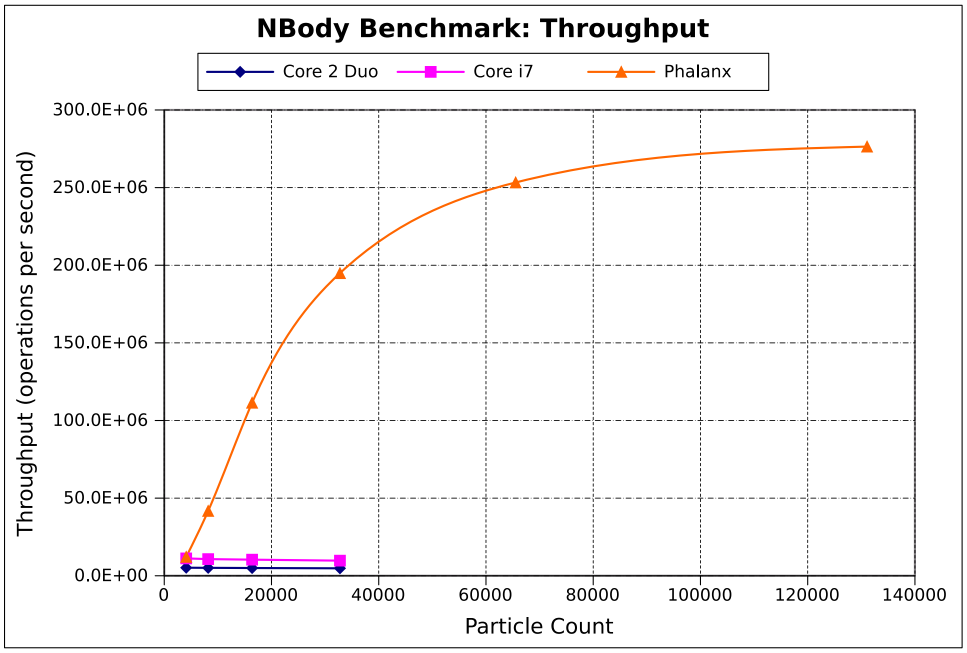


Figure 7.3.3: A benchmark of NBody simulation showing the throughput versus particle count.

7.4 Performance Analysis

Due to the higher intensity of NBody simulation, a higher performance gain is observed in the NBody simulation application than in the matrix multiplication application. The speedups reach 40$\times$ and 20$\times$ for Core 2 Duo and Core-i7 baselines, respectively, in Figure 7.3.2. Figure 7.3.3 shows that both CPUs have a constant throughput for the measured range. When particle count is 131×10^3, the throughput of the Phalanx setup approaches its saturation point. By projecting the trend of constant CPU throughput to this point, the maximum speedups are 57$\times$ and 28$\times$ for Core 2 Duo and Core i7 baselines, respectively.

The parallel version is *embarrassingly parallel*. Parallelism is achieved by merely separating the computation of each particle into different CUDA threads. Considering the amount of work to program this embarrassingly parallel version of the NBody simulation kernel, the amount of speedup is promising.

Chapter 8

Conclusion and Future Works

This thesis demonstrated that CUDA is well-suited for massive parallel computations in distributed systems. This thesis showed that the virtual architecture of CUDA can easily expand to the architecture of distributed systems. Streaming multiprocessors in a GPU becomes machine nodes in a distributed system. By scaling CUDA for distributed systems, programmers no longer need to use the less expressive the message-passing model when using MPI as the underlying messaging infrastructure. As a result, programmers can write CUDA applications that scale from a manycore GPU to a distributed system of hundreds of multicore processors. Together with cloud computing, Phalanx aims to simplify the development of distributed applications. Applications developed using Phalanx allow individuals or businesses to easily offload computation to compute instances in cloud services.

Large computation tasks cannot run efficiently on traditional CPUs. As seen in the matrix multiplication case study, large data set affect data locality drastically. On the other hand, Phalanx can adjust for the intensity of different types of computation by expanding the distributed system. For the NBody case study, Phalanx achieved 28$\times$ speedup over serial execution on Core-i7. With today's cloud computing, a distributed system can grow dynamically as computation becomes larger.

Network performance is the main concern for any distributed system. The principle of balance presented in Section 5.4.1 can serve as a general guide to configure distributed systems. The optimal setup depends on the computation. With cloud services, it is possible to allocate compute resources to fit the intensity of a computation. Although this thesis lacks explorations in large distributed systems and in cloud services due to limited resources, future works will include performance studies on large systems.

Future systems will combine different processor architectures for different types of computation. Funded by DARPA, NVIDIA's Echelon project aims at providing a heterogeneous architecture that delivers 16-Tflops at high energy efficiency [15]. Although Phalanx does not support CPU-GPU heterogeneity, future works will implement this feature so that programmers can rely on a single programming framework for heterogeneous parallel computing. Current version of Phalanx have been lightly tested for x86-ARM heterogeneity. Code generation for other architectures, such as Sparc and PowerPC, is also possible. In fact, Phalanx should work on any architecture that LLVM supports. However, MPI implementation may not support every processor architecture. For instance, OpenMPI has only started its ARM support, which is not available in the stable release.

Finally, this thesis explored a new approach to distributed heterogeneous programming. The exploration is still incomplete with various limitations and pending features. In the future, the development of Phalanx will continue. Hopefully, this new approach would be integrated to the standard developement tools for programming distributed heterogeneous systems.

Bibliography

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2010, ch. 1.

[2] ——, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2010, ch. 4.

[3] F. Gebali, *Algorithms and Parallel Computing*. John Wiley & Sons, Apr. 2011.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2010, ch. 7.

[5] W. Thies, “Language and compiler support for stream programs,” Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Feb. 2009. [Online]. Available: http://groups.csail.mit.edu/commit/papers/09/thies-phd-thesis.pdf

[6] *MPI: A Message-Passing Interface Standard Version 2.2*, Message Passing Interface Forum, Sept. 2009. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, Budapest, Hungary, Sept. 2004, pp. 97–104.

[8] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, “Open MPI: A high-performance, heterogeneous MPI,” in *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, Sept. 2006.

[9] R. Kumar, T. G. Mattson, G. Pokam, and R. F. V. der Wijngaart, “The case for message passing on many-core chips,” in *Multiprocessor System-on-Chip*, 2011, pp. 115–123.

[10] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, “Languages and compilers for parallel computing,” J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89740-8_2

[11] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, “Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT ’10. New York, NY, USA:

ACM, 2010, pp. 353–364. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854318

[12] *PTX: Parallel Thread Execution ISA Version 2.2*, NVIDIA, Oct. 2010.

[13] *NVIDIA CUDA C Programming Guide Version 4.1*, NVIDIA, Nov. 2011. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[14] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002, *See* `http://llvm.cs.uiuc.edu`.

[15] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, Sept.-Oct. 2011.

[16] R. Vuduc and K. Czechowski, "What gpu computing means for high-end systems," *Micro, IEEE*, vol. 31, no. 4, pp. 74–78, Jul.-Aug. 2011.

[17] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc, "Balance principles for algorithm-architecture co-design," in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, ser. HotPar'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001252.2001261

[18] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A hands-on Approach.* Morgan Kaufmann, 2010, ch. 6, pp. 103–111.

[19] H. Nguyen, *GPU Gems 3.* Addison-Wesley Professional, 2007, ch. 31. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html

Appendix A

Source Listing for Case Studies

A.1 Matrix Multiplication Case Study

Listing A.1: Phalanx driver for matrix multiplication kernel using shared memory.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <phalanx_host.hpp>
#include <phalanx_util.hpp>
#include "kernelconfig.hpp"
float MA[COUNT];
float MB[COUNT];
float MC[COUNT];
float GOLD[COUNT];
float *pA = MA;
float *pB = MB;
float *pC = MC;
int32_t pW = MATRIX_SIZE;
void* parameters[]={ &pA, &pB, &pC, &pW };

int main(int argc, char** argv){
    using phalanx::communicator;
    communicator.init(argc, argv);
    const int NUM_OF_PROC = 16;
    communicator.set_hosts("169.254.8.58,169.254.8.95");
    communicator.set_wdir("/home/michael/bin/mpiapp");
    using namespace std;
    srand(time(0));

    cout << BLOCK_COUNT*BLOCK_SIZE << endl;

    // init
    for(unsigned int i=0; i<COUNT; ++i){
        MA[i]=i;
        MB[i]=i;
    }

    // invoke
    dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridDim(BLOCK_COUNT, BLOCK_COUNT);
    phalanx::Timer timer;
    timer.start();
    phalanx::launchKernel("worker", gridDim, blockDim,
        parameters, sizeof(parameters)/sizeof(void*),
        NUM_OF_PROC);
    timer.end();
    std::cerr << "Phalanx time: " << timer.duration() << '\n';

    if(false){
        timer.start();
        // golden values
        unsigned int N = MATRIX_SIZE;
        for(unsigned int y=0; y<N; ++y){
            for(unsigned int x=0; x<N; ++x){
                GOLD[x+y*N] = 0;
                for(unsigned int k=0; k<N; ++k ){
                    GOLD[x+y*N] += MA[k+y*N] * MB[x+k*N];
                }
            }
        }
        timer.end();
        std::cerr << "CPU time: " << timer.duration() << '\n';
        //error checking
        for(unsigned int y=0; y<N; ++y){
            for(unsigned int x=0; x<N; ++x){
                float expect = GOLD[x+y*N];
                float error = fabs(expect-MC[x+y*N])/expect;
                if(error>1e-8){
                    cerr << "Error at x, y: " << x << ", " <<
                        y << '\n';
                    cerr << error << '\n';
                    cerr << MC[x+y*N] << '\n';
                    return 1;
                }
            }
        }
    }
    cout << "\nAll is well!\n";
    communicator.finalize();
    return 0;
}
```

Listing A.2: Matrix multiplication kernel using shared memory.

```
enum {
    BLOCK_SIZE = 32,
    MATRIX_SIZE = BLOCK_SIZE*BLOCK_COUNT,
};
__global__
void matrixMultipliyKernel(float A[], float B[], float P[],
   int N){

    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;
    const int by = blockIdx.y;

    const int i = tx + bx * blockDim.x;
    const int j = ty + by * blockDim.y;

    __shared__ float Acache[MATRIX_SIZE*BLOCK_SIZE];
    __shared__ float Bcache[BLOCK_SIZE*MATRIX_SIZE];

    for(unsigned int t=0; t<MATRIX_SIZE/BLOCK_SIZE; ++t){
        unsigned int offset = t*BLOCK_SIZE;
        Acache[(offset + tx) + ty*MATRIX_SIZE] = A[(offset +
           tx) + j*MATRIX_SIZE];
        Bcache[tx + (ty+offset)*BLOCK_SIZE] = B[i + (ty+offset
           )*MATRIX_SIZE];
    }
    __syncthreads();

    if( i>=N || j>=N ) return;

    float result = 0;
    for( int k=0; k<N; ++k ){
        result += Acache[k+ty*MATRIX_SIZE] * Bcache[tx+k*
           BLOCK_SIZE];

    }

    P[i+j*N] = result;

}
```

Listing A.3: Serial matrix multiplication.

```
void matrixMultipliySerial(float A[], float B[], float C[],
   int N){
    for(unsigned int y=0; y<N; ++y){
        for(unsigned int x=0; x<N; ++x){
            C[x+y*N] = 0;
            for(unsigned int k=0; k<N; ++k ){
                C[x+y*N] += A[k+y*N] * B[x+k*N];
            }
        }
    }
}
```

A.2 NBody Case Study

Listing A.4: Phalanx driver for NBody simulation kernel using shared memory.

```
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <cassert>

#include <phalanx_host.hpp>
#include <phalanx_util.hpp>
#include "nbody.h"

static const bool VERIFY = false;
static const bool BENCHMARK = true;
static double phalanx_time_average = 0;
static double host_time_average = 0;
enum {
    BLOCK_SIZE = 1024,
    PARTICLE_COUNT = BLOCK_SIZE * 128,
    NUMBER_OF_PROCESSOR = 16,
    NUMBER_OF_FRAME = 2,
};

void device_nbody_simulation(Vector3f NewPos[], Vector3f
   CurPos[],
        Vector3f OldPos[], double Mass[], unsigned int count,
           double dt) {
    using namespace phalanx;
    dim3 blockDim(BLOCK_SIZE);
    dim3 gridDim(count / BLOCK_SIZE);

    void * parameters[]={
            &NewPos, &CurPos, &OldPos, &Mass, &count, &dt
    };

    std::cerr << "Particle␣Count␣=␣" << count << '\n';
    // CUDA code starts here
    phalanx::Timer timer;
    timer.start();

    phalanx::launchKernel("worker", gridDim, blockDim,
                          parameters, sizeof(parameters)/
                             sizeof(void*),
                          NUMBER_OF_PROCESSOR);

    timer.end();
    if (BENCHMARK)
        std::cerr << "phalanx:␣" << timer.duration() << '\n';
    phalanx_time_average += timer.duration();
}

double      mass[PARTICLE_COUNT];
Vector3f  P1[PARTICLE_COUNT];
Vector3f  P2[PARTICLE_COUNT];
Vector3f  P3[PARTICLE_COUNT];
Vector3f  Gold[PARTICLE_COUNT];

int main(int argc, char ** argv) {
    using phalanx::communicator;
    communicator.init(argc, argv);
    communicator.set_hosts("169.254.8.58,169.254.8.95");
    communicator.set_wdir("/home/michael/bin/mpiapp");
    srand(time(NULL));

    Vector3f * newpos = P1;
    Vector3f * oldpos = P2;
    Vector3f * curpos = P3;

    for (unsigned int i = 0; i < PARTICLE_COUNT; ++i) {
        const unsigned int PositionFactor = 80;

        oldpos[i].x = curpos[i].x = (double) (rand() %
           PositionFactor)
              - (double) PositionFactor / 2;
        oldpos[i].y = curpos[i].y = (double) (rand() %
           PositionFactor)
              - (double) PositionFactor / 2;
        oldpos[i].z = curpos[i].z = (double) (rand() %
           PositionFactor)
              - (double) PositionFactor / 2;

        mass[i] = (rand() % 20) / 10.0f;

        Vector3f color = { 1.0f / PARTICLE_COUNT * i, 1.0f, 1.0f };
    }
    for (unsigned int T = 0; T < NUMBER_OF_FRAME; ++T) {
        fprintf(stderr, "Frame␣%d\n", T);

        const double dt = 10.0f / 30;

        if (VERIFY)
            host_nbody_simulation(Gold, curpos, oldpos, mass, PARTICLE_COUNT,
                    dt);
        device_nbody_simulation(newpos, curpos, oldpos, mass, PARTICLE_COUNT,
                dt);

        for (unsigned int i = 0; i < PARTICLE_COUNT; ++i) {
            if (VERIFY) {
                std::cerr << "checking␣i␣=␣" << i << '\n';
                check(Gold[i].x, newpos[i].x);
                check(Gold[i].y, newpos[i].y);
                check(Gold[i].z, newpos[i].z);
            }

            if (newpos[i].x != newpos[i].x)
                newpos[i].x = 0;
            if (newpos[i].y != newpos[i].y)
                newpos[i].y = 0;
            if (newpos[i].z != newpos[i].z)
                newpos[i].z = 0;
        }

        //rotate pointers
        Vector3f * temp = oldpos;
        oldpos = curpos;
        curpos = newpos;
        newpos = temp;
    }
    std::cerr << "Average␣Time␣per␣Frame\n";
    std::cerr << "\tPhalanx␣=␣" << phalanx_time_average / NUMBER_OF_FRAME << "\n";
    std::cerr << "\t␣␣␣Host␣=␣" << host_time_average / NUMBER_OF_FRAME << "\n";
```

```

    communicator.finalize();
    return 0;
}
```

Listing A.5: NBody simulation kernel using shared memory.

```
#define BIG_G (80.0f * 6.67e-4) // modified gravitation
   constant

typedef struct {
    double x, y, z;
} Vector3f;

enum {SHARED_CHUNK=1024};

__global__
void nbodySimulationKernel(Vector3f NewPos[], const Vector3f
   CurPos[], const Vector3f OldPos[], const double Mass[],
   const unsigned int count, const double dt){
    const unsigned int pidx = threadIdx.x + blockIdx.x *
       blockDim.x;
    Vector3f force={0, 0, 0};
    __shared__ Vector3f shCurPos[SHARED_CHUNK];
    __shared__ double     shMass[SHARED_CHUNK];
    const Vector3f pos = CurPos[pidx];
    const Vector3f oldpos = OldPos[pidx];
    const double mass = Mass[pidx];
    Vector3f newpos = NewPos[pidx];
    // N body physics
    for( unsigned int h=0; h<count/SHARED_CHUNK; ++h ){
        // Preload
        __syncthreads();
        shCurPos[threadIdx.x] = CurPos[h*SHARED_CHUNK+
           threadIdx.x];
        shMass[threadIdx.x] = Mass[h*SHARED_CHUNK+threadIdx.x
           ];
        __syncthreads();
        // N body physics
        for( unsigned int k=0; k<SHARED_CHUNK; ++k){
            if( (k+SHARED_CHUNK*h)!=pidx ){
                const Vector3f other_pos = shCurPos[k];
                const double other_mass = shMass[k];

                double mass2 = other_mass;
                Vector3f diffpos = {
                    other_pos.x - pos.x,
                    other_pos.y - pos.y,
                    other_pos.z - pos.z,
                };
                double dist_square = diffpos.x*diffpos.x +
                   diffpos.y*diffpos.y + diffpos.z*diffpos.z;
                double dist_inv_cube = rsqrt(dist_square*
                   dist_square*dist_square);
                force.x += BIG_G * mass2 * diffpos.x *
                   dist_inv_cube;
                force.y += BIG_G * mass2 * diffpos.y *
                   dist_inv_cube;
                force.z += BIG_G * mass2 * diffpos.z *
                   dist_inv_cube;
            }
        }
    }
    // Verlet integration
    Vector3f accel = {force.x, force.y, force.z};
    newpos.x = 2*pos.x - oldpos.x + accel.x*dt*dt;
    newpos.y = 2*pos.y - oldpos.y + accel.y*dt*dt;
    newpos.z = 2*pos.z - oldpos.z + accel.z*dt*dt;
    NewPos[pidx] = newpos;
}
```

Listing A.6: Serial NBody simulation function.

```
void nbodySimulationSerial(Vector3f NewPos[], const Vector3f
   CurPos[], const Vector3f OldPos[], const double Mass[],
   const unsigned int count, const double dt) {
    for (unsigned int pidx = 0; pidx < count; ++pidx) { // for
        every particle
        Vector3f force = { 0, 0, 0 };
        const Vector3f & pos = CurPos[pidx];
        const Vector3f & oldpos = OldPos[pidx];
        const double & mass = Mass[pidx];
        Vector3f & newpos = NewPos[pidx];
        // N body physics
        for (unsigned int k = 0; k < count; ++k) {
            if (k != pidx) {
                const Vector3f & other_pos = CurPos[k];
                const double & other_mass = Mass[k];
                double mass2 = mass * other_mass;
                Vector3f diffpos = { other_pos.x - pos.x,
                    other_pos.y - pos.y,
                        other_pos.z - pos.z, };
                double dist_sqrt = diffpos.x * diffpos.x +
                    diffpos.y * diffpos.y
                        + diffpos.z * diffpos.z;
                double dist_cube = sqrt(dist_sqrt*dist_sqrt*
                    dist_sqrt);
                force.x += BIG_G * mass2 * diffpos.x /
                    dist_cube;
                force.y += BIG_G * mass2 * diffpos.y /
                    dist_cube;
                force.z += BIG_G * mass2 * diffpos.z /
                    dist_cube;
            }
        }
        Vector3f accel = { force.x / mass, force.y / mass,
            force.z / mass };
        // verlet integration
        newpos.x = 2 * pos.x - oldpos.x + accel.x * dt * dt;
        newpos.y = 2 * pos.y - oldpos.y + accel.y * dt * dt;
        newpos.z = 2 * pos.z - oldpos.z + accel.z * dt * dt;
    }
}
```