EXPLORING NOVEL MANY-CORE ARCHITECTURES FOR SCIENTIFIC COMPUTING

by

Long Chen

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical & Computer Engineering

Fall 2010

EXPLORING NOVEL MANY-CORE ARCHITECTURES FOR SCIENTIFIC COMPUTING

by

Long Chen

Approved: _____

Kenneth Barner, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____

Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _____

Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Hui Fang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Andres Márquez, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to Prof. Guang R. Gao for his inspiration, excellent guidance, support and encouragement. His attitude, vision, and insights toward research problems have been the most inspiration and made this research work a rewarding experience. I owe an immense debt of gratitude to him for not only giving me the invaluable guidance and support about this research work, but also taking care of my life. His rigorous scientific approach and endless enthusiasm have influenced me greatly. Without his kind help, this thesis and many other works would have been impossible.

I would also like to thank Prof. Xiaoming Li, Prof. Hui Fang, Prof. John Cavazos, and Dr. Andres Márquez for serving on my dissertation committee. Their comments and suggestions were very helpful in improving the quality of this work.

I want to sincerely acknowledge all the help from many members in CAPSL, University of Delaware. In particular, I would like to thank Dr. Ziang Hu, Dr. Haiping Wu, Dr. Weirong Zhu, Dr. Juan del Cuvillo, Andrew Russo, Geoffery Gerfin, Junmin Lin, Dr. Guangming Tan, Dr. Fei Chen, and Joseph Manzano for providing insightful comments and helpful advice on my research. Their kind assistance and friendship have also made my life colorful.

My internship at Pacific Northwest National Laboratory is a valuable asset for gaining experience and understanding in parallel programming. I appreciate Dr. Villa Oreste, Dr. Sriram Krishnamoorthy, and Dr. Daniel G. Chavarría-Miranda for for their friendly help and support, without which my dissertation could not be completed.

Special thanks to Michael Merrill for his initial FFT implementation. Thanks also go to the faculties in the Department of Electrical & Computer Engineering, University of Delaware, for their constant encouragement and valuable advice. Acknowledgment is extended to the University of Delaware for providing me the research facilities and challenging environment during my researching and studying time.

My parents will always be an inspiration to me. My wife, Yuan, has always been there for me. I would thank them for their support, understanding, patience and love during this process of my pursuit of knowledge. This dissertation, thereupon, is dedicated to them for their unconditional love.

For my wife, *Yuan Zhang*, who offered me unconditional love and support throughout the course of this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The rapid revolution in microprocessor chip architecture due to the many-core technology is presenting unprecedented challenges to the application developers as well as system software designers: how to best exploit the computation potential provided by such many-core architectures?

The scope of this dissertation is to study programming issues for many-core architectures, and the contributions of this dissertation are in two main areas.

Optimizing the Fast Fourier Transform for IBM Cyclops-64

To understand issues in designing and developing high-performance algorithms for many-core architectures, we use the fast Fourier transform (FFT) as a case study to investigate the above issues on the IBM Cyclops-64 many-core chip architecture. We analyze the optimization challenges and opportunities for FFT problems, and identify domain-specific features of the target problems and match them well with some key many-core architecture features. We quantitatively address the impacts of various optimization techniques and effectiveness of the target architecture. The resulting FFT implementations achieve excellent performance results in terms of both speedup and absolute performance. To assist the algorithm design and performance analysis, we present a model that estimates the performance of parallel FFT algorithms for an abstract many-core architecture. This abstract architecture captures generic features and parameters of several real many-core architectures; therefore the performance model is applicable for any architecture with similar features. We derive the performance model based on cost functions for three main components of an execution: the memory accesses, the computation, and the synchronization. The

experimental results demonstrate that our model can predict the performance trend accurately, and therefore can provides valuable insights for designing and tuning FFT algorithms on many-core architectures.

Exploring Fine-grained Task-based Execution on Graphics Processing Unit-enabled Systems

Using many-core Graphics Processing Unit (GPU) is gaining popularity in scientific computing. However, the conventional data parallel GPU programming paradigms, e.g., NVIDIA CUDA, cannot satisfactorily address certain issues, such as load balancing, GPU resource utilization, overlapping fine-grained computation with communication, etc. The problem is exacerbated when trying to effectively exploit multiple GPUs concurrently, which are commonly available in many modern systems. Our solution to this problem is a fine-grained task-based execution framework for GPU-enabled systems. Our framework allows concurrent execution of fine-grained tasks on GPU-enabled systems. The granularity of task execution is finer than what is currently supported in CUDA; the execution of a task only requires a subset of the GPU hardware resources. Our framework provides means for solving the above issues and efficiently utilizing the computation power provided by the GPUs. We evaluate our approach using both micro-benchmarks and a molecular dynamics (MD) application that exhibits significant load imbalance. Experimental results with a single-GPU configuration show that our fine-grained task-based solution can utilize the hardware more efficiently than the CUDA scheduler for unbalanced workload. On multi-GPU systems, our solution achieves near-linear speedup, good dynamic load balance, and significant performance improvement over other techniques based on standard CUDA APIs.

Chapter 1

INTRODUCTION

Traditionally, scientists employ both experimental and theoretical approaches to solve problems in the fields of science and engineering, such as astronomy, biology, chemistry, physics, etc. With the advent of computer machinery, scientists have been able to transform a theory or a realistic phenomena into an algorithm, analyze and understand the problem through computing and simulations. As a matter of fact, the use of computation and simulation has now become so prevalent and essential a part of the scientific process that many people believe that the scientific paradigm has been extended to include simulation as an additional dimension [90]. On the other hand, as the workhorse, computing systems have been playing a critical role for scientific computing, and hardware advances have allowed scientists to investigate problems in greater detail and more complex systems than previous generations of hardware did.

Currently, the computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. As the traditional single-core processor architectures are no longer able to take advantage of the integrated circuit (IC) technology advances due to some fundamental issues, i.e., power consumption, heat dissipation, memory wall, etc., computer architects look for other ways to utilize the transistor budget. By integrating a number of simple processors/cores on a single die, it is believed that this many-core chip technology has higher power-efficiency, improved heat dissipation, better memory latency tolerance, and many other benefits. Projections and

early prototypes indicate that in the very near future dozens, if not hundreds, of general-purpose and/or special-purpose cores will be included on a single chip. Many researchers think that many-core architectures are going to become the mainstream for the parallel computing in the future. However, unlike previous hardware evolutions, this shift in the hardware roadmap will have a profound impact on scientific computing community by posing unprecedented challenges in the management of parallelism, locality, scalability, load balance, energy, fault-tolerance, etc. It is an open question whether the existing parallel programming approaches will continue to scale to future computing systems built with many-core chips.

1.1 Scientific Computing

Scientific computing is the field of study "concerned with constructing mathematical models and quantitative analysis techniques and using computers to analyze and solve problems in science and engineering domains" [154], such as climate prediction, materials science, structural biology, superconductivity, semiconductor design, drug design, human genome, quantum chromodynamics, turbulence, speech and vision, relativistic astrophysics, vehicle dynamics, nuclear fusion, combustion systems, oil and gas recovery, ocean science, vehicle signature, undersea surveillance [90], etc. Algorithms and mathematical methods used in scientific computing varies from one domain to another domain. Some commonly applied algorithmic methods include numerical analysis, molecular dynamics, numerical linear algebra, discrete Fourier transform, Gaussian elimination, Cholesky factorizations, Newton's method, Monte Carlo method, etc.

Scientific computing is typically of the form of various computer simulations or computations in different scientific disciplines. Such simulation/computation normally requires massive amounts of calculations that are impossible or very costly

to observe through empirical means. For example, the National Oceanic and Atmospheric Administration of United States Department of Commerce uses supercomputers for weather and climate prediction. The primary system processes billions of bytes of weather observation data everyday, and it can perform 69.7 trillion calculations per second [110], which would take a person millions of years to process with a calculator.

On the other hand, advances in scientific modeling activities lead to an ever increasing performance demand. For instance, molecular dynamics (MD) [55] simulations are used in the fields of biology, chemistry, and medicine to model the motions of molecular systems, including proteins, cell membranes, and deoxyribonucleic acid (DNA), at an atomic level of detail. Early MD simulations were carried out for small systems; a MD simulation in 1977 was conducted with a size of only 500 atoms and a simulation time of 9.2×10^{-12} second [96]. Because long, accurate MD simulations could in principle provide answers to some of the most important outstanding questions, the state-of-the-art MD simulations usually work on systems of very large size and very long simulation times. For example, in 2006, a simulation of the complete satellite tobacco mosaic virus was conducted with a size of $1,000,000$ atoms, and a simulation time of 5.0×10^{-8} second [54]. Advances in hardware capability enable investigating larger systems and more application functionalities, which grow in significance and place even greater demands on performance. For example, many of the most important biological processes occur over timescales on the order of a millisecond [129]. However, due to the limitation of the current technology, the vast majority of MD simulations have been limited to the nanosecond timescale.

1.2 Architecture Shift: from Single-core to Multi-core/Many-core

In 1965, Gordon Moore predicted that the transistor density of semiconductor chips would double approximately every 18 to 24 months, which is known as Moore's law [98]. It predicted computers would not only have more transistors

but also faster transistors. Many people misinterpret Moore's law as a predictor of central processor unit (CPU) clock frequency, the most commonly used metric in measuring computing performance. Indeed, the processor frequency of the traditional single-core processor has followed Moore's law for 40 years. This made it relatively easy to improve the performance of the traditional software, including the scientific computer applications. Most users just simply relied on the increasing capabilities and speed of single-core processors to get free performance improvement. However, this frequency increase could no longer be sustained due to the following problems.

- The single most important problem is the increasing power density, which is an unsolvable problem for conventional single-core processor designs. The number of transistors per chip has greatly increased in recent years, each of these transistors consumes power and produces heat. If this rate continued, processors would soon be producing more heat per square centimeter than the surface of the sun [59]. This is so-called *Power Wall*. Because of this, several of the next generation processors, such as the Tejas Pentium 4 processor from Intel, were canceled or redefined due to power consumption issues [156].

- Memory speeds are not increasing as quickly as processor speeds. These diverging rates imply an impending *Memory Wall*, in which memory accesses dominate code performance. These wasted clock cycles can nullify the benefits of frequency increases in the processor.

- Advances in IC technology allow the hardware feature size to continue dropping. As feature size drops, interconnect delay often exceeds gate delay and becomes the most serious performance problem to be solved in future IC design, and it can eventually cancel the speed increases of the transistors.

- Most single-core processors are designed to exploit the instruction level parallelism (ILP) in programs. ILP approaches could overlap the execution of instructions and improve performance without affecting the standard single-core processor programming model. While exploiting ILP was the primary focus of processor designs for a long time, however, ILP can be quite limited or hard to exploit in many applications [72], which is known as *ILP Wall*. Meanwhile, the higher level parallelisms, i.e., thread-level parallelism (TLP) and data-level parallelism (DLP), occurring naturally in a large number of applications cannot be exploited with the ILP approach.

Because of limits described above, the era of taking advantage of Moore's law on the traditional single-core processor designs appeared to be coming to an end. Since 2005, the computing industry changed course when all major processor manufacturers, such as Intel, IBM, Sun, AMD, turned to multi-core designs, where a number of simple cores are integrated on a single die [8, 21, 34, 39, 78, 80, 88, 95]. The multi-core architectures are believed to be able to extend the benefits of Moore's law by doubling the number of standard cores per die with every semiconductor process generation starting with a single-core processor.

There are many advantages to building multi-core processors out of smaller and simpler cores:

- Since the power consumption of a single core drops significantly with the reduction in frequency, multi-core architectures can provide a power-efficient way to achieve performance by running multiple cores with moderate clock rate [30, 134].

- A small core is an economical element that is easy to shut down and reconfigure, which allows a finer-grained ability to control the overall power and performance.

- Multi-core architectures partition resources, including memory, into individual small parts, and thus alleviate the effect of the interconnect delay and reduce contention on a shared global memory. The availability of local storage on each core serves to reduce contention on a globally shared main memory, and if data is distributed adequately among the cores, there is an effective increase in overall concurrency.

- Multi-core chip architectures naturally support thread-level parallelism, which is expected to be exploited in future applications and multiprocessor-aware operating systems and environments [70].

- A small and simple processing element of a processor is easy to design and functionally verify. In particular, it is more amenable to formal verification techniques than complex architectures with out-of-order execution.

- Performance and power characteristics of smaller hardware modules are easier to predict within existing electronic automation-design flows [138].

While the multi-core processors, like dual-core, quad-core, even hexa-core microprocessors, have been released into the market of servers and personal computers, both the industry and academia are actively exploiting the design space of the many-core architectures by integrating an even larger number of cores (tens or hundreds) into a single chip. Examples include Intel teraflops research chip [149] and Single-chip Cloud Computer [81], IBM Cyclops-64 chip architecture [43], NVIDIA CUDA [108], ATI Stream [6], ClearSpeed CSX700 [34], Tilera [145], Berkeley RAMP Gold [141], MIT RAW [142] and ATAC [89], Stanford SmartMemories [127], etc. Researchers also predicted that 1000-core chip would be achieved when 30nm technology is available [9].

1.3 Challenges for Programming Many-core

While these many-core architectures provide high theoretic performance, such increase of performance cannot be harnessed as simply as what we did with single-core processors. The majority of software community was very used to the idea of gaining increased performance by upgrading machines with a faster processor. Unfortunately this kind of automatic improvement will not be possible when one upgrades to a many-core computer. Although a many-core chip can run multiple programs simultaneously, it does not complete a given program in less time, or finish a larger program in a given amount of time, without extraordinary effort. The problem is that most programs are written in sequential programming languages, and these programs must be modified and optimized to exploit possible performance gains enabled by certain hardware features.

For the first time, many-core architectures demand that the mainstream software community engages in parallel processing, which until now was reserved for the rarefied field of supercomputing. On the other hand, this shift in the hardware roadmap poses unprecedented challenges to the software development community. For example, the programmer will be faced with the scalability challenge of expressing, coordinating and exploiting multi-level parallelism provided by the many-core systems. The programmer will also be faced with the locality challenge of optimizing data movement in a highly non-uniform (explicitly managed) memory hierarchy with order-of-magnitude gaps that separate data accesses to core-local memory, on-chip global memory, and intra-node off-chip memory, and communications with remote nodes. While to exploit architectural features and eventually obtain the desired performance is the ultimate goal for programmers of this many-core era, no consensus has been reached on how to do so.

On the other hand, people have been doing parallel programming for many years on vector machines, clusters, SMPs, etc. Different approaches have been

proposed and utilized. Here we present a brief review of these parallel programming approaches, and point out their ineffectiveness in this many-core era.

1.3.1 Explicit Threading

Explicit threading approach includes POSIX threads (Pthreads) [4], Sun Solaris threads, Windows threads, and other native threading application programming interfaces (APIs). It is primarily designed to express the natural concurrency that is present in most applications, and to improve the performance and responsibility. This approach usually offers a comprehensive set of routines to provide a fine-grain control over threading operations, such as create, manage, synchronize threads, etc. Programmers control the application by explicitly call these routines. In the situations where threads have to be individually managed, this approach would be the more natural choice. By devoting sufficient time and effort, programmers may be able to parallelize the problem and achieve good performance.

However, because explicit threading is an inherently low-level API that mostly requires multiple steps to perform simple threading tasks, it demands considerable effort from the programmer's side. Also, this approach does not offer encapsulation or modularity. Therefore, manually managing hundreds or thousands threads definitely would be a nightmare for the vast majority of programmers. Due to this reason, developers have been increasingly looking for other simpler alternatives.

1.3.2 OpenMP

Jointly defined by a group of major computer hardware and software vendors, OpenMP [2] is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. It gives programmers a simple and effective interface for developing parallel applications on shared-memory systems.

OpenMP uses a *fork-join* model of parallel execution, which is illustrated in Figure 1.1. In an OpenMP program, the programmer has to explicitly specify the start and end of each *parallel region* with OpenMP APIs. All OpenMP programs begin with a single *master thread*. At first this master thread executes sequentially. When the master thread encounters a parallel region construct, it creates (fork) a set of parallel working threads. Then statements in that parallel region construct are then executed by all threads in parallel. When all threads finish the corresponding execution in that parallel region construct, they synchronize and terminate (join), except the master thread continuing.

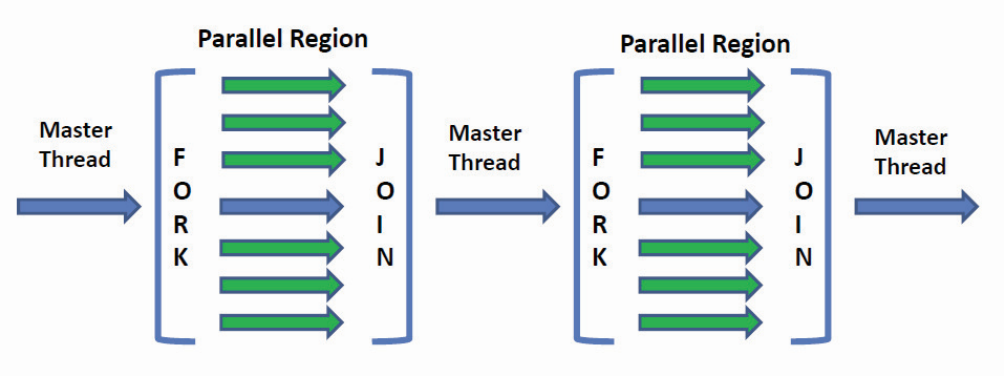


Figure 1.1: OpenMP Fork-Join Model

For applications that are characterized by easy data decomposition and simple thread orchestration, OpenMP is usually a simple and sufficient way for parallelizing programs. Because of this, OpenMP has become a very successful model for developing shared-memory parallel applications, especially for conventional symmetric multiprocessors (SMPs) [15, 82, 124].

If we simply treat many-core chips as SMPs, OpenMP could become a promising choice. However, it is important to recognize that OpenMP does have several weaknesses for programming many-core chips.

- OpenMP is not meant for non-shared memory systems by itself. This excludes the possibility of directly utilize OpenMP on a large number of parallel systems. Many scientific computing systems consist of a large number of many-core chips, and thus have a hierarchical architecture of both shared memory space (within one chip) and distributed resources (across chips). OpenMP alone will not be sufficient to handle this situation. On the surface, a hybrid-programming model (by mixing shared memory inside the node and message passing between the nodes) seems to be intuitive. However, most attempts do not achieve the performance of the equivalent message passing codes (or a share memory codes) [25, 26, 74, 128]. Also, it is not clear how to smoothly make the interaction between different programming models [32, 131].

- OpenMP does not guarantee to make the most efficient use of hardware resources. This may introduce some crucial issues for utilizing many-core architectures. In particular, many-core architectures usually offer unique capabilities that are fundamentally different from SMPs, which present significant new opportunities. For example, the peak bandwidth between two processors is 780 MB/s for a SGI SMP machine, Origin2000 [38], while it is 4 GB/s for the IBM Cyclops-64 many-core chip architecture[44]. Such huge on-chip inter-core communication bandwidth on a many-core chip is many times greater than is typical for an SMP, to the point where it is less likely to be a performance bottleneck. Inter-core latencies also are far less than are typical for an SMP system. If we simply treat many-core chips as traditional SMPs, then we may miss some very important opportunities for new architectures and algorithm designs that can exploit these new features.

- OpenMP doest not guarantee code correctness. It is the programmer's responsibility to make sure the resulting code is correct, free of threading problems, like data dependencies, race conditions, deadlocks, etc.

- OpenMP has its major strength on loop optimizations; loops can be parallelized easily with OpenMP. However, for other forms of parallel structures, for example, event-driven tasks, the possibility that the opportunity for parallel loop threads is limited.

1.3.3 Message Passing

Message passing paradigm is the main alternative to shared data processing. In a message passing program, processes do not communicate implicitly through shared data; they send and receive explicit data messages. The message processing model consists of a set of processes that have only local memory but are able to communicate by message passing primitives.

The message passing model has gained wide use in the field of parallel computing due to several advantages.

- The message passing model naturally fits on parallel supercomputers and clusters of workstations, where separate processors are connected by a communications network. But this does not necessarily limit message passing to the domain of distributed memory architectures: it can run on shared memory systems as well.

- Message passing offers a rich set of functions for constructing parallel algorithms.

- By giving programmer explicit control of data locality, message passing usually can achieve an effective use of the memory hierarchy of modern architectures, thus achieve a satisfied performance.

The message passing programming model has been effectively standardized by Message Passing Interface (MPI) [3]. It is portable, stable, widely available,and it has become the *de facto* standard for writing parallel programs on distributed systems.

However, MPI was designed for communication between computers over networks and incurs too much protocol overhead and wastes the extremely low inter-core latency that many-core offers. Moreover, MPI does not address support for distributed data structures. In this many-core era, the aforementioned locality problem is aggravated; programmers have to explicitly deal with decomposing data, mapping tasks, and performing synchronization, the massively increasing number of cores of deep memory hierarchy may introduce additional challenges to programmers. To this end, some researchers argue that MPI model has fundamental flaw that may not survive as a viable parallel programming model for future many-core systems [159].

Due to the above issues, while the majority of the software community believes that many-core processors are going to become the mainstream in the future, people have not yet reached (or even come close to reaching) a consensus on how to efficiently and effectively exploit the performance potential of those architectures.

1.4 Contributions

Programming many-core systems is a new area; no general model exists that can be used to verify the performance of the algorithms/applications/methodologies developed before they are implemented. It is thus the goal of this dissertation to use case studies to understand issues in designing and developing scalable, high-performance scientific computing algorithms for many-core architectures, get in-depth experience on programming and optimizing applications on those architectures, and then provide insights into implementation effort and performance behavior of optimizations and algorithmic properties for many-core architectures.

In this dissertation, we investigate the following two problem/architecture combinations as case studies,

- **Fast Fourier Transform on IBM Cyclops-64**. Fast Fourier Transform

(FFT) [35] is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFT is of great use across a large number of fields, like spectral analysis, data compression, partial differential equations, polynomial multiplication, multiplication of large integers [36, 94], etc. The IBM Cyclops-64 (C64) chip employs the many-core design by integrating 160 general-purpose processing elements, same number of memory banks, and a crossbar on a single chip. A C64 chip features massively on-chip parallelism, massive on-chip memory bandwidth, a large register file for each thread unit and an explicitly managed multiple level memory hierarchy without data cache.

- **Molecular Dynamics on NVIDIA CUDA**. Molecular dynamics (MD) [55] is a simulation method in which atoms and molecules are allowed to interact for a period of time by approximations of known physics, giving a view of the motion of the particles. This kind of simulation is frequently used in the fields of biology, chemistry, and medicine, as well as in materials science, to allow scientists to study the motion of individual atoms in a way which is not possible in traditional laboratory experiments. CUDA stands for Compute Unified Device Architecture, a parallel architecture developed by NVIDIA for general purpose parallel computing. CUDA devices have one or multiple streaming multiprocessors, each of which consists of one instruction issue unit, eight scalar processor cores, and two transcendental function units. CUDA architecture features both on-chip memory and off-chip memory. Unlike general-purpose CPUs, CUDA is a throughput-oriented architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very fast.

FFT and MD were chosen because they represent commonly used techniques

in a wide variety of scientific applications and have performance characteristics typical of many scientific applications. In addition, they have small code segments whose behavior we can understand and directly track to specific architectural characteristics. Each aforementioned application has certain unique characteristics, and consequently, each algorithm has to be designed and according to the features the problem dictates. Determining parallelism, problem decomposition, runtime load balancing, scalability, and performance analysis are important issues that are considered.

Accordingly, the main contributions of this dissertation are summarized in two main areas as follows,

- **Optimizing the Fast Fourier Transform for IBM Cyclops-64**

    1. We design and implement scalable high-performance parallel FFT algorithms for the C64 architecture. We analyze the optimization challenges and opportunities for FFT problems, and identify domain-specific features of the target problems and match them well with some key many-core architecture features. The impacts of various optimization techniques and effectiveness of the target architecture are addressed quantitatively. The optimization procedure, together with the experimental results, provides valuable information for compilation optimizations and algorithm design and optimization for many-core architectures.
    2. We propose a performance model that estimates the performance of parallel FFT algorithms for an abstract many-core architecture, which captures generic features and parameters of several real many-core architectures. It is therefore applicable for any architecture with similar features. We derive the performance model based on cost functions for three main components of an execution: the memory accesses, the computation, and the synchronization. We estimate the memory access delay by jointly

considering the memory access pattern of an FFT algorithm and the architecture configuration; we estimate the computation time by simulating the execution of the computation kernel; we estimate the synchronization cost via an experiment-based approach. This performance model provides valuable insights for designing FFT algorithms on many-core systems, and tuning them to achieve the maximum performance.

3. We evaluate our performance model on the C64 architecture. Experimental results from both simulations and the executions on the real hardware have verified the effectiveness of our performance model; our model can predict the performance trend accurately. The experimental results also reveal that the memory access delay has a crucial impact on performance of a parallel FFT algorithm for many-core architectures. Therefore programmers are suggested to optimize use of local memory and higher radix algorithms to reduce memory traffic requirements.

- **Exploring Fine-grained Task-based Execution on Graphics Processing Unit-enabled Systems**

  1. We propose a fine-grained, task-based execution framework for systems equipped with graphics processing units (GPUs). The framework allows computation tasks to be executed at a finer granularity than what is supported in existing GPU APIs such as NVIDIA CUDA. This fine-grained approach is particularly attractive because of the following reasons. First, this fine-grained scheme is expected to utilize the hardware more efficiently than the CUDA scheduler for unbalanced workload on single-GPU systems. Second, it provides means for achieving efficient, and dynamic load balancing on multi-GPU systems. While scheduling fine-grained tasks enables good load balancing among multiple GPUs,

concurrent execution of multiple tasks on each single GPU solves the hardware underutilization issue when tasks are small. Third, since our approach allows the overlapping executions of homogeneous/heterogeneous tasks, the programmers will have the flexibility to arrange their applications with fine-grained tasks and apply dataflow-like solutions to increase the efficiency of the program execution.

2. We implement our framework with CUDA. We evaluate the performance of the basic operations of this implementation with micro-benchmarks.

3. We evaluate the solutions based on our framework with a MD application. Experimental results with a single-GPU configuration show that our solutions can utilize the hardware more efficiently than the CUDA scheduler, for unbalanced problems. For multi-GPU configurations, our solutions achieve nearly linear speedup, load balance, and significant performance improvement over alternative implementations based on the canonical CUDA paradigm. Performance analyses reveal that the interaction between the task execution granularity and the particular algorithm can lead to significant impact upon the performance.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 gives a brief introduction of the C64 architecture and its system software infrastructure. Chapter 3 presents the background knowledge of FFT. Chapter 4 presents parallel 1D and 2D FFT algorithms that are optimized for the C64 chip architecture with detailed performance analyses. Chapter 5 presents a model for performance prediction of parallel FFT algorithms for an abstract many-core architecture, and its evaluation with the C64 architecture. Chapter 6 gives a brief introduction of the NVIDIA

CUDA architecture. Chapter 7 presents the background knowledge of MD simulations. Chapter 8 presents our fine-grained task-based framework for GPU-enabled systems. Chapter 9 evaluates the solutions based on our framework with micro-benchmarks and a MD application. Chapter 10 concludes this dissertation with future work.

PART I

OPTIMIZING THE FAST FOURIER TRANSFORM FOR IBM CYCLOPS-64

Chapter 2

IBM CYCLOPS-64

Cyclops-64 (C64) is a petaflop supercomputer project under development at IBM Research Center. C64 is designed to serve as a dedicated compute engine for running high performance applications. Using a cellular organization, a C64 petaflop machine consists of thousands of C64 many-core chips connected through a 3D-mesh network.

2.1 C64 Chip Architecture

The C64 chip, shown in figure 2.1, is the core computation engine of the C64 supercomputer system. One C64 chip features massive parallelism with 80 64-bit processors, each consisting two thread units (TUs), two 32KB SRAM banks, and one floating-point unit (FPU). Each thread unit is a single-issue, in-order RISC processor operating at a moderate clock rate (500MHz). Therefore, a C64 chip contains 160 processing elements.

Other on-chip components include 16 shared instruction caches (each is shared by 5 processors), 4 off-chip DRAM controllers, A-Switch, and etc. All on-chip resources are connected to an on-chip pipelined crossbar network with a large number of ports (96×96), which sustains a 4GB/s bandwidth per port, thus 384GB/s in total. The Instruction Set Architecture (ISA) of C64 supports *Floating Multiply-Add* instructions, which can be issued at every cycle. Therefore, the theoretical peak performance of a C64 chip is 80Gflops.

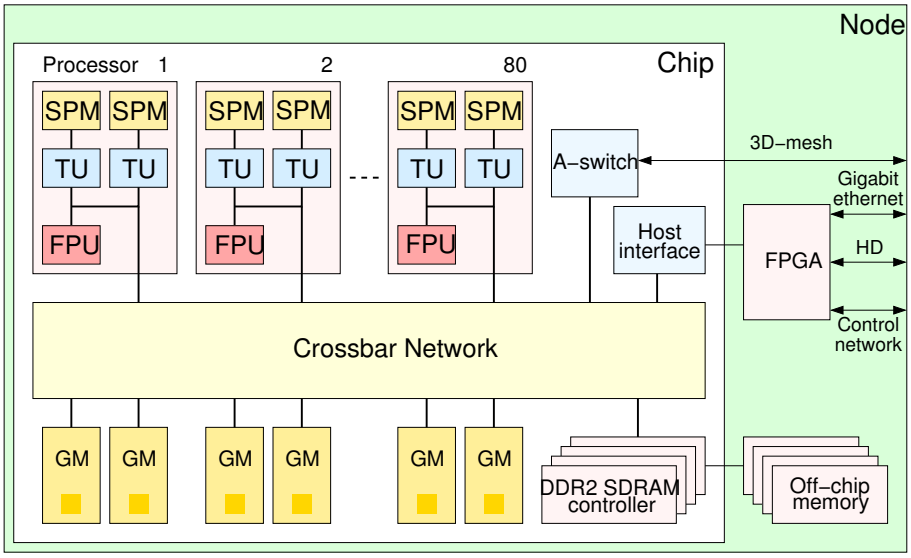
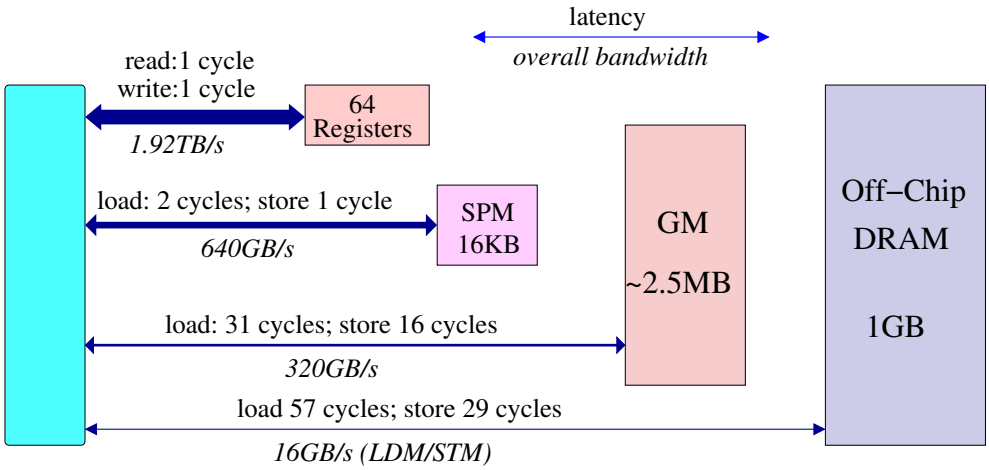


Figure 2.1: Cyclops-64 Chip Architecture

Courtesy: This figure was first created by Alban Douillet and then successively revised by Juan del Cuvillo and the author of this dissertation.

The C64 architecture has a segmented memory space, including the scratch-pad memory (SPM), on-chip global interleaved memory (GM), and off-chip DRAM. It is interesting to note that C64 does not have data cache. Instead, the on-chip SRAM banks are partitioned into the SPM and GM. Both the SPM and GM are globally addressable through the crossbar network by all TUs. While the GM are accessible among all threads on the chip of a uniform latency, the SPM is regarded as the fast local memory of the corresponding thread unit. Figure 2.2 shows the latency and bandwidth for accessing different segments in the C64 memory hierarchy.

An important property of the crossbar switch is that memory access instructions issued by one TU to any on-chip GM bank, or off-chip DRAM bank, experience the same latency in the crossbar. This equal-latency property makes the on-chip memory model as sequential consistency [160], which implies that no *memory fence* instruction is required to enforce ordering relation between memory accesses.



Courtesy: This figure was first created by Ziang Hu and then revised the author of this dissertation.

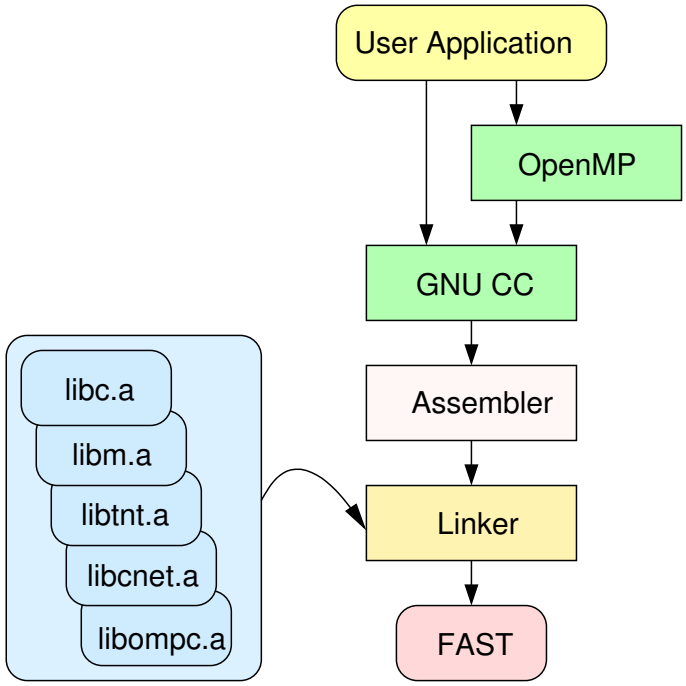
Figure 2.2: Cyclops-64 Memory Hierarchy

All thread units within a chip are connected by a 16-bit wide signal bus, which provides a means to efficiently implement barriers. Furthermore, the C64 ISA features a large number of atomic in-memory instructions. All these greatly facilitate the thread-level parallelism with fast inter-thread synchronizations. C64 provides no hardware support for context switch, and uses a non-preemptive thread execution model.

2.2 System Software

Figure 2.3 shows the C64 system software toolchain [41], which is used for software and application development on the C64 system. The toolchain consists of following basic components:

- **Binary utilities (binutils):** C64 assembler, linker, and objdump, etc. The binutils is ported from GNU binutils-2.11.2 [61].



Courtesy: Juan del Cuvillo.

Figure 2.3: Cyclops-64 System Software Toolchain

- **GNU CC compilers:** C and Fortran compilers, which are ported from GCC-3.2.3/GCC-4.0.0/GCC-4.1.1 suite [60]. To fully exploit the explicitly addressable multi-layered memory hierarchy of C64, the compiler, assembler, and linker are enhanced to support segmented memory spaces that are not contiguous. In other words, multiple sections of code, initialized and uninitialized data can be allocated on different memory regions.

- **FAST simulator:** FAST stands for Functionally Accurate Simulator Toolset [40]. It is an execution-driven, binary-compatible simulator of a multi-chip

Table 2.1: FAST Simulation Parameters

Component	# of units	Params./unit
Threads	160	single in-order issue, 500MHz
FPUs	80	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	160	32KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

multithreaded C64 system. Before the actual C64 chip is available, the development and research of system software and scientific and engineering applications are conducted on FAST. It accurately reproduces the functional behavior of hardware components such thread units, on-chip and off-chip memory banks, and the 3D-mesh network. Table 2.1 shows the major simulation parameters of FAST.

- **TiNy-Threads microkernel/runtime system library:** The thread virtual machine (TVM) of C64 is called TiNy-Threads (TNT) [42]. The TVM includes the TNT non-preemptive thread model, memory model, and synchronization model. In particularly, in the TNT thread model, thread execution is non-preemptive and software threads map directly to hardware thread units in a one-to-one fashion. In other words, after a software thread is assigned to a hardware thread unit, it will run on that hardware thread unit until completion. Based on TNT TVM, a microkernel and the TNT runtime system are customized for the unique features of the C64 architecture [42]. The TNT library provides user and library developers an efficient Pthreads-like API for thread level parallel programming purpose.

- **Standard C and math libraries:** These libraries are derived from those in newlib-1.10.0 [102]. Functions (libc/libm) are thread safe, i.e. multiple threads can call any of the functions at the same time.

- **CNET communication protocol and library:** The CNET communication library is used to manage the A-switch communication hardware [44] to provide user-level remote memory read/write functionality.

- **SHMEM:** The SHMEM [114] shared memory access library, which is built on CNET, is developed to support high-level shared memory programming model across C64 nodes. SHMEM provides a shared global address space, data movement operations between locations in that address space, and synchronization primitives that greatly simplify programming for a multi-chip system such as C64.

Chapter 3

FAST FOURIER TRANSFORM

The fast Fourier transform (FFT) is a fast algorithm for computing the discrete Fourier transform (DFT). In the literature, the FFT has been extensively studied and implemented as an important frequency analysis tool in many areas such as spectral analysis, data compression, partial differential equations, polynomial multiplication, multiplication of large integers, etc.

3.1 Discrete Fourier Transform

Before describing the FFT a brief introduction to DFT is first given. Basically, the computational problem for the DFT is to compute the sequence $X(k)$ of N complex-valued numbers given another sequence of data $x(n)$ of length N, according to Equation 3.1.

$$X(k) = \sum_{n=0}^{N-1} x(n)\omega_N^{kn}, \quad 0 \leq k \leq N-1 \tag{3.1}$$

where $\omega_N = e^{-i2\pi/N}$. In general, the data sequence $x(n)$ is also assumed to be complex-valued. Similarly, The inverse DFT (IDFT) becomes

$$x(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(k)\omega_N^{-kn}, \quad 0 \leq n \leq N-1 \tag{3.2}$$

For each value of k, direct computation of $X(k)$ involves N complex multiplications ($4N$ real multiplications) and $(N-1)$ complex additions ($4N-2$ real additions). Consequently, to compute all N values of the DFT requires N^2 complex multiplications and $N^2 - N$ complex additions.

Direct computation of the DFT is basically inefficient primarily, because it does not exploit two important properties of ω_N. In particular, these two properties are:

$$\omega_N^{k+N/2} = -\omega_N^k \tag{3.3}$$

$$\omega_N^{k+N} = \omega_N^k \tag{3.4}$$

where Equations 3.3 and 3.4 are known as the *symmetry* property, and the *periodicity* property, respectively.

3.2 Fast Fourier Transform

FFT algorithms, on the other hand, exploit these two properties and achieve a computationally efficient solution for solving DFT. Without loss of generality, let us consider the computation of the $N = 2^v$ point DFT by the *divide-and-conquer* approach. We split the N-point data sequence into two $N/2$-point data sequences $f_1(n)$ and $f_2(n)$, corresponding to the even-numbered and odd-numbered samples of $x(n)$, respectively, that is,

$$\begin{aligned} f_1(n) &= x(2n) \\ f_2(n) &= x(2n+1) \end{aligned}$$

Thus $f_1(n)$ and $f_2(n)$ are obtained by decimating $x(n)$ by a factor of 2. Now the N-point DFT can be expressed in terms of the DFT's of the decimated sequences as follows:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)\omega_N^{kn}, \qquad k = 0, 1, \cdots, N-1 \\ &= \sum_{n \in even} x(n)\omega_N^{kn} + \sum_{n \in odd} x(n)\omega_N^{kn} \\ &= \sum_{m=0}^{N/2-1} x(2m)\omega_N^{2mk} + \sum_{m=0}^{N/2-1} x(2m+1)\omega_N^{k(2m+1)} \end{aligned} \tag{3.5}$$

With the substitution of $\omega_N^2 = \omega_{N/2}$, , Equation 3.5 can be expressed as,

$$\begin{aligned} X(k) &= \sum_{m=0}^{N/2-1} f_1(m)\omega_{N/2}^{km} + \omega_N^k \sum_{m=0}^{N/2-1} f_2(m)\omega_{N/2}^{km} \\ &= F_1(k) + \omega_N^k F_2(k), \qquad k = 0, 1, \cdots, N-1 \end{aligned} \tag{3.6}$$

where $F_1(k)$ and $F_2(k)$ are the $N/2$-point DFTs of the sequence $f_1(m)$ and $f_2(m)$, respectively.

Since $F_1(k)$ and $F_2(k)$ are periodic, with Equation 3.4, we have $F_1(k+N/2) = F_1(k)$ and $F_2(k+N/2) = F_2(k)$, for a period of $N/2$. Further, by applying Equation 3.3, the original DFT could be expressed as,

$$\begin{aligned} X(k) &= F_1(k) + \omega_N^k F_2(k), \qquad k = 0, 1, \cdots, N/2-1 \\ X(k+\frac{N}{2}) &= F_1(k) - \omega_N^k F_2(k), \qquad k = 0, 1, \cdots, N/2-1 \end{aligned} \tag{3.7}$$

One important observation of the above equations is that the direct computation of both $F_1(k)$ and $F_2(k)$ requires $(N/2)^2$ complex multiplications. Also, additional $N/2$ complex multiplications are required to compute $\omega_N^k F_2(k)$. Hence the computation of $X(k)$ requires $2(N/2)^2 + N/2 = N^2/2 + N/2$ complex multiplications. This is around *half* of complex multiplications required for the direct computation of $X(k)$. The decimation of the data sequence can be repeated again and again until the resulting sequences are reduced to one-point sequences. For $N = 2^v$, this decimation can be performed $v = \log_2 N$ times. Thus the total number of complex multiplications is reduced to $(N/2)\log_2 N$. The number of complex additions is $N \log_2 N$.

Consequently, the FFT gives an $\Theta(N \log_2 N)$ algorithm for computing DFT. The algorithm described above is usually referred to as the radix-2 Cooley-Tukey FFT algorithm [35], and the specific computation is known as a butterfly operation, which is shown in Figure 3.1.

Generally speaking, the implementation of the above recursive FFT algorithm introduces non-negligible recursion overhead, thus it is not favored. Another

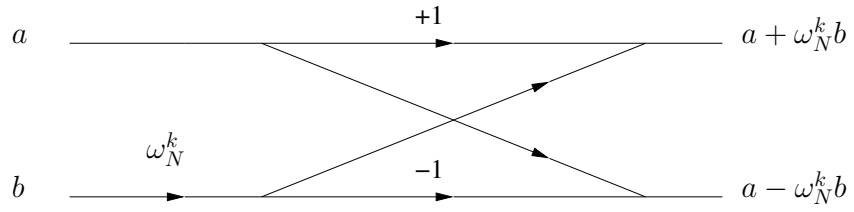


Figure 3.1: Cooley-Tukey Butterfly Operation

approach is to employ the iterative implementation. The iterative algorithm subdivides the resulting sub-problems iteratively until the problem size becomes one. In order to achieve such an implementation, the input data has to be reordered before the butterfly computations are performed.

For example, if we consider the case where $N = 8$, we know that the first decimation yields the sequence $x(0), x(2), x(4), x(6)$, $x(1), x(3), x(5), x(7)$, and the second decimation results in the sequence $x(0), x(4)$, $x(2), x(6)$, $x(1), x(5)$, $x(3), x(7)$. This shuffling of the input data sequence has a well-defined order as can be ascertained from observing Figure 3.2 and Figure 3.3, which illustrate the decimation of the eight-point sequence.

In the Cooley-Tukey algorithm, this permutation is performed before the butterfly computations. Figure 3.4 shows an example of the iterative FFT decomposition of 8 points using the Cooley-Tukey algorithm. Before the butterfly computation, the bit-reversal permutation is performed on the input data. Then the computation is decomposed through 3 stages of butterfly operations.

3.3 Multi-dimensional Discrete Fourier Transform

The ordinary DFT computes the one-dimensional (1D) dataset: a sequence of data $x(n)$ that is a function of one discrete variable n. More generally, the multi-dimensional DFT of a multi-dimensional array $x(n_1, n_2, \cdots, n_d)$ that is a function

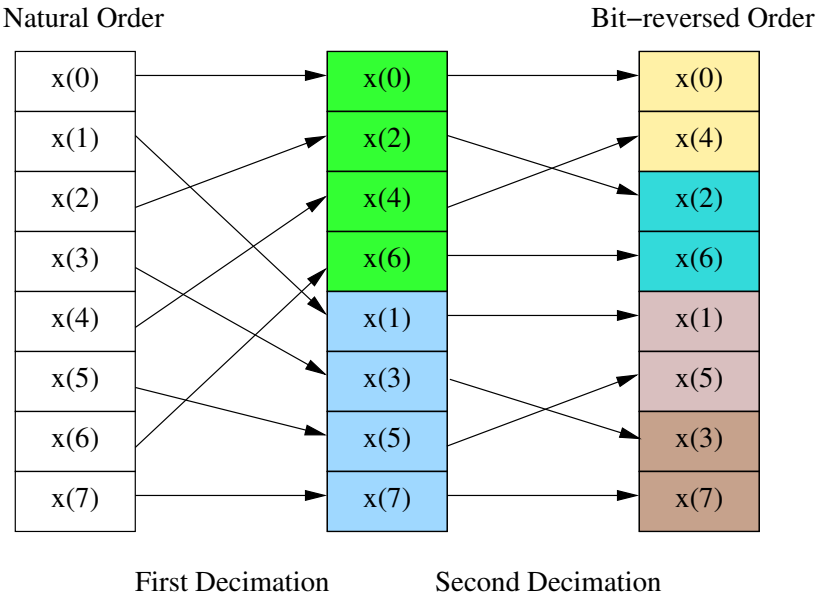


Figure 3.2: Bit Reversal of 8-point Data

of d discrete variables $n_l = 0, 1, \cdots, N_l - 1$ for l in $1, 2, \cdots, d$ is defined as,

$$X(k_1, k_2, \cdots, k_d) = \sum_{n_1=0}^{N_1-1} \left(\omega_{N_1}^{k_1 n_1} \sum_{n_2=0}^{N_2-1} \left(\omega_{N_2}^{k_2 n_2} \cdots \sum_{n_d=0}^{N_d-1} \omega_{N_d}^{k_d n_d} x(n_1, n_2, \cdots, n_d) \right) \cdots \right)$$

where $\omega_{N_l} = e^{-i2\pi/N_l}$.

Computationally, the multi-dimensional DFT is simply the composition of a sequence of 1D DFTs along each dimension. For example, in the two-dimensional (2D) case $x(n_1, n_2)$ one can first compute the N_1 independent DFTs of the rows, i.e., along n_2, to form a new array $y(n_1, k_2)$, and then compute the N_2 independent DFTs of y along the columns (along n_1) to form the final result $X(k_1, k_2)$. Or, one can transform the columns and then the rows, the order is immaterial because the nested summations above are commutative. This is known as a *row-column* algorithm. Because of this, given a 1D FFT algorithm, one way to efficiently compute the multidimensional DFT is to perform 1D FFT alternately on each dimension of

$(n_2n_1n_0)$		$(n_0n_2n_1)$		$(n_0n_1n_2)$
(000)	→	(000)	→	(000)
(001)	→	(100)	→	(100)
(010)	→	(001)	→	(010)
(011)	→	(101)	→	(110)
(100)	→	(010)	→	(001)
(101)	→	(110)	→	(101)
(110)	→	(011)	→	(011)
(111)	→	(111)	→	(111)

Figure 3.3: Binary Representation of the Bit Reversal

the data, interleaved with data transpose steps. This method is called the multi-dimensional FFT algorithm, and is easily shown to have a $\Theta(N \log_2 N)$ complexity, where $N = N_1 N_2 \cdots N_d$ is the total number of data points.

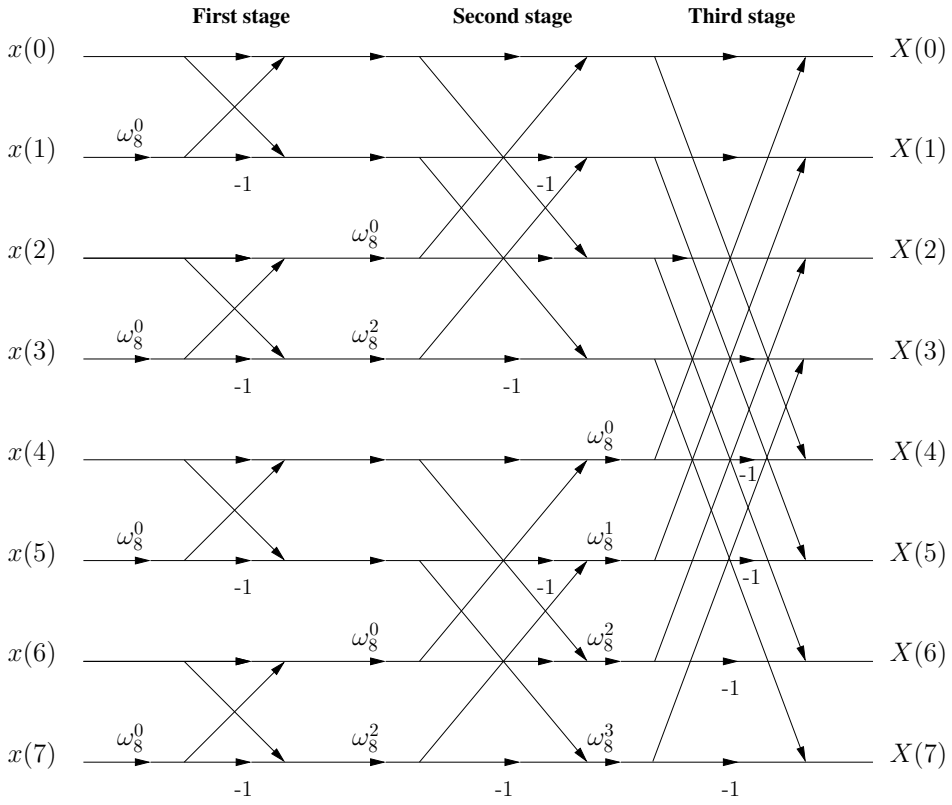


Figure 3.4: 8-point Cooley-Tukey FFT Example

Chapter 4

OPTIMIZING FFT ALGORITHMS

In this chapter, we discuss our experiences on the implementation, analysis and optimizations of the FFT on C64 architecture. In the experiments, we consider the data sizes of 2^{16} and 256×256 for 1D FFT and 2D FFT, respectively. In both cases, the input data are double-precision complex numbers, and can fit into on-chip GM. The twiddle factors are pre-computed and stored in on-chip GM as well. All the experiments were conducted on the FAST simulator, using up to 128 TUs on a C64 chip (unless otherwise explicitly stated).

We start with a base parallel implementation. Then, we carefully analyze the FFT algorithm features, identify a set of important issues on problem decomposition, load balancing, work distribution, data-reuse, register tiling, and instruction scheduling taking into account the memory hierarchy, propose optimization methods to address these issues and demonstrate how we explore corresponding C64 architecture features. We then set up the experiments based on the analysis, and find the optimal parameters that match the C64 architecture features.

4.1 1D FFT

For the 1D FFT, we employ the iterative radix-2 Cooley-Tukey algorithm, which requires $\Theta(N \lg_2 N)$ complex multiplication operations for N-point data.

4.1.1 Base Parallel Implementation

Before we go into details about the implementation, let us first introduce an important definition: work unit. A *work unit* is an arbitrarily defined piece of the work that is the smallest unit of concurrency that the parallel program can exploit. In other words, an individual work unit could be executed by only one processing element. Given this definition, the concurrency in a problem can only be exploited across work units. The size of a work unit may vary in different implementations. If the amount of work in a work unit is small, it is a fine-grained work unit; otherwise, it is a coarse-grained work unit. In this base parallel FFT implementation, we consider a butterfly operation to be a work unit, which includes 1) read 2-point data and the twiddle factor from GM, 2) perform a butterfly operations upon them, and then, 3) write the 2-point results back to GM. We call this a 2-point work unit because it contains 2 points that can be computed independently from other data. This design is best described in Figure 4.1, which shows the computing of a 4-point FFT with 2-point work units. In this figure, each work unit is shown as a rectangle. To achieve a balanced workload among all threads, the work units are assigned to threads in a round-robin configuration, during each stage of the FFT computation; the color of a rectangle means that this work unit is assigned to a specific thread for computing. Barriers are used to synchronize threads before the next pass starts. Here we want to clarify that *pass* is different from the butterfly computation *stage*. One pass may include one or more multiple butterfly computation stage(s).

Figure 4.2 show how this approach is implemented with TNT libraries (refer to Chapter 2), where arrays $x[]$ and $w[]$ are of *double* type and are located in GM.

This fine-grained approach matches the natural granularity of the FFT in the sequential program structure, which is the smallest unit of concurrency that the FFT exposes. This parallel implementation has a performance of 6.54Gflops.

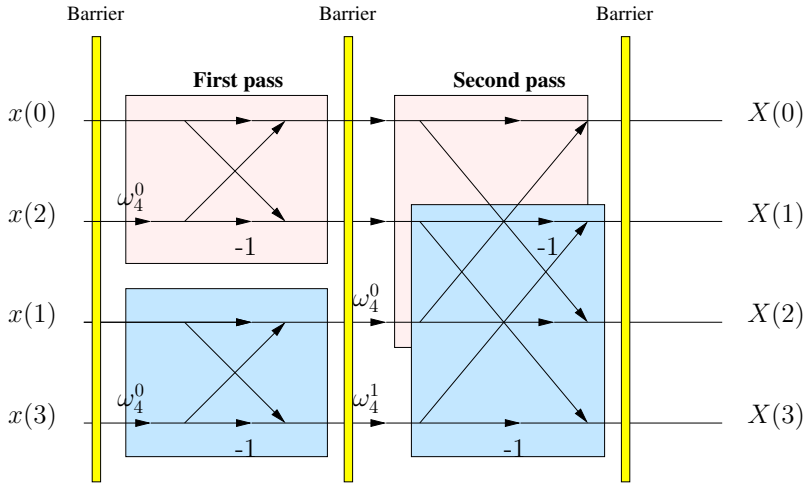


Figure 4.1: Example of 2-point Work Unit

4.1.2 Optimal Work Unit

In the above implementation, at each pass, barriers are used to control the accesses to the shared data, which imply large synchronization overhead. Decreasing the number of synchronizations can reduce such overhead and potentially improve the performance. On the other hand, since the function that processes each work unit is the kernel part in the FFT computation, we would like to have a closer look into this function and see whether any optimizations could be applied with regard to the large register files of C64. Referring to Figure 4.2, in the base implementation, a work unit consists of 6 load operations, 10 double-precision floating point operations, and 4 store operations, besides the integer operations for computing the indexes. We definitely cannot reduce the number of floating point operations, which is inherent to the FFT algorithm itself. Then, could we reduce the number of memory operations? Obviously, the answer is no in this case.

Let us look at an alternative approach. By using 2-point work units, a 4-point FFT computation can be completed in two passes and requires 2 such work units at each pass, 4 in total. In other words, this computation requires 24 load operations,

```
...
/* determines the logic thread ID of a calling thread */
my_thread = tnt_my_thread();
/* determines the number of available threads */
threads   = tnt_num_spmd_thread();
...
/* there are two passes for a 4-point FFT with 2-point work units */
for (pass = 0; pass < 2; pass ++) {
  /* there are two work units at each pass,
   * for a 4-point FFT with 2-point work units */
  for (work = my_thread; work < 2; work += threads) {
    double a_r, a_i, b_r, b_i, w_r, w_i, \
           t0_r, t0_i, t1_r, t1_i, t2_r, t2_i;
    ...
    /* compute the indexes for loading data */
    a_index = ...
    b_index = ...
    w_index = ...
    /* load from the memory */
    a_r = x[a_index];
    a_i = x[a_index + 1];
    b_r = x[b_index];
    b_i = x[b_index + 1];
    w_r = w[w_index];
    w_i = w[w_index + 1]
    /* t0 = w * b */
    t0_r = w_r * b_r + w_i * b_i;
    t0_i = w_r * b_i - w_i * b_r;
    /*  a = a + w * b = a + t0 */
    t1_r = a_r + t0_r;
    t1_i = a_i + t0_i;
    /*  b = a - w * b = a - t0 */
    t2_r = a_r - t0_r;
    t2_i = a_i - t0_i;
    /* store back to the memory */
    x[a_index]     = t1_r;
    x[a_index + 1] = t1_i;
    x[b_index]     = t2_r;
    x[b_index + 1] = t2_i;
  }
  tnt_barrier_wait(NULL);
}
```

Figure 4.2: TNT Code Segment of 2-point Work Units

40 floating point operations, and 16 store operations. Instead of containing 2 points, if one work unit has 4-point data that can be computed independently from other data, the thread can read all data into registers, perform required computation, and write back the results. Following the convention, we call this 4-point work unit. Figure 4.3 shows the idea of 4-point work unit with the computing of a 4-point FFT, where each work unit is shown as a rectangle. The workload of a 4-point work unit includes 1) read 4-point data and the corresponding 4 twiddle factors from GM, 2) perform 2-stage butterfly operations upon 4 points, and then, 3) write the 4-point results back to GM. In this case, this work unit consists of 16 load operations, 40 double-precision floating point operations, and 8 store operations. Similar to the base implementation, threads need to be synchronized after all of them finish their 4-point work units, which are 2-stage FFT computations. Compared with the previous implementation, this method eliminates half of the barriers (besides the first barrier used before the entire computing), and reduces the number of memory operations by 40%, and increases the percentage of floating point operations to the total number of instructions from 50% to 62.5% [1].

This is definitely an encouraging sign to achieve better performance. Let us extend this idea more ambitiously. Assuming we have a machine with an unlimited number of registers. Then, In general, using a work unit of N-point data can get rid of $(\lg_2 N - 1)$ barriers. Moreover, the percentage of floating point operations to the total number of instructions is $\frac{5N \lg_2 N}{6N \lg_2 N + 4N}$. One can verify this formula with the above two examples, i.e., Figure 4.1 and Figure 4.3. If one work unit has 2^{16}-point data, then only one barrier is needed and the percentage of floating point operations to the total number of instructions would increase to 80%! It is clear that the more data one work unit has, the better computation-communication ratio

[1] Please note that we ignore the integer operations to simplify the analysis. While this introduces inaccuracy, the trend remains the same.

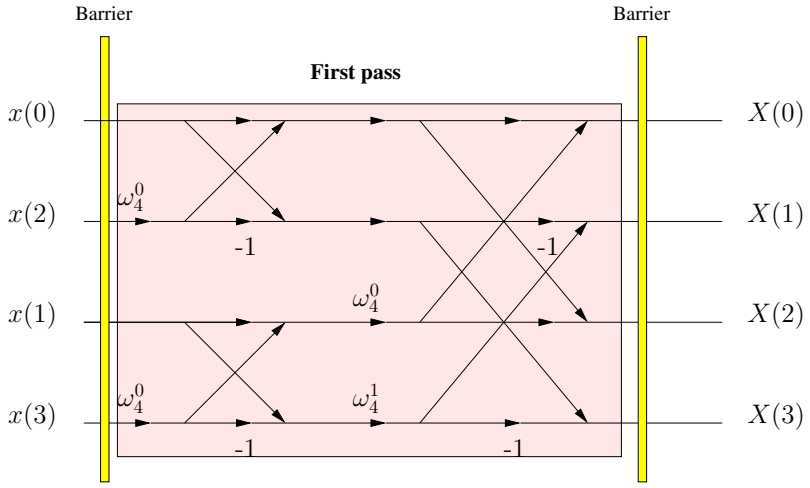


Figure 4.3: Example of 4-point Work Unit

we could achieve.

However, no practical machine has unlimited registers. Further, a huge work unit may limit the concurrency exposed by the program. More data a work unit has, less threads we need for a given FFT computation. So we have to decide an appropriate size of the work unit, which should expose enough parallelism and still fully utilize the register file without serious register spilling.

Let us examine the above example again. Although each C64 TU has a total of 64 registers, some of them cannot be used in user-level applications, for example, R0 (Permanent Zero), R1 (Interrupt return location), etc. Also, function input arguments are passed through registers. Moreover, some intermediate computing results have to be stored in registers as well. So, in general, the number of registers available for user-level programs are around 50 on a C64 TU. For 4-point work unit, it needs 8 registers for input data, 4 registers for the corresponding indexes, another 8 registers for the twiddle factors. Thus the total number of registers needed would be 20, besides few registers used to keep intermediate results. While completing this work unit will not cause register spilling, it does underutilize the register file;

about half of the registers are not used during the entire computation. Considering the case of 8-point work unit, it requires 16 registers for input data, 8 registers for the corresponding indexes, another 24 registers for the twiddle factors. The total number of registers necessary would be around 48. In theory, executing such 8-point work unit on C64 will use most of the available registers of a C64 TU, and it will not generate (serious) register spilling. If we go a little bit further with the 16-point work unit, however, the total number of registers needed during the computation increases to 112, which imposes much greater pressure on the register file and will certainly introduce serious register spilling and thus typically will slow down the computation. Therefore, based on our analysis, the 8-point work unit could be the best choice for C64. Note that 8-point work unit implies a 3-stage FFT computation. Given a FFT computation with n-point data, when $\lg_2 n$ cannot be divided exactly by 3, the last $(\lg_2 n - \frac{\lg_2 n}{3})$ stage(s) can be computed with 4-point work units or 2-point work units.

Our analysis and conclusions have been confirmed by the experimental results with different sizes of work units, which are shown in figure 4.4. Obviously the 8-point work unit outperforms other work unit sizes. After applying this 8-point work unit, we reach a performance of 13.17Gflops, which is 101.5% improvement over the base parallel implementation.

4.1.3 Special Handling of the First Stages

As shown in figure 3.4, every butterfly operation performs on consecutive data during the first stage. For example, the top left butterfly operation acts upon $x(0)$ and $x(4)$, which are contiguous in the memory after the bit-reversal permutation. It holds true that all points within the same work unit are consecutive in the memory before the first stage, for any valid size of work unit. This implies that less registers are required for the first $\lg_2 M$ stages, as when M-point work unit is used, only the

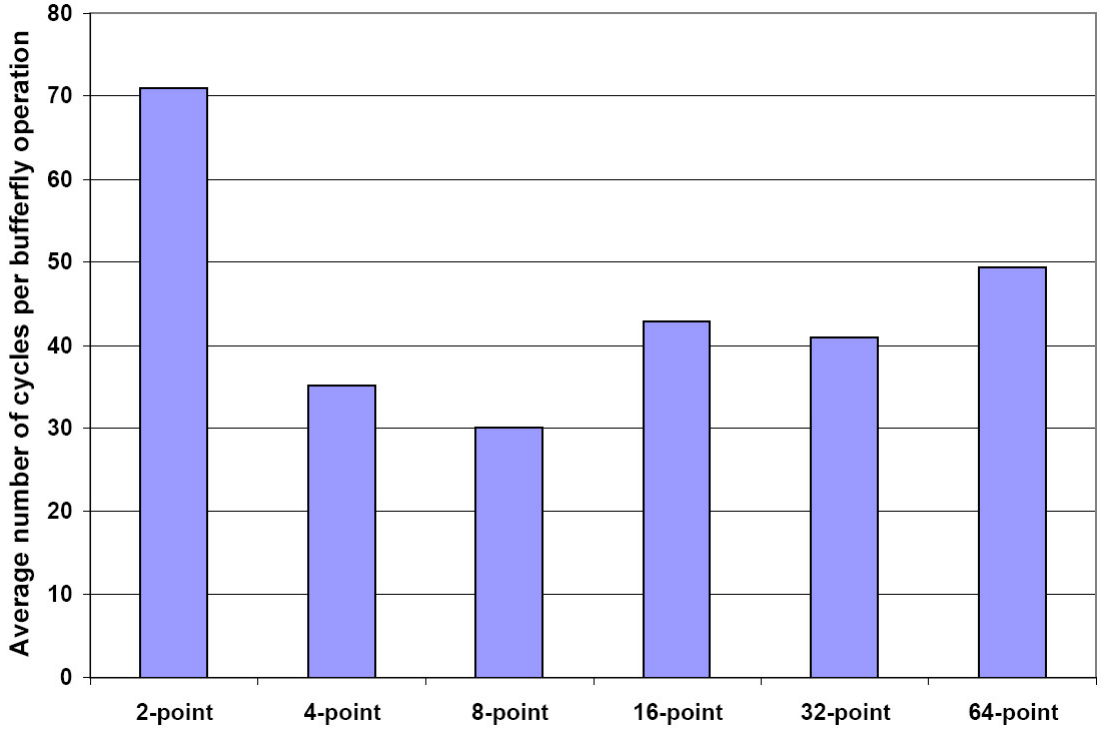


Figure 4.4: Number of Cycles per Butterfly Operation versus the Size of Work Unit

starting pointer and the size of the work unit are needed to access this work unit, instead of computing the indexes for all the points and keeping them in registers.

Inspired by this observation, we try to search the appropriate M, the size of work unit for the first $\lg_2 M$ stages. Since it is clear that $M \geq 8$, let us consider 16-point work unit again. We need 32 registers for the input data and 1 register for the starting address of this work unit, another 64 registers for the 32 twiddle factors. Thus the total number of registers necessary would be 97, which still exceeds the maximum available registers in C64 architecture. It seems that 8-point is the maximum size of work unit that we can use during the entire FFT; however, let us look at figure 3.4 more carefully. In this figure, all butterfly operations performed during the first stages are using the same twiddle factor ω_8^0. In the second stage,

only 2 distinct twiddle factors are used, i.e., ω_8^0 and ω_8^2. In general, in the i-th stage of a complete FFT computation, 2^{i-1} distinct twiddle factors are used, and they include all twiddle factors used in the preceding stages. In other words, during the execution of the first $\lg M$ stages, there are fewer twiddle factors being used. By knowing this fact, we re-consider the possibility of using 16-point work unit. Instead of 64, we only need 16 registers to keep 8 distinct twiddle factors used in the fist 4 stages. Thus the total number of registers required is 49, which can fit into the C64 register file. Further, we define these 8 twiddle factors as *macros* in the program. This approach further improve the performance by reducing the number of index-computing operations. While the inaccuracy introduced is well under control[2]. After applying these approaches for the first 4 stages, we achieve an improvement of 28.4% over the earlier implantation, while the absolute performance reaches 16.92Gflops.

4.1.4 Eliminating Unnecessary Memory Operations

Mathematically, in a 8-point work unit, all twiddle factors used in the "first" stage of this 8-point computation (not the first stage of the complete FFT computation) are of the same value. Half of the twiddle factors used in the "second" stage are of the same value, all the twiddle factors have distinct values in the "third" stage. Thus, only 1, 2, and 4 distinct twiddle factors are needed for the first, second, and third stage of the 8-point work unit computation, respectively. Thus we can reduce the computation for the indexes of the twiddle factors and subsequent memory operations. By eliminating these unnecessary instructions, we have an absolute performance of 17.97Gflops, which is a 6.2% improvement over the previous number.

[2] After applying the FFT and a subsequent IFFT, the variance between the results and the original data is in the order of $O(10^{-14})$.

4.1.5 Loop Unrolling

Recall that in the entire FFT computation, aside from the butterfly computations, the *bit-reversal permutation* usually accounts for substantial portion of the overall FFT computation time. Specifically, in the current implementation, this permutation takes 5.7% of the total execution time. In the kernel loop of the bit-reversal permutation, once the indexes of two points, to be permuted, are computed, the two corresponding points will be read from GM, swapped and written back to GM. Since C64 ISA has *Bit Gather* instructions that can be used to perform fast index computation, the most time-consuming part is the memory operations. To hide the memory latency, we unroll this kernel loop 4 times. By doing this, we accomplish an improvement of 25.0% for the permutation part, leading to a 1.4% improvement on the overall performance.

4.1.6 Register Renaming and Instruction Scheduling

The C64 architecture does not have data cache and each memory operation may have different latency depending on the target memory segment, i.e., SPM, GM, and DRAM. But most existing compilers assume a cache latency (cache hit) or a uniform memory latency (cache miss) when they do instruction scheduling. By manually applying register renaming and instruction scheduling on several kernel functions, we hide most of the latencies due to memory operations and floating point operations, and achieve a 13.7% improvement. The performance reaches 20.72Gflops.

4.1.7 Comparison with Memory Hierarchy Aware Compilation

While the above manual optimizations can achieve a relatively high performance, the entire process is tedious and error-prone. The different delays of memory instructions when accessing different memory segments have to be carefully investigated and manipulated. On the other hand, this work would be an ideal job for a

smart compiler that could identify the segments where variables reside, and apply the corresponding latencies when scheduling the instructions. Inspired by this idea, the C64 compiler was later tailored such that it accounts for the different latencies when accessing variables specified with segment pragmas when applying instruction scheduling (which is not part of this thesis). By employing this memory hierarchy aware compiler with the code from 4.1.5, it achieves a 8.8% improvement, which corresponds to a performance of 19.84Gflops. While the absolute performance is a little bit lower than the manually optimized code in 4.1.6 (it is still comparable to the latter), this optimization dramatically reduces the effort to achieve a high performance implementation on architectures with a deep memory hierarchy like C64. Figure 4.5 shows the performance of this optimized implementation. From these plots, we observe that the performance of this implementation scales nearly linearly up to 128 threads.

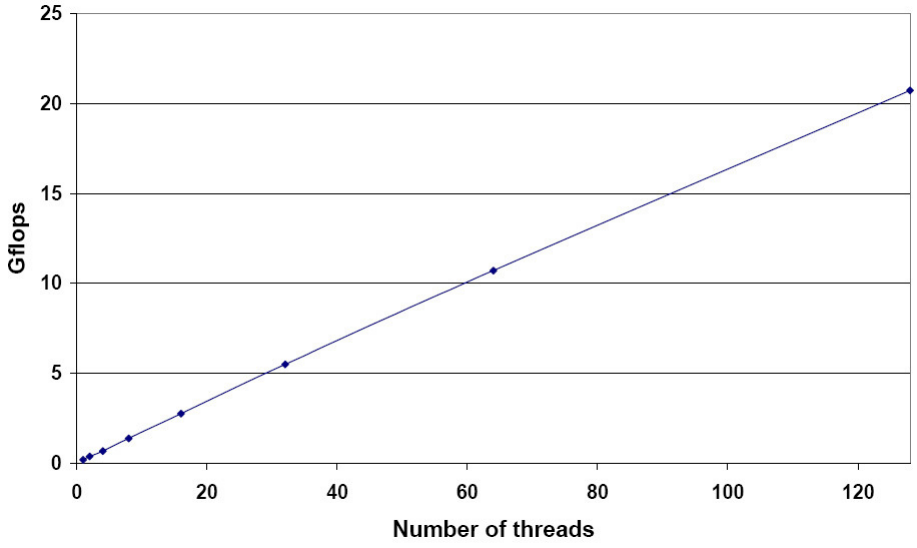


Figure 4.5: Performance of the Optimized 1D FFT Implementation

So far, we finish the optimizing 1D FFT implementation. We list all the

Table 4.1: 2^{16} 1D FFT Incremental Optimizations

Optimizations	GFLOPS	Speedup Over Base Version	Incremental Speedup
Base	6.54	1.00	0%
Optimal W.U.	13.17	2.02	101.5%
Special App.	16.92	2.59	28.4%
Eli. MEM Ops.	17.97	2.75	6.2%
Loop Unroll.	18.23	2.79	1.4%
Reg. & Inst.	20.72	3.17	13.7%

techniques applied and the corresponding results in Table 4.1. Figure 4.6 shows the graphic representation of those incremental optimizations. Note that the effect of the memory hierarchy aware compiler is not shown in either Table 4.1 or Figure 4.6. Among all optimizations, the ones of most significant improvement are the optimal work unit and the special approach for the first 4 stages. As we have discussed, they are achieved by carefully investigating the features of the FFT algorithm and C64 architecture features. Moreover, by examining the mathematical nature of FFT, redundant memory operations with regard to the twiddle factors was removed. All above optimizations show that the domain-specific knowledge is very important, some time critical, to achieve a desirable performance. On the other hand, traditional optimization techniques may still be able to play some roles in the many-core era, for example, the loop unrolling technique used in the bit-reversal permutation. However, efforts may be needed to identify those valid techniques. Meanwhile, it is clear that many-core system software, especially the compiler, needs to address many challenges due to many-core architectures. For example, the performance improvement due to the manual register renaming and instruction scheduling is significant, 13.7% over the previous version. Inspired by this observation, the compiler was later tailored with the memory hierarchy aware instruction scheduling, and was able to show a satisfactory performance with minimum programmer effort.

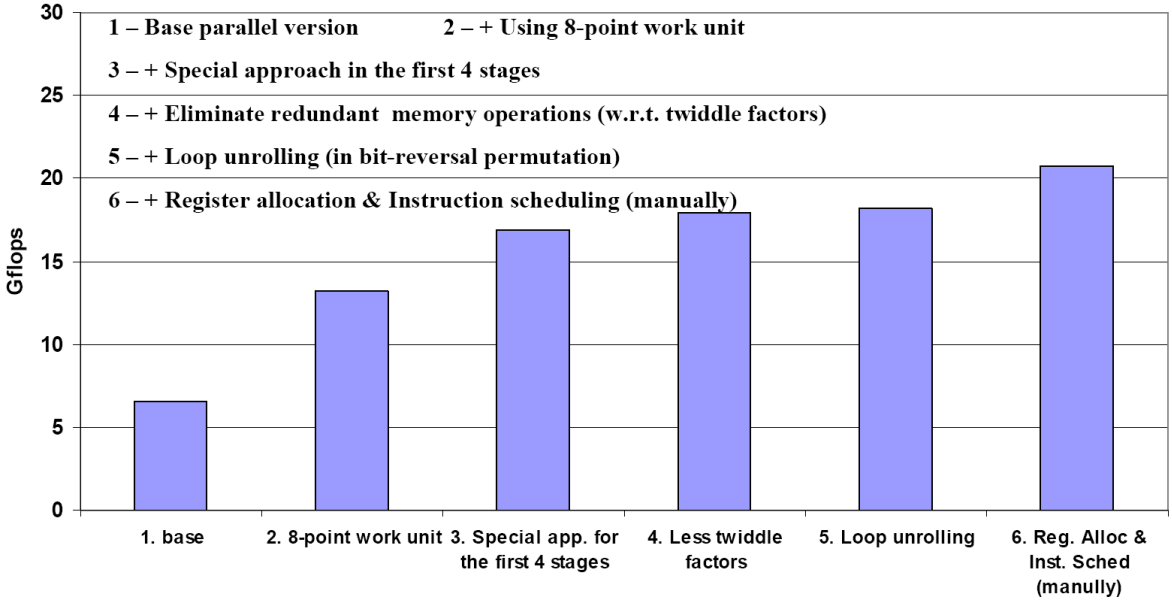


Figure 4.6: Effect of Optimization Techniques of 1D FFT Implementation (without the Memory Hierarchy Aware Compilation)

4.2 2D FFT

As mentioned in Chapter 3, the multidimensional FFT problem can be solved by performing 1D FFT alternately on each dimension of the data interleaved with data transpose steps. That is, for a $N \times N$ 2D FFT $x(i, j)$, one can simply perform a sequence of 1D FFTs by any 1D FFT algorithm: first transform along the row dimension $x(:, j)$, after all row FFTs are done, then transform along the column dimension $x(i, :)$. This is known as the conventional *row-column* algorithm. This method is easily shown to require $\Theta(N^2 \lg_2 N)$ complex multiplication operations. Our implementation of the parallel 2D FFT follows this row-column algorithm.

4.2.1 Base Parallel Implementation

In the base implementation, we simply employ one row/column FFT as a work unit. All row FFTs are independent of each other, so they can be computed in

parallel, the same with all column FFTs. After completing all row FFTs, a barrier is used to synchronize all threads before they perform the column FFTs. Work units are distributed to threads in the round-robin configuration. By utilizing the optimized 1D FFT implementation presented in the previous section, this parallel implementation achieves a performance of 15.11Gflops.

4.2.2 Load Balancing

The work unit scheme used in the base implementation is straightforward and can be easily implemented. However, it may hurt the performance due to the non-trivial load imbalance. For example, given a 64×64 2D FFT, using more than 64 threads will not produce any performance gain over using exactly 64 threads: while the first 64 threads are working on their own work units, other threads will remain idle because there is no work unit available for them. In other words, this simple work unit scheme does not expose enough concurrency to keep all threads busy at all times, thus limits the speedup achievable. To resolve this issue, we should use fine-grain work units and distribute them over all threads evenly. So, instead of having one entire 1D FFT as a work unit, we divide each row/column FFT into small tasks.

In this way, multiple threads may work on one single row/column FFT, just like what we did for 1D FFT. Based on what we have learned from 1D FFT, we still use 8-point work unit. While this "new" work unit scheme reduces the load imbalance issue, it needs more barriers to synchronize threads working on the same row/column FFT. Thanks to C64's hardware barrier support, these barriers do not introduce much overhead.

4.2.3 Work Distribution and Data Reuse

Given a set of work units defined in the previous section, one can distribute these work units to threads in a common round-robin scheduling. This method performs well and can distribute work units evenly as possible to all threads. However, it does not exploit the nature of the 2D FFT. In the 2D FFT computation, the exact same set of operations are repeatedly performed on each row/column FFT, including the bit-reversal permutation and the butterfly computation. For example, if $x(a, 0)$ and $x(b, 0)$ need to be swapped during the bit-reversal permutation, $x(a, j)$ and $x(b, j)$ need to be swapped during the bit-reversal permutation as well, for $0 \leq j < N$. This also holds true for the butterfly computation: if a butterfly operation with a twiddle factor ω is to be performed on $x(0, a)$ and $x(0, b)$, this butterfly operation should be performed on $x(i, a)$ and $x(i, b)$, for $0 \leq i < N$, with the same twiddle factor. This provides great opportunity for data reuse, thus it can reduce the index computations and memory operations. A *major-reversal* work distribution scheme is employed to exploit this opportunity. Namely, when a thread completes a work unit consisting of $\{x(a, i_o), x(a, i_1), \cdots, x(a, i_n)\}$ in a row FFT $x(a, :)$, instead of going row-major and locating another work unit in the same row FFT, it reuses the computed indexes, i.e., $\{i_0, i_1, \cdots, i_n\}$, and twiddle factors by going column-major to the row FFT $x(a+1, :)$ and locating the work unit consisting of $\{x(a+1, i_o), x(a+1, i_1), \cdots, x(a+1, i_n)\}$ as its next work unit. The procedure repeats until this thread finishes all its workload. The similar procedure applies to column FFTs and the bit-reversal permutation. After using the fine-grained work unit and this major reversal work distribution scheme, the performance reaches 19.37Gflops.

4.2.4 Memory Hierarchy Aware Compilation

We apply the updated compiler, with the memory hierarchy aware instruction scheduling, to the 2D FFT implementation, which introduces another 3.25% improvement over the previous compilation, thus the overall performance raises to

20.00Gflops. Finally, similar to our 1D FFT implementation, the optimized 2D FFT implementation also scales nearly linearly up to 128 threads, as shown in Figure 4.7.

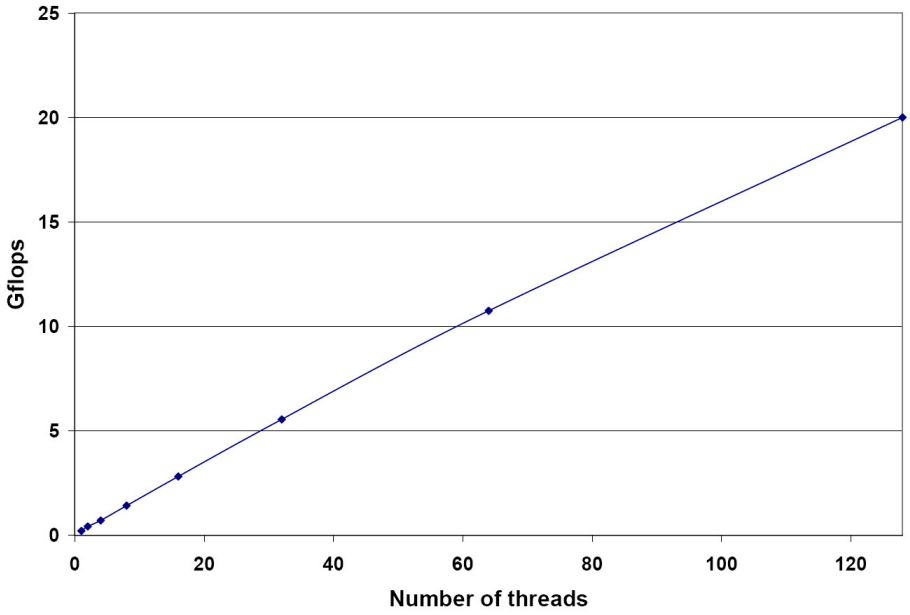


Figure 4.7: Performance of the Optimized 2D FFT Implementation

4.3 Related Work

The FFT problem has been extensively studied on various machines. A large number of literature addresses the distributed memory FFT implementations on the hypercube architecture [58, 84, 123] by taking advantage of the small communication delay between processors that are physically close in the network. Other parallel FFT implementations have been investigated on arrays [85] and mesh architectures [130].

There is also a large body of literature concerned with shared-memory FFTs. The communication pipelining technique was proposed for solving the FFT on the

Connection machine [137]. Two different scheduling strategies for single-vector FFTs and three different approaches to the multiple FFTs are discussed in [23]. The authors also develop models of performance that are consistent with their experiments on the Denelcor HEP. Additional performance studies on shared-memory FFT are discussed in [12, 105, 153]. Moreover, by using the Kronecker notation, the work in [83] shows how to design parallel DFT algorithms with various architecture constraints. The significance of considering memory hierarchy to an effective FFT implementation has been pursued in [13]. The work in [27] shows how to use local memory to compute the FFT efficiently on CRAY-2. The issue of data re-use is also discussed in [5, 10]. Further, an excellent review of various sequential and parallel DFT algorithms proposed in the literature until 1991 appeared in [94]. Two dataflow-based multithreaded FFTs [144] are presented to exploit the features of EARTH [143], a fine-grain dataflow architecture. Performance evaluation and analysis of several scientific computing kernels, including FFT, on the IBM Cell architecture [78], a heterogeneous multi-core architecture, are reported in [155]. Results demonstrate the tremendous potential of the Cell architecture for scientific computations in terms of both raw performance and power efficiency. Lately, general-purpose computing on graphics processing units (GPGPU) is becoming popular because of the high peak performance. As a result, several work of FFT on GPUs have been reported [63, 68, 99, 106, 111].

FFTW [57] is a library for computing the DFT. For small DFTs, it calls special code modules, called *codelets*. These are pre-generated and highly optimized using standard and DFT specific optimization techniques [56]. For large DFTs, it use a dynamic programming approach to determine the best execution *plan* to break down into codelets. It supports multithreaded programming interface, and is portable and adaptable on various SMP architectures with a cache-based memory hierarchy.

SPIRAL [119] is a program generator for linear transforms such as the DFT. It automates the implementation task from problem specification to program. Its approach is to use a specialized signal processing language and a code generator with a feedback loop, which allows the systematic exploring of possible choices of formals and code implementations to choose the best combination. Recently, SPIRAL has presented an approach to automatic generation of parallel FFT code for SMP and multi-core architectures [53]. The generated parallel FFT codes using OpenMP as well as Pthreads are evaluated on several CMP/SMP architectures.

4.4 Summary

In this chapter, we presented the implementation and optimizations of the FFT on the C64 many-core architecture, together with extensive analysis. The results demonstrate that many-core architectures like C64 can be used to achieve excellent performance results with respect to both speedup and absolute performance for DSP problems like FFT. For instance, the best result of the FFT obtained on a 3.60GHz Intel Xeon Pentium 4 processor is 5.5Gflops [49], which is only around one quarter of the performance received by using one C64 chip.

The study also shows that application development on such many-core architectures is not easy. We should carefully consider both architecture features and properties of the application/algorithm itself to achieve the best performance. Almost all optimizations applied in our work involve problem-specific features that can be matched to certain architecture features, such as register file size with work units, using fast barrier operations, and so on.

Moreover, the study shows that many-core system software, especially the compiler, faces more challenges. In the study we shows that memory hierarchy aware instruction scheduling may dramatically improve the performance while reducing the burden on the programmers.

The overall contributions in this chapter, together with other software development experiences on C64 [77, 140, 150], clearly highlight the benefits and advantages of employing many-core architectures and serves as a basis for future research.

Chapter 5

PERFORMANCE MODEL OF FFT ALGORITHMS

While many-core architectures are becoming increasingly attractive platforms for high performance computing, it is difficult for programmers to fully explore their computing capabilities, partly due to a lack of performance modeling that can assist the design of parallel algorithms and direct applications performance tuning. For instance, when designing and tuning FFT algorithms for many-core architectures, programmers may ask the following questions:

- What is the expected performance of an FFT implementation programmed in a high-level language for a many-core architecture?
- How does the performance of a parallel FFT algorithm change with the problem size?
- How scalable is an FFT algorithm, given a problem size?

A performance model that can answer these questions provides valuable insights for designing FFT algorithms on many-core systems, and tuning them to achieve the maximum performance.

In this chapter, we first present an abstract many-core architecture that captures generic features and parameters of a specific class of many-core architectures. We then propose a performance model that estimates the performance of parallel FFT algorithms for this abstract many-core architecture. We instantiate our performance model in the context of the IBM C64 architecture.

5.1 Abstract Many-core Architecture

In this study, we restrict our analysis to a specific class of many-core architectures. We abstract their main architectural features into a generic form as illustrated in Figure 5.1.

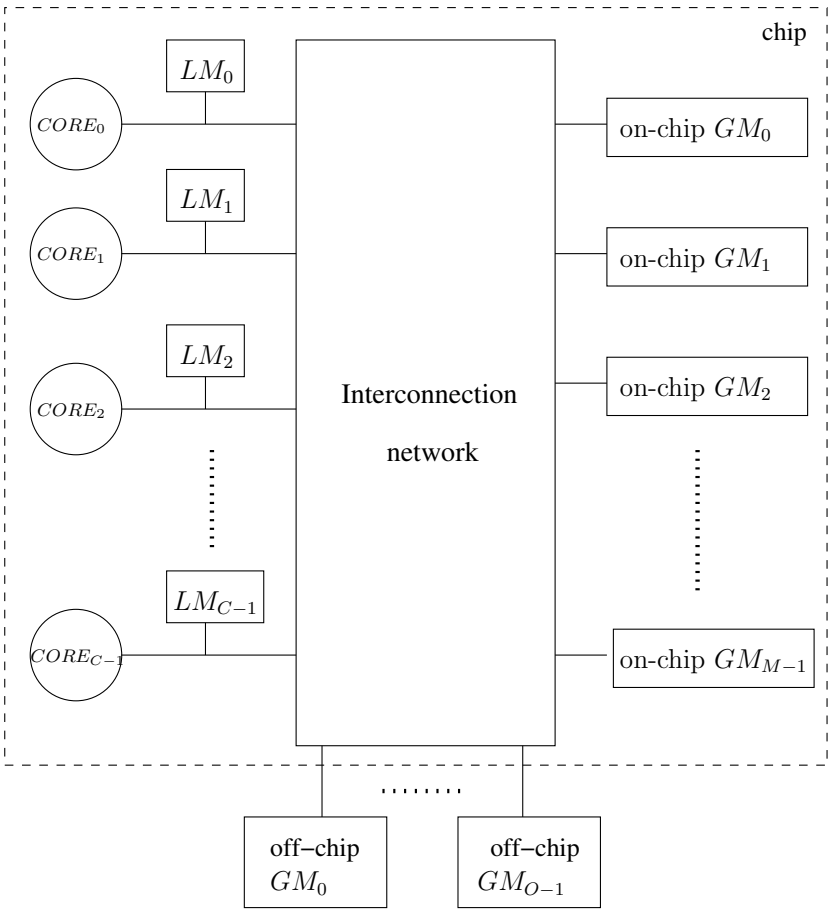


Figure 5.1: An Abstract Many-core Architecture

This abstract many-core architecture consists of a large number of identical cores/processors, each of whom has one or more processing elements (PEs), and a three-layer memory hierarchy, i.e., the local memory (LM), the on-chip global

memory (on-chip GM), and the off-chip global memory (off-chip GM). The on-chip GMs and off-chip GMs are interleaved to achieve higher memory bandwidth. The memory hierarchy is explicitly software addressable to all cores. An on-chip interconnection network connects cores/processors to global memories. All PEs can access on-chip GMs and off-chip GMs via the network. An LM may or may not be globally accessed by all PEs, however, its associated core/PE can access it through some "back-door" with very low latency. We simplify the interconnection network by assuming that the unloaded latency [38] of global memory accesses, either on on-chip GMs or on off-chip GMs, is equal, regardless of the origin or the destination of the access. Instances of such architecture include IBM C64, ClearSpeed CSX700 [34], etc.

To better understand the performance issues, we characterize this abstract architecture with a set of major architectural parameters, which are summarized in Table 5.1. These parameters and their denotations will be used in our following discussion. While there exist more general parallel machine models in the literature with fewer parameters, like LogP [37] and BSP [147], our abstract architecture (and the corresponding parameters) is developed for a specific class of state-of-the-art many-core architectures. Therefore it involves some low-level details, and we do not claim that this abstract architecture/analysis can be immediately applied to a large diversity of parallel architectures.

5.2 FFT Algorithms

In this section, we first briefly review the sequential one-dimensional radix-2 Decimation-In-Time (DIT) Cooley-Tukey FFT algorithm, and then present its parallel counterpart and a parallel radix-4 algorithm. The performance model presented in Section 5.3 are derived for the parallel FFT algorithms presented in this section.

Table 5.1: Parameters of the Abstract Many-core Architecture

C	number of cores in one chip
P	number of PEs in one core
M	number of on-chip memory modules
O	number of off-chip memory modules
B_{in}	bandwidth of a inbound link between a core and the network, measured in bytes per cycle.
B_{out}	bandwidth of a outbound link between a core and the network, measured in bytes per cycle.
L_{LM}	latency for the one-way local memory access.
B_{net}	network bandwidth of the on-chip Interconnection network, measured in bytes per cycle.
B	bandwidth of a single memory module, measured in bytes per cycle.
W	granularity of the interleaved memory system, measured in bytes

5.2.1 Sequential FFT Algorithm

The Cooley-Tukey radix-2 DIT FFT algorithm is one of the most common FFT algorithms. It takes a divide and conquer method that recursively breaks down an N-point DFT into two $N/2$-point DFTs. The time complexity of such algorithm is $\Theta(N \log_2 N)$. More specifically, an N-point FFT, where $N = 2^t$, proceeds in t stages, each of whom is composed of 2^{t-1} *butterfly* operations. The stride between two input data a and b in a butterfly is 2^{i-1}, where i is the stage where the butterfly occurs, $1 \leq i \leq t$. 2^{i-1} different twiddle factors are used in the i-th stage.

Figure 5.2 outlines Algorithm SEQ-R2-FFT, a sequential radix-2 DIT Cooley-Tukey algorithm.

The input data and the pre-computed *long weight vector* [14], which is a stacking of twiddle factors used from the first stage to the last stage, are stored in array x and array ω, respectively. Note that a *bit-reversal permutation* needs to be performed on input x before the butterfly computation stages. Such permutation is not shown in Figure 5.2, and the input x has already been reordered. This is true in our following discussion, and we will not make any further clarification.

```
Algorithm SEQ-R2-FFT

1.      n ← 2^{t-1}
2.      for p ← 1 to t do
3.        l ← 2^p
4.        s ← 2^{p-1}
5.        for i ← 0 to n − 1 do
6.          k ← i/s
7.          j ← i mod s
8.          τ ← ω[s − 1 + j] ∗ x[kl + j + s]
9.          x[kl + j + s] ← x[kl + j] − τ
10.         x[kl + j] ← x[kl + j] + τ
11.       endfor
12.     endfor
```

Figure 5.2: Sequential Radix-2 DIT Cooley-Tukey FFT Algorithm

In Figure 5.2, the outer **for** loop (line 2) iterates all stages, and the inner **for** loop (line 5) calculates butterfly operations in each stage iteratively. For a N-point SEQ-R2-FFT, $N = 2^t$, we denote the data points accessed in the i-th butterfly, $1 \leq i \leq N/2$, in the p-th stage, $1 \leq p \leq t$, as $M(p, i)$. An interesting memory access pattern of SEQ-R2-FFT is captured in the following observation:

Observation 5.2.1 *In the p-th stage of SEQ-R2-FFT, data points accessed from the* $a \cdot (2^c)$*-th* [1] *iteration to the* $((a+1) \cdot 2^c - 1)$*-th iteration, i.e.,* $\bigcup_{k=a\cdot 2^c}^{(a+1)\cdot 2^c-1} M(p, i)$*, constitute either one continuous data region, or two separate continuous data regions with equal lengths, where c is an integer between* 0 *and* $\log_2 N/2$*, and a is another integer between* 0 *and* $(N/2c) - 1$*.*

For example, let $a = 1$, and $c = 1$. These values imply the iterations from iteration 2 to iteration 3. In stage 2, data points accessed during this range are $\{x(4), x(6), x(5), x(7)\}$, which constitute a continuous data region from $x(4)$ to $x(7)$. In stage 3, data points accessed from iteration 2 to iteration 3

[1] *We regard* $2^0 = 0$ *here.*

are $\{x(2), x(6), x(3), x(7)\}$, which establish two separate continuous data regions $(x(2), x(3))$ and $(x(6), x(7))$ with length 2. Detailed discussion of this observation and its application to our performance modeling can be found in Section 5.3.3.

5.2.2 Parallel FFT Algorithms

In the above sequential algorithm, read-after-write data dependence exists between stages, while butterfly operations in one stage are mutually independent. This dependence relation reminds us a straightforward parallel algorithm that concurrently executes butterflies within each stage and synchronizes two executive stages using a global barrier. We assume that the total number of PEs available in the abstract architecture model is far less than the input problem size N, then butterfly operations in one stage can be almost evenly distributed among PEs in a cyclic mode. This parallel algorithm PAR-R2-FFT, executed by PE_e, $0 \leq e < P \cdot C$, is given in Figure 5.3. Similarly, utilizing the same parallelization scheme, a parallel radix-4 Cooley-Tukey algorithm, PAR-R4-FFT, is presented in Figure 5.4. In this algorithm, ω is a $\frac{N-1}{3}$ by 3 array, which is a stacking of twiddle factors and their squares and cubes used from the first stage to the last stage [94].

Note that both Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT are straightforward parallel version of their sequential counterparts, and are not optimized for any specific architecture. While highly optimized algorithms could achieve very good performance, as we demonstrated in Chapter 4, simple algorithms are beneficial to the illustration of our performance model.

5.3 Performance Estimation Strategy

In this section, we first introduce assumptions that are used throughout the chapter. We then present the strategy to estimate the performance of parallel algorithms proposed in Section 5.2.2.

Algorithm PAR-R2-FFT

```
1.    n ← 2^(t−1)
2.    for p ← 1 to t do
3.      l ← 2^p
4.      s ← 2^(p−1)
5.      for i ← e to n − 1 step P · C do
6.        k ← i/s
7.        j ← i  (mod s)
8.        τ ← ω[s − 1 + j] · x[kl + j + s]
9.        x[kl + j + s] ← x[kl + j] − τ
10.       x[kl + j] ← x[kl + j] + τ
11.     endfor
12.     barrier
13.   endfor
```

Figure 5.3: Parallel Radix-2 DIT Cooley-Tukey Algorithm

5.3.1 Assumptions

In order to simplify the modeling, we take the following assumptions.

- We assume that each core and memory bank has an infinite incoming buffer and an infinite outgoing buffer out of the network interface. Therefore, no request/response packet will be dropped.
- We assume that every PE in a core participates in the FFT computation.
- We assume that the problem size N is much larger than the total number of PEs participating in the computation, i.e., $N \gg P \cdot C$, which is often true for most scientific applications.
- We do not consider the cost associated with the bit-reversal permutation, because it is not directly related to the cost of butterfly operations. Our model can be easily extended to incorporate this cost, though.

Algorithm PAR-R4-FFT

```
1.    n ← 4^{t-1}
2.    for p ← 1 to t do
3.      l ← 4^p
4.      s ← 4^{p-1}
5.      ν ← (s − 1)/3
6.      for i ← e to n − 1 step P · C do
7.        k ← i/s
8.        j ← i  (mod s)
9.        α ← x[kl + j]
10.       β ← ω[v + j, 0] · x[kl + s + j]
11.       γ ← ω[v + j, 1] · x[kl + 2s + j]
12.       δ ← ω[v + j, 2] · x[kl + 3s + j]
13.       τ_0 ← α + γ
14.       τ_1 ← α − γ
15.       τ_2 ← β + δ
16.       τ_3 ← β − δ
17.       x[kl + j] ← τ_0 + τ_2
18.       x[kl + s + j] ← τ_1 − iτ_3
19.       x[kl + 2s + j] ← τ_0 − τ_2
20.       x[kl + 3s + j] ← τ_1 + iτ_3
21.     endfor
22.     barrier
23.   endfor
```

Figure 5.4: Parallel Radix-4 DIT Cooley-Tukey Algorithm

- While the system noise may have significant impact on the performance [16, 17, 115], this issue is out of the scope of this study, and we ignore it in our modeling.

- If not explicitly stated otherwise, we assume that thread private data resides in local memories, and shared data, e.g, x, and ω reside in global memories.

- To further simplify the analysis, we assume that all architectural parameters are even numbers. In particular, N, P, C and W are powers of two.

Due to the above assumptions, we do not claim that our performance model can predict the accurate execution time of an application; rather, we attempt to use this model to quantitatively evaluate the performance impact (trend) of algorithms and architectural features on many-core systems.

5.3.2 Basic Strategy

Both Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT described in Section 5.2.2 have an iterative structure. More specifically, some synchronization-free computation pattern is repeated in every stage, and a global barrier is enforced after each stage to guarantee that all operations in that stage have completed. For example, an N-point PAR-R2-FFT proceeds in $\log_2 N$ stages, each of which composed of a set of independent butterfly operations evenly distributed among PEs. Each butterfly operation starts from loading two input data points and a pre-calculated twiddle factor from the global memory, followed by a computation kernel, and finally ends with storing two output points back to memory, as shown in Figure 3.1, The execution time of such N-point PAR-R2-FFT can therefore be calculated by

$$T_{FFT} = \sum_{r=1}^{log_2 N} (max(T_C(r,p) + T_M(r,p) + T_B(r,p))), 0 \leq p < P \cdot C \tag{5.1}$$

where $T_C(r,p)$, $T_M(r,p)$ and $T_B(r,p)$ denote the computation time, memory access time and synchronization time, respectively, of PE_p in stage r. The execution time of a N-point PAR-R4-FFT can be obtained in a very similar way, except that the algorithm proceeds in $\log_4 N$ stages, and each butterfly operation works on a 4-point input dataset. For the sake of brevity, we focus our analyses on Algorithm PAR-R2-FFT, and only show the difference when necessary.

In our abstract architecture, all PEs are identical. Since butterfly operations in one stage are evenly distributed among PEs, and every butterfly takes the same

amount of computation time, we can regard of $T_C(r,p)$ being same for all r and p. We let T_C denote the total computation time, i.e., the computation time for $N/2$ butterfly operations, in each stage.

Similarly, $T_B(r,p)$ can be regarded as same for all r and p, since the semantics of a global barrier requires that all PEs wait at the barrier before any of them is allowed to proceed. We let T_B denote the synchronization time immediately after each stage.

Equation (5.1) can therefore be simplified as

$$T_{FFT} = \sum_{r=1}^{\log_2 N} max(T_M(r,p)) + \log_2 N \cdot T_C + \log_2 N \cdot T_B, 0 \leq p < P \cdot C \qquad (5.2)$$

Using $T_M(r)$ as the short for $max(T_M(r,p))$, we can rewrite Equation 5.2 as

$$T_{FFT} = \sum_{r=1}^{\log_2 N} T_M(r) + \log_2 N \cdot T_C + \log_2 N \cdot T_B, 0 \leq p < P \cdot C \qquad (5.3)$$

By deriving cost functions for $T_M(r)$, T_C and T_B, we can quantitatively estimate the performance of Algorithm PAR-R2-FFT and PAR-R4-FFT on the abstract architecture model.

5.3.3 Estimated Memory Latency

We now derive the cost function for memory access delay for Algorithm PAR-R2-FFT.

Estimated Local Memory Latency

As described in Section 5.1, each PE can access its associated LM through some exclusive "back-door", without going through the network. Hence, we can simply treat the latency of accessing a PE's associated LM as a constant.

Estimated Global Memory Latency

In our abstract architecture model, the memory access delay of load/store operations issued to GMs is determined by the unloaded latency [38] and contention delays. The unloaded latency is the transmission time under ideal conditions. It is determined by the system design and is fixed for a given architecture. The contention delay occurs when multiple requests compete for some hardware resource. There are four types of contention delays in our abstract architecture model: (1) the *outbound* delay, when multiple PEs from the same core compete for a shared channel to inject memory access requests to the network, (2) the *network* delay, when multiple memory accesses compete for the network transmission, (3) the *memory contention* delay, when memory access requests are waiting to be handled by a memory bank, and (4) the *inbound* delay, when multiple data elements are loaded to the same core (for memory loads only). Each type of contention delays can be roughly calculated by dividing the size of the request (in bytes) by the average service rate (in bytes/cycle).

In Algorithm PAR-R2-FFT, the longest memory access delay occurs when multiple memory loads/stores are issued to GMs in a burst. Let T_{ld} and T_{st} denote the time (in cycles) to complete a burst of $P \cdot C$ load and store requests, one from each PE, respectively. T_{ld} and T_{st} can be represented as

$$\begin{aligned} T_{ld} &= \underbrace{\frac{P \cdot S_r}{B_{out}} - 1}_{outbound} + \underbrace{\frac{P \cdot C \cdot S_r}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot C \cdot S_d}{B_m} - 1}_{memory} \\ &+ \underbrace{\frac{P \cdot C \cdot S_d}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot S_d}{B_{in}} - 1}_{inbound} \end{aligned} \tag{5.4}$$

$$T_{st} = \underbrace{\frac{P \cdot S_d}{B_{out}} - 1}_{outbound} + \underbrace{\frac{P \cdot C \cdot S_d}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot C \cdot S_d}{B_m} - 1}_{memory} \tag{5.5}$$

In the above equations, S_r and S_d are the size (in bytes) of a single request/response, with or without containing the data of x or ω, respectively[2]. Then $P \cdot S_r$ is the total size of load requests issued by a single core, and $P \cdot S_d$ is the total size of the store requests issued by a single core, or the total size of the response delivered back to a single core. Similarly, $P \cdot C \cdot S_d$ is the total size of data to be served for a memory access burst. B_m, the *aggregate effective memory bandwidth*, denotes the real achievable memory bandwidth (in bytes/cycle) when a burst of memory accesses are handled. Note that a memory load travels the network twice for sending the request and receiving the response, while a memory store travels the network only once.

Due to varieties of interconnection networks in topology, routing algorithms, switching strategy, and flow control mechanism, it is hard to induce a general equation for network delay, hence we focus on a type of interconnection network - crossbar switch - in this study. The methodology presented here can be easily extended to other types of networks.

A crossbar switch is one form of the multistage networks that allows any input port to communicate with any output port in one pass through the network [73]. One important property of the crossbar switch is its *non-blocking connectivity* within the switch, which allows concurrent connections between multiple input-output pairs with a constant transmission time per packet, provided that inputs/outputs are always available during the connections [72]. For a many-core architecture employing a crossbar switch as the interconnection network, the components annotated with "network" in Equation (5.4) and (5.5) are constants. Furthermore, under our assumption that every core and memory bank has infinite incoming and outgoing buffers out of the network interfaces, B_{in} and B_{out} are considered as constants too.

[2] To simplify the expression, we assume that a load response and a store request are of an equal size. In the actual hardware, they may have different sizes. This fact does not affect our method presented here.

In this study we focus on the cases where x and ω are residing in the on-chip GM. Off-chip memory accesses usually involve more complicated hardware behaviors through the datapath, and the corresponding analysis will be a natural extension of the method presented in this study.

For a many-core architecture employing a crossbar switch as the interconnection network, B_m is an accumulated bandwidth of accessed memory banks. It is worth to note that the value of B_m may be different for memory operations performed on x and ω, since, in a given stage, different PEs always access distinct data elements in x, while they probably attempt to load the same twiddle factor from ω, especially in the first several butterfly stages. When multiple PEs access the same data, they introduce more contention in memory banks. This implies that different contention delay may occur when accessing x and ω through the execution. To clarify this point, we denote $B_{m\_x}(r)$ as the aggregate effective bandwidth for loading/storing x during stage r, and denote $B_{m\_\omega}(r)$ as the aggregate effective bandwidth for loading ω during stage r. To determine the exact value of $B_{m\_x}(r)$ and $B_{m\_\omega}(r)$, we assume, without any loss of generality, that x and ω are aligned to a memory bank boundary.

In our analysis, we consider that a PE can issue load/store requests in a pipelined way, i.e., one request per machine cycle, which is true for modern architectures. A single radix-2 butterfly operation contains 3 load requests (2 for x and 1 for ω), and 2 store requests (for x). The completion time of the pipelined requests is determined by when the last request is finished, i.e., the longest delay. For a burst of radix-2 butterfly operations, we have the following equations to compute the latency T_{ld_p} and T_{st_p}, for a pipelined load and store, respectively,

$$T_{ld\_p}(B_{m\_x}, B_{m\_\omega}) = \underbrace{\frac{3P \cdot S_r}{B_{out}} - 1}_{outbound} + 2D + \underbrace{\frac{3P \cdot S_d}{B_{in}} - 1}_{inbound}$$

$$+ \quad \underbrace{max(\frac{2P \cdot C \cdot S_d}{B_{m\_x}} - 1, \frac{P \cdot C \cdot S_d}{B_{m\_\omega}} - 1)}_{memory} \tag{5.6}$$

$$T_{st\_p}(B_{m\_x}) \quad = \quad \underbrace{\frac{2P \cdot S_d}{B_{out}} - 1}_{outbound} + D + \underbrace{\frac{2P \cdot C \cdot S_d}{B_{m\_x}} - 1}_{memory} \tag{5.7}$$

where the constant D is the one way transmission latency of the crossbar switch.

Determining $B_{m\_x}(r)$

Since $B_{m\_x}(r)$ is an accumulated bandwidth of all accessed memory banks, the key issue is to determine the number of memory banks accessed in a burst of butterfly operations. Recall that Observation 5.2.1 in Section 5.2 states that data points accessed from iteration $a \cdot 2^c$ to iteration $(a+1)2^c - 1$ in one stage constitute either one continuous region or two separate continuous regions on x. This reminds us to transform the problem of calculating $B_{m\_x}(r)$ into determining the number of memory banks that are "covered" by those continuous region(s).

We first discuss the case where all accessed elements in x during a burst constitute one continuous region, which happens during the first $\log_2(2P \cdot C)$ stages. Denote the length of such continuous region as L_1, where $L_1 = 2P \cdot C \cdot S_d$, and such region spans $\lceil \frac{L_1}{W} \rceil$ memory banks. We then have

$$B_{m\_x}(r) = min(M, \lceil \frac{L_1}{W} \rceil) \cdot B \tag{5.8}$$

Next we discuss the case where all accessed elements in x during a burst constitute two separate continuous data regions with equal lengths, which happens in stage r, where $\log_2(2P \cdot C) < r \leq \log_2 N$. Denote the length of such regions as L_2, and denote the distance between two regions (i.e., the distance from the start of the first region to the start of the second region) as Z, where $L_2 = P \cdot C \cdot S_d$, and $Z = 2^{r-1} \cdot S_d$. Since both regions are aligned to the memory bank boundary, the

number of memory banks on which each region spans is $\lceil \frac{L_2}{W} \rceil$, and the number of memory banks "covered" by the distance Z is $\lceil \frac{Z}{W} \rceil$. Note that if Z is long enough, then the second region might "fall off" the end of the last memory bank and "wrap around" to the start of the first memory bank, as illustrated in Figure 5.5(c).

If $\lceil \frac{L_2}{W} \rceil \geq M$, $B_{m\_x}(r)$ is simply $M \cdot B$, since all memory banks will be simultaneously active in serving memory access requests in a burst. Otherwise, we have to investigate the relative positioning of those two regions. Figure 5.5 lists all of three possible scenarios. In the figure, the shaded boxes represent memory regions, and the dotted arrow lines show the wrap-around.

Scenario 1. As shown in Figure 5.5(a), two regions are not overlapping on memory banks. This occurs when $(M \cdot W - 2L_2) \geq (Z \ mod \ (M \cdot W) - L_2) \geq 0$, that is, $(M \cdot W - L_2) \geq Z \ mod \ (M \cdot W) \geq L_2$. In this case, the number of covered memory banks is $\lceil \frac{2L_2}{W} \rceil$, and we have

$$B_{m\_x}(r) = \lceil \frac{2L_2}{W} \rceil \cdot B \tag{5.9}$$

Scenario 2. Figure 5.5(b) shows one kind of overlapping of two regions. When this scenario happens, it satisfies the condition $0 \leq Z \ mod \ (M \cdot W) < L_2$. The number of covered memory banks $Y = \lceil \frac{Z \ mod \ (M \cdot W) + L_2}{W} \rceil$, and we have

$$B_{m\_x}(r) = min(M, Y) \cdot B \tag{5.10}$$

Scenario 3. Another kind of overlapping is shown in Figure 5.5(c). The second region falls off the end of the last memory bank and wraps around to the first memory bank. In this scenario Z satisfies the condition $Z \ mod \ (M \cdot W) \geq M \cdot W - L_2$. The number of covered memory banks is $\lceil \frac{L_2}{W} \rceil + \lceil \frac{M \cdot W - (Z \ mod \ (M \cdot W)}{W} \rceil$, and we have

$$B_{m\_x} = (\lceil \frac{L_2}{W} \rceil + \lceil \frac{M \cdot W - (Z \ mod \ (M \cdot W)}{W} \rceil) \cdot B \tag{5.11}$$

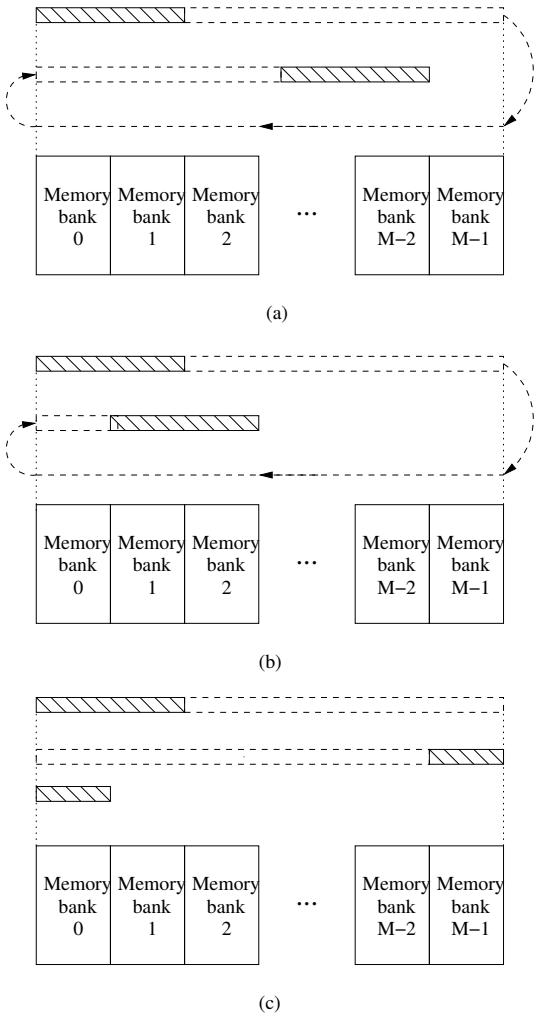


Figure 5.5: Relative Positioning of Two Memory Regions

Determining $B_{m\_\omega}(r)$

Determining $B_{m\_\omega}(r)$ is different from what we have done with $B_{m\_x}(r)$, since the number of distinct twiddle factors accessed in each stage varies through the execution. Here we list two possible scenarios.

Scenario 1. In the case of $P \cdot C \cdot S_d \leq W$, $B_{m\_\omega}(r)$ is always equal to B, because $P \cdot C$ requests of ω always fit into one memory bank[3].

[3] $B_{m\_\omega}(r)$ could be a little bit larger than B, since the requested ω in a burst may reside in two consecutive memory banks; the first bank holds only one twiddle

Scenario 2. When $P \cdot C \cdot S_d > W$, $B_{m\_\omega}(r)$ can be easily determined as B for the first $\log_2 \frac{2W}{S_d}$ stages, because the number of distinct twiddle factors used in each stage does not exceed $\frac{W}{S_d}$. For the rest stages, $B_{m\_\omega}(r)$ is mutually decided by the number of memory banks holding the twiddle factors used in stage r, and the number of requested (distinct) twiddle factors in a burst. This can be generalized as

$$B_{m\_\omega}(r) = min(2^{r-log_2(\frac{2W}{S_d})}, \frac{P \cdot C \cdot S_d}{W}, M)B \tag{5.12}$$

Given Equations (5.6) to (5.12), $T_{m\_b}(r)$, the memory latency for a burst of radix-2 butterfly operations, one for each PE, in stage r, can be estimated by

$$T_{m\_b}(r) = T_{ld\_p}(B_{m\_x}(r), B_{m\_\omega}(r)) + T_{st\_p}(B_{m\_x}(r)) \tag{5.13}$$

Since the workload is evenly distributed to all PEs, and every PE performs $\frac{N}{2P \cdot C}$ identical butterfly operations during each stage, the overall memory latency for Algorithm PAR-R2-FFT can be approximated as

$$\sum_{r=1}^{\log_2 N} T_M(r) = \frac{N}{2P \cdot C} \sum_{r=1}^{\log_2 N} T_{m\_b}(r) \tag{5.14}$$

With slight modifications to Equations (5.6) to (5.14), one can obtain the overall memory latency for Algorithm PAR-R4-FFT. For example, to estimate the latency for a pipelined load for a radix-4 butterfly, instead of having 3 loads for a radix-2 butterfly shown in Equation (5.6), we simply substitute with 7 loads i.e., 4 for x and 3 for ω.

---

factor, and the next one holds at least $(\frac{W}{S_d} - 1)$ twiddle factors. However, the occurrences are few along the computation, and we approximate it as the bandwidth of a single bank.

5.3.4 Estimated Computation Time

To estimate the computation time, we examine the generated instruction sequence of the computation kernel, and use a simplified PE model to approximate the execution time, under ideal conditions: no interference from other PEs, no instruction fetch delays, and perfect branch prediction.

Since memory latency has already been taken care of in Section 5.3.3, all memory instructions are removed from the instruction sequence. The PE model executes the remaining instructions in a pipelined way, i.e., one instruction per cycle. Instructions are executed in-order, such that if one instruction is stalled due to data dependence, no later instruction can be issued. Special care needs to be taken when any shared hardware resource in a core is competed by PEs. Our PE model simply "perfect shuffles" P sets of such instructions into a new sequence, in which all original data dependence relation is preserved, and executes this interleaved sequence. The estimated execution time of this interleaved instruction sequence is used as the execution time of a single set on this PE model. Given this model and the architecture specification, we can express T_C as a function of N, P, and C.

5.3.5 Estimated Barrier Overhead

Given the complexity and variety of barrier implementations, it is difficult to estimate T_B without knowing the details of the real architecture/software. We thus propose an experiment-based approach in our modeling. This approach makes every PE call the barrier function many times, and reports the average elapsed time per call. In this way, we can obtain the cost function of the barrier waiting time as a function of P and C.

5.4 Case Study: IBM Cyclops-64

In this section we evaluate our performance model in the context of the IBM C64 chip architecture. We first instantiate the cost functions the algorithms with

C64 architectural parameters. For simplicity, only the instantiation procedure of Algorithm PAR-R2-FFT is given. Then we evaluate our performance model by comparing the predicted performance with the experimental results.

5.4.1 Instantiation of Cost Functions

The C64 architecture is an instance of the abstract architecture model proposed in Section 5.1. C64 features an explicitly addressable three-level memory hierarchy, including 160 local memories (LMs), one for each PE, 160 on-chip global memories (GMs), and 4 off-chip GMs. Both on-chip GMs and off-chip GMs are interleaved by a 64-byte boundary, and are accessible to all PEs on a chip.

All cores and memory banks are connected to an on-chip pipelined crossbar switch with 96×96 ports. In particularly, 80 ports are shared by 160 on-chip GM units, and 4 ports connected to the off-chip GM controllers. Each port can consume one request packet and send up to 8-byte data to the network/memory in one cycle, while all the other packets waiting in an associated FIFO queue.

As a summary, Table 5.2 lists major architectural parameters of C64. W_{GM} is the granulate of interleaved on-chip GMs. Since two on-chip GMs share one crossbar switch port, it can be approximated that there are 80 on-chip GM banks that are interleaved by a 128-byte boundary.

Table 5.2: Summary of C64 Architectural Parameters

C	80
P	2
M	80
O	4
B_{in}	8 bytes/cycle
B_{out}	8 bytes/cycle
L_{LM}	1 cycle
B_{net}	up to 1140 bytes/cycle
B	8 bytes/cycle
W_{GM}	128 bytes

Cost function for computation time. By examining the instruction sequence generated for Algorithm PAR-R2-FFT on C64, it turns out that a butterfly kernel is implemented by 14 integer instructions and 10 floating-point instructions, besides memory operations. According to the simplified PE execution model proposed in Section 4.4, 14 integer instructions can be completed in 14 cycles, and the interleaved 2 sets of floating-point instructions, one set from each PE, can be finished in 20 cycles. A butterfly kernel then can be accomplished in $14+20 = 34$ cycles, and the cost function for computation time is given by

$$T_C = 34 \cdot \frac{N}{2P \cdot C} = \frac{17N}{2C} \tag{5.15}$$

Cost functions for memory latency. Equation (5.6) and Equation (5.6) in Section 4.5 formulate the pipelined memory load and store latency, respectively. In our experiments both x and ω are double-precision complex numbers stored in on-chip GMs, hence S_r and S_d, the size of a memory access request/response with respect to a single data point, are both 16 bytes. By substituting variables in Equations (5.6) and (5.6) with architectural parameters summarized in Table 5.2, we have

$$T_{ld\_p}(r) = 51 + max(\frac{8C}{B_{m\_x}(r)}, \frac{4C}{B_{m\_\omega}(r)}) \tag{5.16}$$

$$T_{st\_p}(r) = 22 + \frac{8C}{B_{m\_x}(r)} \tag{5.17}$$

Both $B_{m\_x}(r)$ and $B_{m\_\omega}(r)$ can be calculated from Equation (5.8) to (5.12) in the similar way. Due to the space limitation, we simply skip the detailed substitution here.

An investigation on instruction sequence of the butterfly kernel reveals that T_{st_p} can be mostly hidden in computing a butterfly stage because one PE can start executing the next butterfly before the store requests issued in the current butterfly to complete. Therefore, we ignore T_{st_p} in the estimation of the memory

latency of a butterfly operation. Thus the cost function for estimating the overall memory latency is given by

$$\sum_{r=1}^{\log_2 N} T_M(r) = \frac{N}{4C} \sum_{r=1}^{\log_2 N} T_{ld\_p}(r) \tag{5.18}$$

Cost function for barrier synchronization. We have evaluated the average waiting time of the barrier implementation in the C64 TiNy Threads (TNT) library [42], when all PEs enter the barrier simultaneously. Using the technique of curve fitting, the cost function of the barrier waiting time (with respect to the number of cores) is roughly approximated by

$$T_B = 6C + 203 \tag{5.19}$$

5.4.2 Evaluations and Discussions

In this section, we present a set of extensive evaluations of the proposed performance model. We are particularly interested in the following aspects:

1. Estimated execution time on each stage;
2. Performance impact as the input problem size varies;
3. Performance impact as the number of cores varies;
4. Performance impact as the algorithm changes;
5. Performance impact as the architectural parameters change.

We compare our estimations with experimental results obtained from FAST, the C64 simulator [40]. The experimental results show an average relative error of 16%, when running on up to 16 cores. This average relative error increases as more cores are used, and it reaches 29% at 64 cores. It is worth to note that similar results were obtained on a preliminary version of the real C64 chip.

Estimated execution time on each stage. We first want to compare the predicted execution time of each individual computation stage (plus the waiting time of the following barrier) with the experimental result, since its accuracy is the fundamental requirement for our subsequent analysis. Figure 5.6 and Figure 5.7 show such comparison when computing a 2^{10}-point FFT with Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT, respectively. It can be observed that our performance model can predict the time spent on each stage with relative accuracy. Both figures show that the predicted time is $1\% - 29\%$ higher than the experimental execution time when running on up to 16 cores (part of the data are not shown in the figure). This difference is probably caused by our assumption that all instructions in a butterfly calculation must be stalled until all input data points, and the twiddle factors are loaded. In the actual system, however, one instruction can be executed as soon as all its operands are available and all its dependence relation is resolved, hence a long stall expected in our model can be avoided.

It can be also observed that when more cores are used (e.g., up to 64 cores), the predicted time is $5\% - 31\%$ lower than the experimental execution time (part of the data are not shown in the figure). One possible reason for this difference is that the behavior of the crossbar network cannot be accurately captured by the current method under heavy traffic. We expect that this issue could be alleviated by incorporating a more accurate network model into our performance model.

Performance impact as the problem size varies. We investigate how the predicted execution time and performance change as a function of the problem size, when running on varied number of cores. The results of Algorithm PAR-R2-FFT are summarized in Figure 5.8 and Figure 5.9. Both figures demonstrate that our performance model correctly predicts the performance trend as the problem size increases, when compared with the experimental execution time and performance.

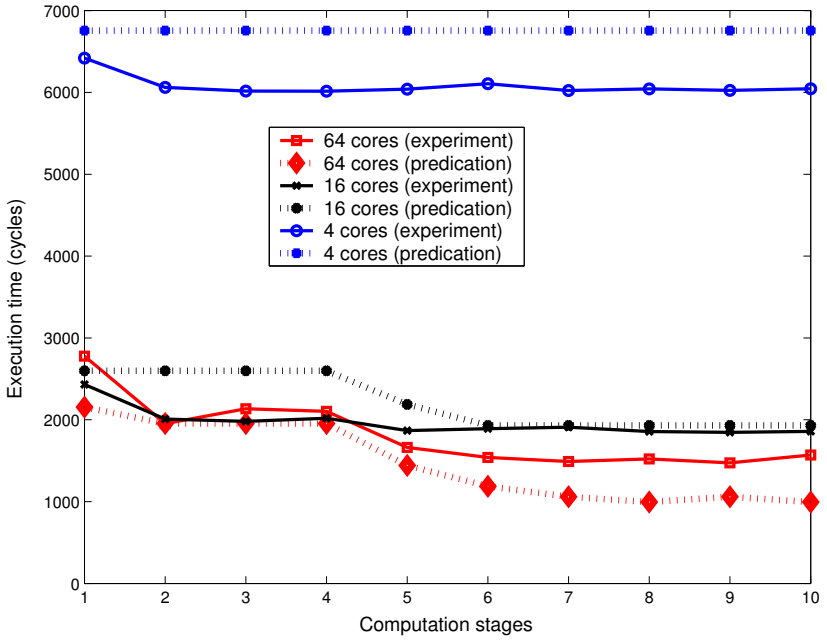


Figure 5.6: Execution Time of Individual Stages, Algo. PAR-R2-FFT

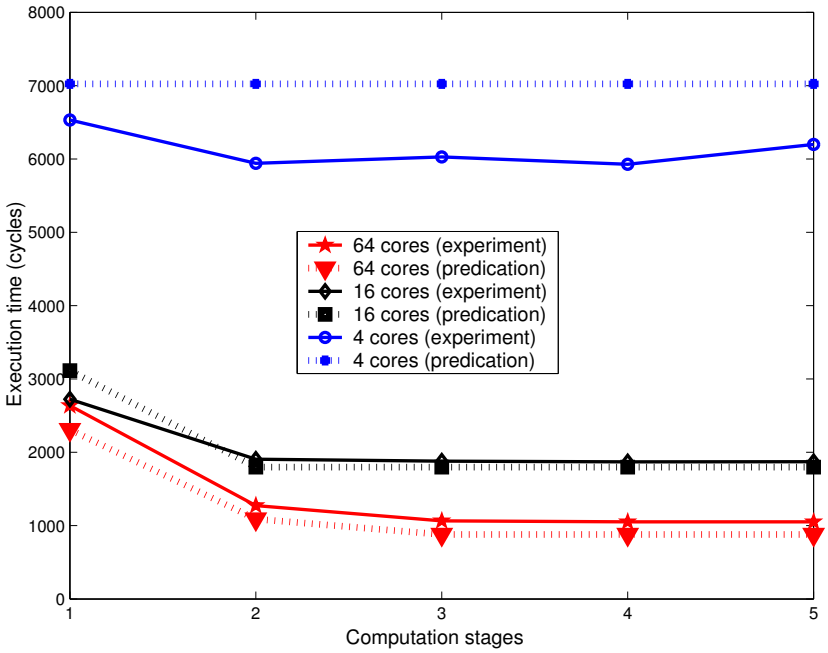


Figure 5.7: Execution Time of Individual Stages, Algo. PAR-R4-FFT

Figure 5.9 shows that, when running on a large number of cores (e.g., 64 cores), the performance increases as the increase of the problem size, while it keeps flat when running on a small number of cores.

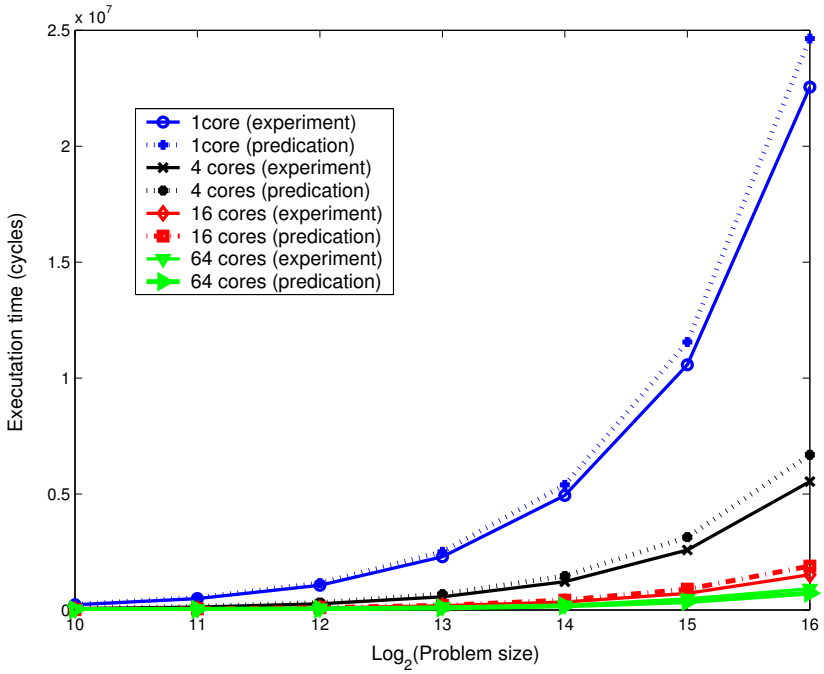


Figure 5.8: Total Execution Time versus the Problem Size, Algo. PAR-R2-FFT

Performance impact as the number of cores varies. We now show how the performance for a fixed input size changes with the number of cores. As shown in Figure 5.10, the estimations closely match the experimental results for all three problem sizes, when running on up to 32 cores. The difference between predicted and simulated performance is becoming rather noticeable, when running on a large number of cores, i.e., up to 29% difference when running on 64 cores. One possible reason is the inaccurate modeling under heavy traffic. Figure 5.11 shows the corresponding speedup curves.

Performance impact as the algorithm changes. From Figure 5.6 we can observe that when running on a large number of cores, the first several stages take

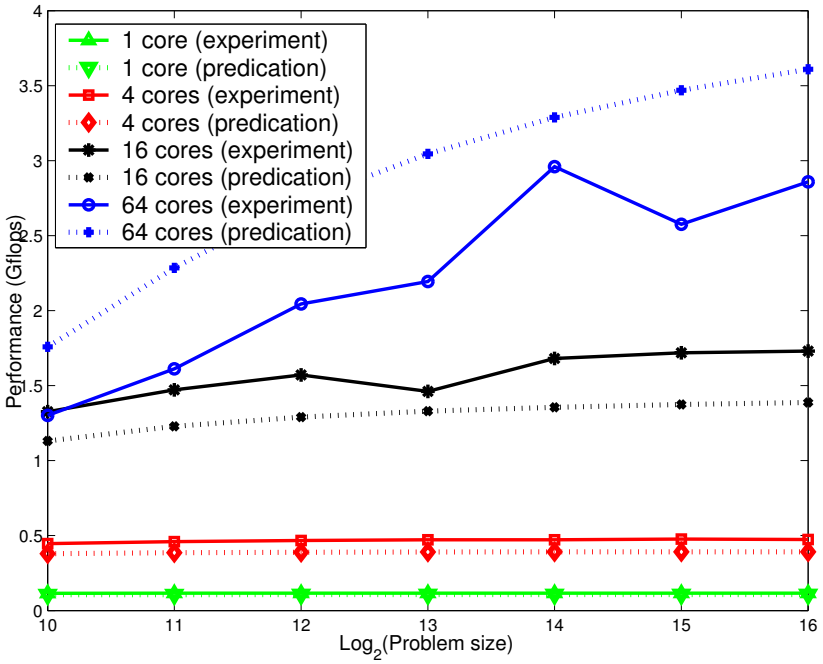


Figure 5.9: Performance versus the Problem Size, Algo. PAR-R2-FFT

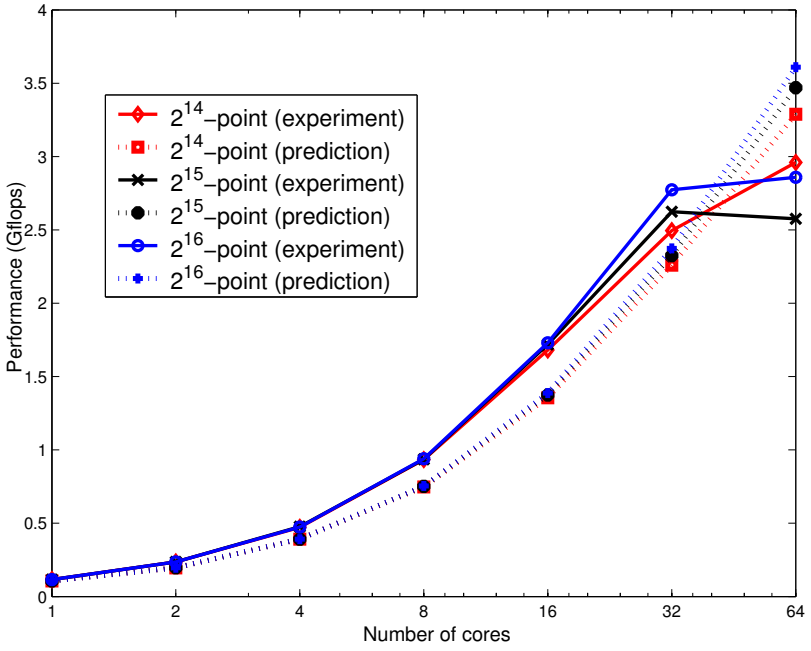


Figure 5.10: Performance versus the Number of Cores, Algo. PAR-R2-FFT

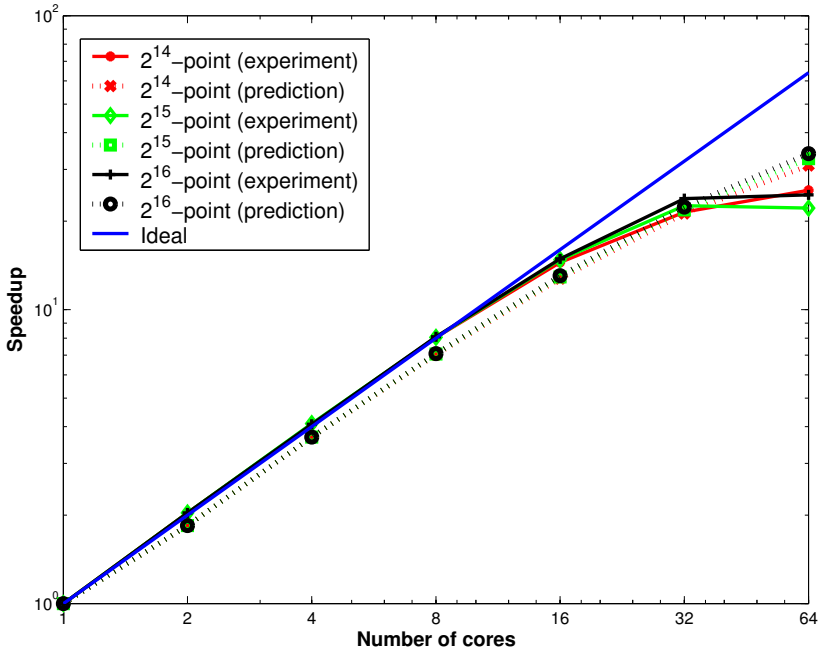


Figure 5.11: Speedup versus the Number of Cores, Algo. PAR-R2-FFT

a much longer time than the rest of stages. A careful investigation into both algorithms indicates that it is probably caused by the contention delay on loading the shared twiddle factors. For example, recall that 2^{i-1} $(1 \leq i \leq log_2 N)$ distinct twiddle factors are used in the i-th stage of Algorithm PAR-R2-FFT. In the first several stages a large number of PEs compete for loading a small number of twiddle factors, resulting in intensive contentions. Based on our performance model, both the accessing latency and the contention in the first stages could be greatly reduced, if each PE keeps a local copy of twiddle factors in its associated LM. We then revised Algorithm PAR-R2-FFT according to this idea. We call this revised algorithm PAR-R2LM-FFT. Due to the limited size of the LM on the C64, in the real implementation, only twiddle factors used in stage 1 to 6 are stored in each PE's associated LM. In the rest of the stages, PEs still have to load the twiddle factors from GMs. The predicted execution time and the experimental execution time of Algorithm PAR-R2LM-FFT for a 2^{10}-point FFT are shown in Figure 5.12. Compared

with Figure 5.6, this new algorithm shows significant performance improvement in the first 6 stages. However, even in the improved algorithm, memory access oper-

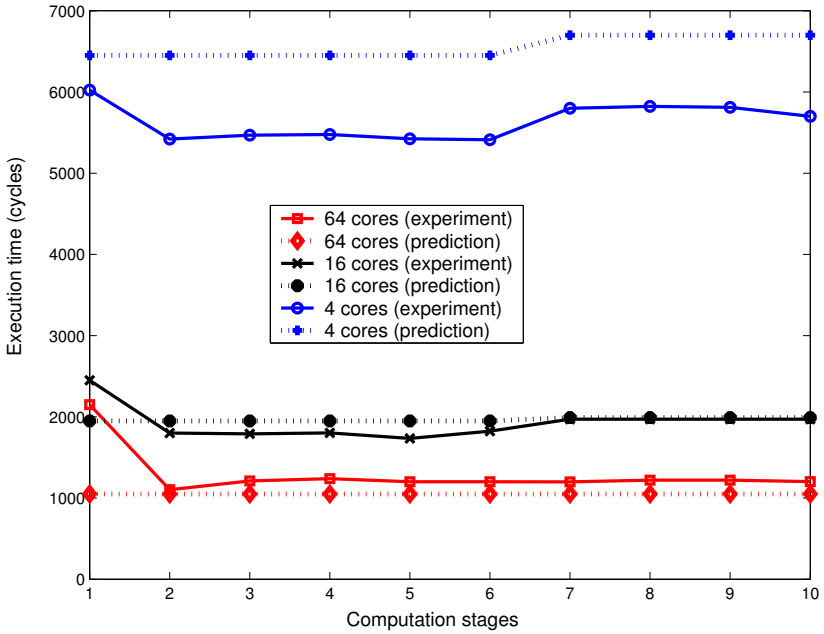


Figure 5.12: Execution Time of Individual Stages, Algo. PAR-R2LM-FFT

ations still cost about $300\% - 500\%$ more time than floating-point operations in a butterfly. This also explains why the achieved performance is far below the theoretic peak performance. One way to improve the performance is to use algorithms concerning data reuse, like higher radix algorithms , which can reduce memory traffic significantly. As shown in Figure 5.13, PAR-R4-FFT doubles the performance for various problem size - system configuration combinations, compared with PAR-R2-FFT. Our performance model shows that up to 140% performance gain could be achieved if a radix-8 FFT algorithm is used, compared with PAR-R2-FFT.

Performance impact as the architectural parameters change. Programmers and architects often want to know the performance impact of architectural changes to the existing algorithms. To this end, we consider a hypothetical many-core machine, C64+, which has the exact same configuration as C64, except that

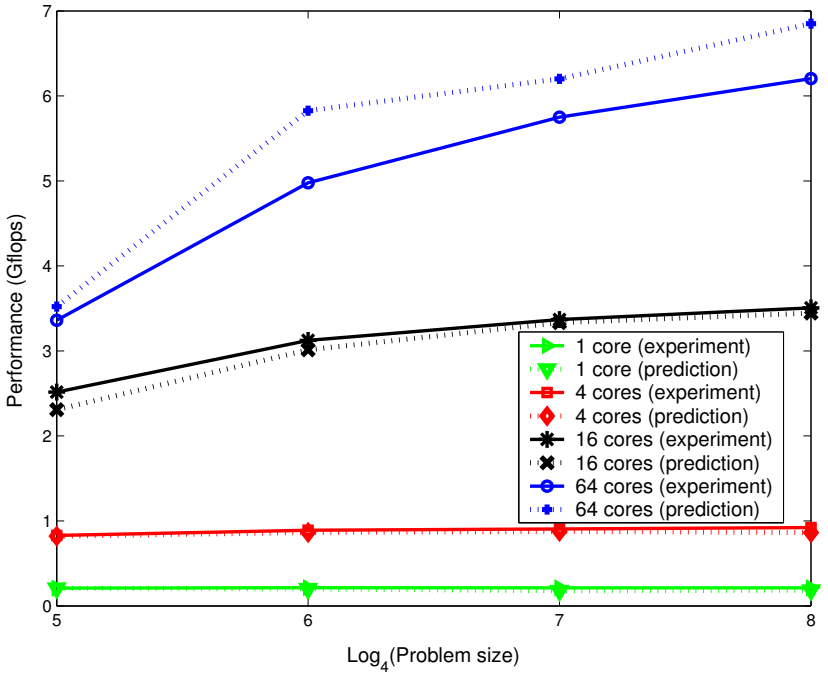


Figure 5.13: Performance versus the Problem Size, Algo. PAR-R4-FFT

each core now has 4 PEs, instead of 2 in the original C64 design. We then apply our performance model with architectural parameters of this C64+ for Algorithm PAR-R2-FFT.

Figure 5.14 shows the predicated performance data for a 2^{10}-point FFT and a 2^{16}-point FFT. From the figure we can observe that adding more PEs to a core does not yield a significant gain of performance for our FFT algorithm. In particular, for the problem size of 2^{10}-point, using more than 16 cores even has a negative performance impact. This is probably due to the increased memory contention delay and the longer barrier waiting time.

5.5 Related Work

A model-driven DFT performance predictor is proposed in [67] to replace the empirical search engine in FFTW. The predictor first models the performance of several frequently used DFT algorithms and DFT codelets (in FFTW), it then

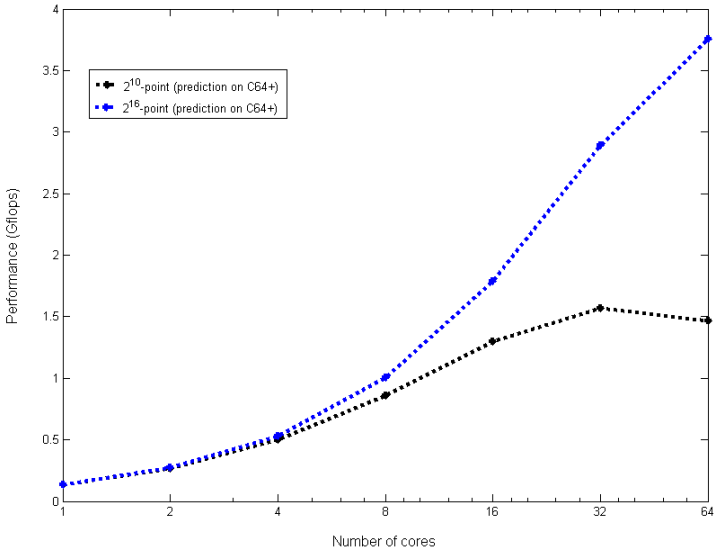


Figure 5.14: Performance Predication for C64+, Algo. PAR-R2-FFT

builds the optimization engine based on those individual DFT models. The parameters for these models are obtained via extensive experiments on the target architecture. The plans produced by the predictor have comparable performance to those plans generated by the empirical search in FFTW. It is not clear how to extend this work to utilize multiple processing elements on many-core architectures. Mathematical model is used in [92] to represent the Cooley-Tukey FFT algorithm as a linear sequence of computational stages. Then, a set of base FFT problems (of different sizes) need to be evaluated on the target architecture to extract the performance features and calibrate the model. This model has been validated on the CELL architecture. Results show that the predicated execution time is very close to the actual measurement. The work reported in [52] shows that a properly built analytical model can successfully find the best DFT algorithms in SPRIAL even the predicated runtime is not accurate. This model incorporates details of the target architecture's memory system, including the TLB, branch prediction, physically addressed caches and hardware prefetching. Unlike these studies, our performance

model does not require running many test cases to adjust the model to fit the actual runtime behaviors of the program. It is obtained only based on the architecture specifications (except the modeling of the barrier overhead).

Performance modeling of FFT is conducted by Cvetanović [153] for an abstract shared memory architecture. The work investigates the impact of the data layout on the memory access latency. Closed-form performance expressions are derived for the best-case and worst-case data layout. This work also approximates that memory operations regarding the input samples are issued by all processors in a burst. Our work differs from this work in several ways. First, while no specific algorithm is studied in [153], we present detailed analyses of two parallel FFT algorithms, together with experimental results on the real system. Secondly, the former study does not consider the memory traffic generated for loading the twiddle factors, and it assumes that the same network contention is produced during each stage, which may not be realistic for all FFT problems. Our work investigates both issues, and take into account their effects upon the execution behaviors.

The technique of using instruction count to estimate the FFT performance is also used in [105], where several FFT algorithms are analyzed for IBM RP3 system. However, the work treats memory and synchronization delays as constants. Since the memory latency may vary due to the different memory access patterns through the execution, this assumption affects the accuracy of the results. Such issue has been explicitly taken into account into our analysis.

5.6 Summary

In this chapter, we have presented a model for performance prediction of parallel Cooley-Tukey FFT algorithms for a many-core architecture. By describing the algorithm and the performance model with respect to an abstract architecture, the analysis techniques presented in this chapter can be applied to other similar systems with minor changes. Due to the regular structure of the FFT problem, we

decomposed the execution of a FFT problem into three parts, i.e., memory accesses, computation, and synchronization. We derived closed-form cost functions for each of the parts. We instantiated the cost functions of the FFT algorithms and the performance model in the context of the IBM Cyclops-64. The experimental results demonstrate the effectiveness of our performance model methodology.

PART II

EXPLORING FINE-GRAINED TASK-BASED EXECUTION ON GRAPHICS PROCESSING UNIT-ENABLED SYSTEMS

Chapter 6

NVIDIA CUDA

CUDA stands for *Compute Unified Device Architecture*, a parallel architecture developed by NVIDIA for general purpose parallel computing. In this chapter we provide a brief introduction of the CUDA architecture and the programming model[1]. More details are available on the CUDA website [108]. In the literature, GPUs and CPUs are usually referred to as the *devices* and the *hosts*, respectively. We follow the same terminology in this dissertation.

6.1 CUDA Architecture

Figure 6.1 shows the hardware model of CUDA. CUDA devices have one or multiple streaming multiprocessors (SMs), each of which consists of one instruction issue unit, eight scalar processor (SP) cores for integer and single-precision floating-point arithmetic operations, two special function units for single-precision floating-point transcendental functions, and on-chip shared memory. For some high-end devices, the SM also has one double-precision floating point unit.

CUDA architecture features both on-chip memory and off-chip memory. The on-chip memory consists of the register file, shared memory, constant cache and texture cache. The off-chip memory consists of the local memory and the global memory. Local memory is mainly used to store automatic variables in the GPU program. Shared memory has 16 banks that are organized such that successive

[1] We focus on the CUDA architecture of compute capability 1.x in this chapter.

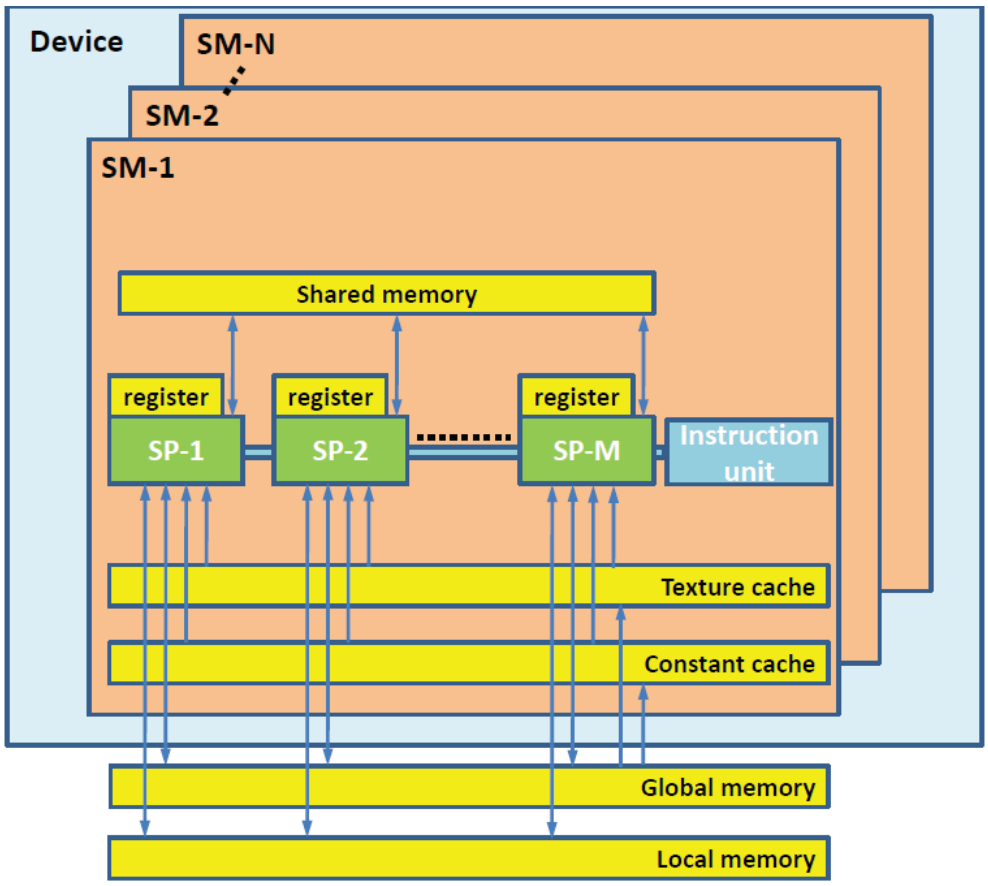


Figure 6.1: CUDA Hardware Model

32-bit words are assigned to successive banks. Both the constant cache and the texture cache are read-only, and are used to cache data reading from the off-chip memory. Constant cache is shared by all functional units in a SM to speed up reads from the constant memory space, which resides in the off-chip memory. The texture cache is shared by 2 or 3 SMs (based on the device design) to speed up reads from the texture memory space, which resides in the off-chip memory. Since there is no ordering guarantee of memory accesses on CUDA architectures, programmers may need to use memory fence instructions to explicitly enforce the ordering, and thus the correctness of the program. The host can only access the global memory of

the device. On some devices, part of the host memory can be pinned and mapped into the device's memory space, and both the host and the device can access that memory region using normal memory load and store instructions.

A CUDA program consists of two parts. One part is the portions to be executed on the CUDA device, which are called *kernels*; another part is to be executed on the host, which we call the *host process*. The device executes one kernel at a time, while subsequent kernels are queued by the CUDA runtime. When launching a kernel, the host process specifies the thread hierarchy to be used for the kernel execution. This thread hierarchy defines how many threads are required to execute the kernel, and how many thread blocks (TB) these threads should be equally divided into. Each thread that execute the kernel is given a unique thread index within a TB, which is a 3-component vector. So that threads within a TB can form a 1D, 2D, or 3D thread space. This provides an convenient way to invoke computation across the elements in a domain such a vector, matrix, or volume. There is a limit to the number of threads per TB, since all thread of a TB share the limited hardware resource of a same SM. TBs are organized into a 1D or 2D grid. The number of TBs in a grid is usually dictated by the size of data being processed or the number of SMs in the system. Figure 6.2 shows an example of such a grid-block-thread hierarchy.

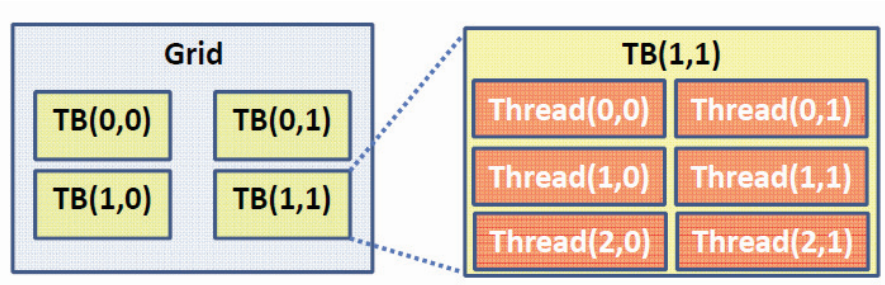


Figure 6.2: Example of CUDA Thread Hierarchy

A thread can obtain its logic thread index within a TB and its logic TB index via built-in system variables. The hardware schedules and distributes TBs to SMs with available execution capacity. One or multiple TBs can reside concurrently on one SM, given sufficient hardware resources, i.e., register file, shared memory, etc. TBs do not migrate during the execution. As they terminate, the hardware launches new TBs on these vacated SMs, if there are still some TBs to be executed for this kernel. Each thread is mapped to one SP core, and is executed independently. Moreover, the SM manages the threads in groups of 32 threads called *warps*, in the sense that all threads in a warp execute one common instruction at a time. Branch divergences occurred within a wrap will serialize the different execution paths, which can significantly degrade the performance.

On the device, the global, constant, and texture memory spaces are persistent through the execution of a kernel. Each thread has its own private local memory. All threads can access the global memory space, but only threads within a TB can access the associated shared memory with very low latency. When accessing the global memory, memory loads and stores by threads of a half warp are coalesced by the device into as few as one transaction when certain access requirements are met. For CUDA programs, it is very important to coalesce global memory accesses to reduce the memory traffic and overall latency.

While CUDA provides a barrier function to synchronize threads within a TB, it does not provide any mechanism for communications across TBs. However, with the availability of the atomic instructions and memory fence functions, it is possible to achieve inter-TB communications.

6.2 Software Toolkit

The CUDA software toolkit contains the tools needed to compile and build a CUDA application. It includes following basic components.

- **CUDA driver API:** A language-independent, lower-level API that handles binary or assembly code. Using this API implies direct dealing with initialization, module management, context management, etc.

- **CUDA C:** A set of extensions to the C language and a runtime library for programming CUDA devices. These extensions allow programmers to define a kernel as a C function and use some new syntax to specify the execution environment each time the function is called. This runtime API is built on top of the CUDA driver API, therefore many operations are performed implicitly and resulting code is simpler than using CUDA driver API.

- **nvcc:** The CUDA compiler driver. It first separates the device functions and the host functions from the source code. It then compiles the device functions using NVIDIA compilers/assemblers, and using the specified (available on the host platform) C/C++ compiler to compile the host functions. After both device functions and host functions have been compiled, it embeds the compiled device functions as load images in the host object file.

- **cuda-gdb:** An extension to the standard i386/AMD64 port of gdb. It is designed to present the user with debugging environment that allows simultaneous debugging of GPU and CPU code.

- **profiler:** A performance analysis tool for collecting runtime information for the execution of a CUDA program, such as CPU time, GPU time, memory throughput, number of divergent branches, etc.

Chapter 7

MOLECULAR DYNAMICS

Molecular dynamics (MD) simulations provide scientists a methodology for studying microscopic world on the molecular scale. Rather than attempting to deduce microscopic behavior directly from theories, the MD method tries to apply the constructive approach to reproduce the behavior using various models and computer simulations. Many scientific and engineering branches use MD as a fundamental method, e.g., physics, chemistry, biochemistry, material science, etc. The continually increasing power of computers makes it possible to pose questions of greater complexity, with a realistic expectation of obtaining meaningful answers in these fields. In this chapter we provide a brief introduction of the MD simulations. We will use the term *atom* and *particle* interchangeably.

7.1 Introduction

MD is a simulation method of computing dynamic particle interactions on the molecular or atomic level. Such simulation consists of the numerical, step-by-step, solution of the classical equations of motion, where the laws of classical mechanics are followed, and most notably Newton's law,

$$m_i \ddot{r}_i = F_i \tag{7.1}$$

for each atom i in a system constituted by N atoms. Here, m_i is the atom mass, $r^N = (r_i, \cdots, r_N)$ is the complete set of $3N$ atomic coordinates, $\ddot{r}_i = a_i$ is its acceleration, and F_i is the force acting upon it, due to the interactions with other

atoms. These interactions are simulated using a distance calculation, followed by a force calculation. Force calculations are usually composed of non-bonding forces and bonded forces. When the net force for each particle has been calculated, new positions and velocities are computed through a series of motion estimation equations. The process of net force calculation and position integration repeats for each time step of the simulation. MD is a deterministic technique. Given the initial input data, e.g., positions, velocities, temperature, etc, the physical quantities of the system for each subsequent time step are in principle completely determined.

7.2 Molecular Interactions

To accurately model the physical system in a MD simulation, the most important issue is to choose the potential: a function $U(r^N)$ of "the positions of the nuclei, representing the potential energy of the system when the atoms are arranged in that specific configuration"[46].

Forces then can be derived as the gradients of the potential with respect to atomic displacements,

$$F_i = -\frac{\partial}{\partial r_i} U(r^N) \tag{7.2}$$

The mechanical molecular model uses springs to simulate molecular bonds and spheres that connect atoms. Then the classic mathematics of spring deformation can be used to model the behaviors of bonds. Non-bonded atoms (greater than two bonds apart) interact through van der Waals attraction [66], and electrostatic force [65]. Therefore, a simple potential function for the system is given by

$$U(r^N) = U_{bonding}(r^N) + U_{non-bonded}(r^N) \tag{7.3}$$

7.2.1 Bonding Potentials

For considering the intra-molecular bonding interactions, the following simple molecular model can be used,

$$
\begin{aligned}
U_{bonding}(r^N) &= 1/2 \sum_{bonds} k^r_{ij}(r_{ij} - r_{eq})^2 && (7.4)\\
&+ 1/2 \sum_{bend\ angles} k^\theta_{ijk}(\theta_{ijk} - \theta_{eq})^2 && (7.5)\\
&+ 1/2 \sum_{torsion\ angles} \sum_{m} k^{\phi,m}_{ijkl}(1 + cos(m\phi_{ijkl} - \gamma_m)) && (7.6)
\end{aligned}
$$

Equation 7.4 estimates the energy associated with vibration about the equilibrium bond length, which typically involves the separation $r_{ij} = |r_i - r_j|$ between adjacent pairs of atoms in a molecular framework. In the equation, k^r_{ij} controls the stiffness of the bond spring, r_{eq} defines its equilibrium length. Unique k^r_{ij} and r_{eq} parameters are assigned to each pair of bonded atoms based on their types.

Equation 7.5 estimates the energy associated with vibration about the equilibrium bond angle. In the equation, k^θ_{ijk} controls the stiffness of the angle spring, while θ_{eq} defines it equilibrium angle. The "bend angles" θ_{ijk} are between successive bond vectors such as $r_i - r_j$ and $r_j - r_k$, and therefore involve three atom coordinates,

$$
\begin{aligned}
cos\theta_{ijk} &= \hat{r}_{ij} \cdot \hat{r}_{jk} && (7.7)\\
&= (r_{ij} \cdot r_{ij})^{-1/2}(r_{jk} \cdot r_{jk})^{-1/2}(r_{ij} \cdot r_j k)
\end{aligned}
$$

where $\hat{r} = \frac{r}{||r||}$ is the unit vector.

In the above model, Equation 7.6 represents the amount of energy that makes the total energy consistent with the real physical world for modelling a torsion angle. In the equation, the "torsion angles" ϕ_{ijkl} are defined in terms of three connected bonds, i.e., four atomic coordinates:

$$
cos\phi_{ijkl} = -\hat{n}_{ijk} \cdot \hat{n}_{jkl} \qquad (7.8)
$$

where $n_{ijk} = r_{ij} \times r_{jk}, n_{jkl} = r_{jk} \times r_{kl}$, and $\hat{n}$ is the unit vector normal to the plane defined by each pair of bonds. $k^{\phi,m}_{ijkl}$ controls the amplitude of the curve, m controls

its periodicity, and γ_m shifts the entire curve along the rotation angle axis ϕ_{ijkl}. Unique parameters for torsional rotation are assigned to each bonded quartet of atoms based on their types.

7.2.2 Non-bonded Interactions

The potential energy $U_{non-bonded}$ represents non-bonded interactions between atoms. It is typically split into 1-body, 2-body, 3-body, . . ., terms:

$$U_{non-bonded}(r^N) = \sum_i u(r_i) + \sum_i \sum_{j>i} u(r_i, r_j) + \cdots \tag{7.9}$$

The $u(r)$ term is an externally applied potential field or the effects of the container walls. The clause $j > i$ in the second summation of the 2-body term is used to make sure each atom pair will only be considered once. Traditionally, three-body and higher order interactions are neglected in the simulation. Also, for fully periodic simulations, $u(r)$ is ignored. Therefore, it is usual for MD simulations to concentrate on the pair potential $u(r_i, r_j) = v(r_{ij})$. Then a simple choice for $U_{non-bonded}$ is to write it as a sum of pairwise interactions,

$$U_{non-bonded}(r^N) = \sum_i \sum_{j>i} u(r_i, r_j) \tag{7.10}$$

The development of accurate potentials represents an important research field. There is an extensive literature on the way these potentials are determined experimentally, or modeled theoretically. In this study, we only concentrate on the most commonly used pair-potentials model, the Lennard-Jones pair potential, and Coulomb potentials if electrostatic charges are present. These two models are usually sufficient for representing the essential physics for many simulations.

Lennard-Jones potential

The Lennard-Jones potential (LJ) [91] is a mathematically simple model that describes the interaction between a pair of neutral atoms or molecules.

The LJ potential is given by the expression

$$u^{LJ}(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \tag{7.11}$$

for the interaction potential between a pair of atoms, where ε is the depth of the potential well, σ is the finite distance at which the inter-particle potential is zero, and r is the distance between the particles.

The r^{-12} term, dominating at short distance, models the repulsion between atoms when they are brought very close to each other. The r^{-6} term, dominating at large distance, models the attraction between atoms, which gives cohesion to the system. The parameters ε and σ are chosen to fit the physical properties of the material.

Coulomb potential

When the electrostatic interactions between electrically charged particles are considered, Coulomb potentials [65] are usually added to the simulation. It is expressed as the following equation,

$$u^{Coulomb}(r) = \frac{Q_1 Q_2}{4\pi\epsilon_0 r} \tag{7.12}$$

where Q_1 and Q_2 are the charges, ϵ_0 is the permittivity of free space, and r is the distance between two charged particles.

Potential truncation and long-range corrections

The potentials in Equation 7.11 and 7.12 have an infinite range. In practical applications, it is customary to establish a *cutoff* radius R_c and disregard the interactions between atoms separated by more than R_c, under the assumption that only atoms which are sufficiently close actually interact with each other. This approach results in enormous savings of computer resources, because the number of atomic pairs separated by a distance r grows as r^2 and becomes quickly huge.

However, if we simply ignore an atom pair when it is beyond the cutoff distance, it would induce a jump in the energy, which may induce errors in energy conservation in a simulation. This problem is usually amended by shifting the potential to make it vanishing at the cutoff radius. For example, the LJ potential with the cutoff radius can be rewritten as following,

$$U^{LJ} = \begin{cases} u^{LJ}(r) - u^{LJ}(R_c), & r \leq R_c \\ 0, & r > R_c \end{cases} \tag{7.13}$$

Commonly used truncation radii for the LJ potential are 2.5σ and 3.2σ.

7.3 Time Integration

Time integration algorithms are the engine of MD simulations. They integrate the equation of motion of the interacting particles and follows their trajectory. In this section, we review one of the most popular integration methods for MD simulations, the Verlet algorithm.

The basic idea of the Verlet algorithm [151] is to write two third-order Taylor expansions for the positions $r(t)$, one forward and one backward in time, as follows,

$$r(t + \triangle t) = r(t) + v(t)\triangle t + \frac{1}{2}a(t)\triangle t^2 + \frac{1}{6}b(t)\triangle t^3 + O(\triangle t^4) \tag{7.14}$$

$$r(t - \triangle t) = r(t) - v(t)\triangle t + \frac{1}{2}a(t)\triangle t^2 - \frac{1}{6}b(t)\triangle t^3 + O(\triangle t^4) \tag{7.15}$$

where t is the current time, $\triangle t$ denotes the size of the timestep used for the numerical integration, v is the velocity, a is the acceleration, and b is the third derivative of r with respect to t.

Adding Equation 7.14 and 7.15, we have the basic form of the Verlet algorithm,

$$r(t + \triangle t) = 2r(t) - r(t - \triangle t) + a(t)\triangle t^2 + O(\triangle t^4) \tag{7.16}$$

Note that $a(t)$ can be obtained with the Newton's laws as follow,

$$a(t) = F/m = -\frac{\partial}{\partial r}U(r^N)/m \tag{7.17}$$

7.4 Related Work

MD simulations are computationally intensive. In practice, these simulations have always been limited by the current available computing power. In general, approaches to speedup these computations can be divided into two categories: 1) those that simplify models and identify what may be neglected while still obtain acceptable results, and 2)those that do not affect the accuracy but seek to grain speed by the software or hardware. Among the most prominent practical approaches in the second category is parallel computation. In the past, commodity PC cluster and grids have been used to provide more accessible high performance power at low cost. Therefore, many general purpose MD codes have been developed for clusters, for example, LAMMPS [118], DLPOLY [133], GROMACS [19], and NAMD [116]. However, this cluster-based approach suffers from scalability issues due to the high latencies for communicating between compute nodes when the number of nodes becomes large. Hence, this approach is only suitable for MD simulations with a large number of separate trajectories with short time scales. On the other hand, special-purpose architectures have been used for the MD simulations as well. Examples include Anton [129], FASTRUN [50], Protein Explorer [139], and MD Engine [146]. In general, arithmetic units have been specifically designed for accelerating certain MD simulations. The problem with this approach is obvious: these architectures usually do not have the flexibility to run a variety of algorithms/applications if they cannot utilize the specially designed hardware features.

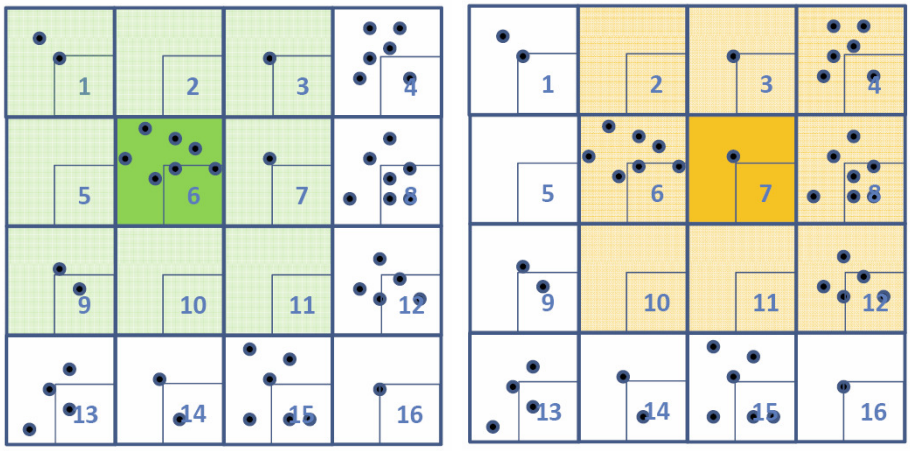
Recently, some studies have successfully tackled the problem of analyzing, evaluating and migrating MD applications to many-core GPUs. These studies demonstrated that kernels of a general purpose molecular dynamic code can run effectively on GPU-enabled systems with performance speedup up to 100-fold over a traditional single-core CPU implementation [7, 18, 45, 117, 121, 135, 148, 158]. Such performance is mainly achieved by carefully choosing optimizations to match

the capabilities of the hardware, including the concurrent code execution on CPU and GPU. Another molecular dynamic application has been accelerated 16 times respect to an optimized implementation on a modern dual-core using the Cell processor [113]. The folding@home client of Stanford [1] achieved a GPU speedup of 100x compared to a single core of a modern processor.

One of the common approaches used to parallelize MD simulations is atom-decomposition [132]. Atom-decomposition assigns the computation of a subgroup of atoms to each processing element (PE). Hereafter we assume that the N atom positions are stored in a linear array, A. We denote P as the number of PEs. A simple atom-decomposition strategy may consist in assigning N/P atoms to each PE. As simulated systems may have non-uniform densities, it is important to create balanced sub-group of atoms with similar number of forces to compute. Non-uniformity is found for instance in gas simulation at molecular level with local variation of temperature and pressure [20]. The computational reason of this load unbalancing is that there is not direct correspondence between the atom position in A and the spatial location in the 3D space. Two common approaches exist in literature to overcome this problem: *randomization* and *chunking*. They are both used in parallel implementations of state-of-the-art biological MD programs such as CHARMM [24] and GROMOS [33]. In randomization, elements in the array A are randomly permuted at the beginning of the simulation, or every certain amount of time steps in the simulation. The array A is then equally partitioned among PEs. In chunking, the array of atoms A is decomposed in more chunks than P, the number of available PEs. Then each PE performs the computation of a chunk and whenever it has finished, it starts the computation of the next unprocessed chunk.

For large scale MD simulations, the system is usually spatially decomposed into small 3D boxes that have a size slightly greater than or equal to the cutoff radius. In this case, for a box in the MD system, the interactions only have to be

evaluated with regard to this box and its 26 neighbor boxes (27 boxes in total). This approach greatly reduce the overall communication cost, however these boxes may contain different numbers of atoms depending on the biological structure simulated. Figure 7.1 shows a 2D example of such decomposition, where the system has been decomposed into 16 boxes. We label these boxes with number $1 - 16$. Each box contains various numbers of atoms. As shown in Figure 7.1(a), the calculation of



(a) Workload for Box-6 (b) Workload for Box-7

Figure 7.1: Example of MD Boxing

the forces for the atoms in box-6 is computed by evaluating the interactions of the atoms in box-6 itself and with the atoms in the non-empty boxes surrounding it, i.e., box-1, -3, -7, and -9. This calculation involves 13 atoms in total. Similarly, as shown in Figure 7.1(b), the calculation of the forces for the atoms in box-7 involves 27 atoms. While the number of atoms in each box is known and so the number of distances to calculate per each box to boxes interaction, the number of force calculations is unknown until the distances are actually evaluated. If the computation of the interactions among these atoms is performed by multiple PEs, the problem of load balancing among PEs is evident: different PEs will receive

different boxes with different atoms. Also note that balancing the workload among PEs using the number of particles in each sub-box to sub-boxes interaction does not solve the problem as this number is only the upper limit of the number of forces to compute, i.e., it is impossible to know beforehand how many particles will be actually close enough to exert a force.

Chapter 8

FRAMEWORK DESIGN

8.1 Motivation

Many-core GPUs have become an important computing platform in many scientific fields due to the high peak performance, cost effectiveness, and the availability of user-friendly programming environments, e.g., NVIDIA CUDA [108] and ATI Stream [6]. In the literature, many works have been reported on how to harness the massive data parallelism provided by GPUs [22, 31, 71, 103, 107, 122, 126, 152]. Beyond single-GPU systems, there is a growing interest in exploiting multiple GPUs for scientific computing [47, 62, 64, 76, 117, 136]. The main benefit of using multi-GPU systems is that such systems can provide a much higher performance potential than the single-GPU systems. Further, multi-GPU systems can overcome certain limitations associated with the single-GPU systems, e.g., limited global memory. Also, these works demonstrate that such platforms can be beneficial in terms of power, price, etc.

However, issues, such as load balancing and GPU resource utilization, cannot be satisfactorily addressed by the current GPU programming paradigm. For example, as shown in Section 9.3, CUDA scheduler cannot handle the unbalanced workload efficiently. Also, for problems that do not exhibit enough parallelism to fully utilize the GPU, employing the canonical GPU programming paradigm will simply underutilize the computation power. These issues are essentially due to fundamental limitations on the current data parallel programming methods. On the

other hand, some applications do require a more refined execution behavior on GPU-enabled systems than CUDA. For example, as suggested in [117], a fine-grained GPU execution model would allow fine-grained message-driven applications to overlap the communication with the computation on GPU clusters.

In this chapter, we propose a fine-grained task-based execution framework that can dynamically balance workload on individual GPUs and among GPUs, and thus utilize the underlying hardware more efficiently.

Introducing tasks on GPUs is particularly attractive for the following reasons. First, although many applications are suitable for data parallel processing, a large number of applications show more task parallelism than data parallelism, or a mix of both [29, 120]. Having a task parallel programming scheme will certainly facilitate the development of this kind of applications on GPUs. Second, by exploiting task parallelism, it is possible to show better utilization of hardware features. For example, task parallelism is exploited in [112] to efficiently use the on-chip memory in the GPU. Third, in task parallel problems, some tasks may not be able to expose enough data parallelism to fully utilize the GPU. Running multiple such tasks on a GPU concurrently can increase the utilization of the computation resource and thus improve the overall performance. Finally, with the ability to dynamically distribute fine-grained tasks between CPUs and GPUs, the workload can potentially be distributed properly to the computation resources of a heterogeneous system, and therefore achieve better performance.

However, achieving task parallelism on GPUs can be challenging; the majority of the conventional NVIDIA CUDA programming methodologies and techniques implies that programmer-defined functions should be executed sequentially

on the GPU[1]. Open Computing Language (OpenCL) [87] is an emerging programming standard for general purpose parallel computation on heterogeneous systems. It supports the task parallel programming model, in which computations are expressed in terms of multiple concurrent tasks where a task is a function executed by a single processing element, such as a CPU thread. However, this task model is basically established for multi-core CPUs, and does not address the characteristics of GPUs. Moreover, it does not require a particular OpenCL implementation to actually execute multiple tasks in parallel.

Next we first present a queue-based design for single-GPU systems. We then generalize the idea and extend it to multi-GPU systems as a general fine-grained task-based execution framework for GPU-enabled systems.

8.2 Design for Single-GPU Systems

In this section, we first describe the basic idea of the fine-grained task-based execution. Then we discuss the necessary mechanisms to perform host-device interactions correctly and efficiently, and then present the design for single-GPU systems in detail.

8.2.1 Basic Idea

With the current CUDA programming paradigm, to execute multiple tasks, the host process has to sequentially launch multiple, different kernels, and the hardware is responsible for arrange how kernels run on the device [109]. This paradigm is illustrated in Figure 8.1. On the other hand, in our design, instead of launching multiple kernels for different tasks, we launch a *persistent kernel* with B thread

[1] The latest NVIDIA Fermi architecture supports only 4 concurrent kernels (will increase to 16). Our approach can provide even finer-grained concurrent execution, and can also be applied to this new architecture to further improve its utilization.

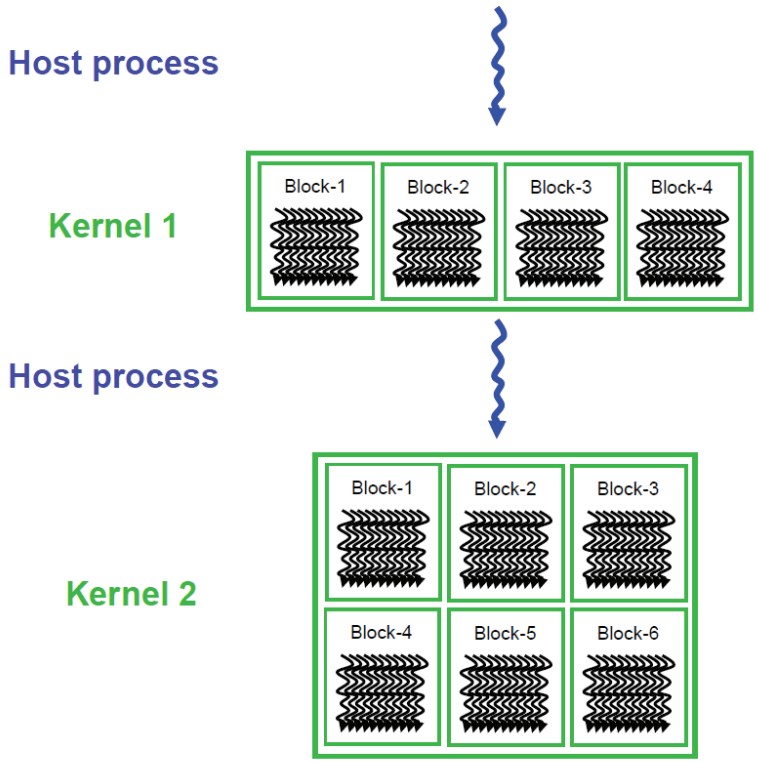


Figure 8.1: CUDA Programming Paradigm

blocks (TBs), where B can be as big as the maximum number of concurrently active TBs that a specific device can support. Since CUDA will not *swap out* TBs during their execution, after being launched, all TBs will stay active, and wait for executing tasks until the kernel terminates. When the kernel is running on the device, the host process enqueues both computation tasks and signaling tasks to one or more task queues associated with the device. The kernel dequeues tasks from the queues, and executes them according to the pre-defined task information. In other words, the host process dynamically controls the execution of the kernel by enqueuing tasks, which could be homogeneous or heterogeneous. This task queue idea is illustrated in Figure 8.2. Because of the nature of the task queue idea, we

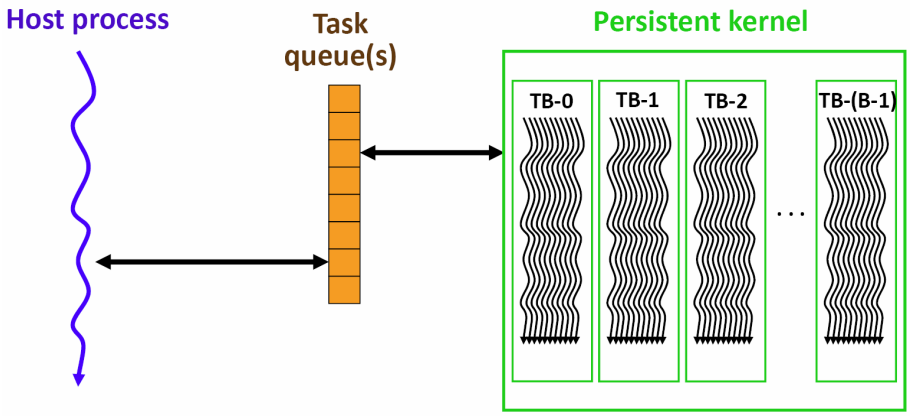


Figure 8.2: Fine-grained Task Queue Paradigm

require that the host process can perform copies between the host memory and device memory concurrently with the kernel execution. This is called "asynchronous concurrent execution" in CUDA, and is available on CUDA devices that support the **deviceOverlap** feature.

8.2.2 Preliminary Considerations

Since task queues are usually generalized as producer-consumer problems, let us first consider the single-producer single-consumer case. Algorithm 1 and Algorithm 2 show the pseudo-code of *enqueue* and *dequeue* operations, respectively, for such scenario on a normal shared memory system. *queue* is a shared buffer between the producer and the consumer. *start* and *end* are indexes of next location for dequeue and enqueue, respectively. At the beginning, both indexes are initialized as 0. By polling *start* and *end*, the producer/consumer can determine if it can enqueue/dequeue tasks.

If we want to establish a similar scheme for the host-device communication, where the host process is the producer, and the kernel is the consumer, the following questions have to be addressed properly.

The very first question is where to keep the queue and associated index

Algorithm 1 ENQUEUE

Require: a task object *task*, a task queue *queue* of a capacity of *size*
Ensure: *task* is inserted into *queue*

1: **repeat**
2: $\quad l \leftarrow (end - start + size) \pmod{size}$
3: **until** $l < (size - 1)$
4: $queue[end] \leftarrow task$
5: $end \leftarrow (end + 1) \pmod{size}$

Algorithm 2 DEQUEUE

Require: a task queue *queue* of a capacity of *size*
Ensure: a task object is removed from *queue* into *task*

1: **repeat**
2: $\quad l \leftarrow (end - start + size) \pmod{size}$
3: **until** $l > 0$
4: $task \leftarrow queue[start]$
5: $start \leftarrow (start + 1) \pmod{size}$

variables. For index variables, a naïve choice would be having *end* in the host's memory system, and having *start* in the device's memory system. In this case, all updates to index variables can be performed locally. However, this choice introduces serious performance issue, i.e., each queue polling action requires an access to a index variable in another memory system, which implies a transaction across the host-device interface. This incurs significant latency (as shown in Section 9.1 for a PCIe bus). On the other hand, having both *start* and *end* on one memory system will not help; either the host process or the kernel has to perform two transactions across the host-device interface, which actually aggravates the situation.

The second question is how to guarantee the correctness of the queue operations in this host-device situation. Since each enqueue/dequeue operation consists of multiple memory updates, i.e., write/read to the queue and write to an index, it is crucial to ensure the correct ordering of these memory accesses while across the host-device interface. For example, for an enqueue operation described in Algorithm 1, by the update of *end* (line-5) is visible to the kernel, the insertion of *task*

(line-4) should have completed. If the ordering of these two memory accesses were reversed by the hardware/software for some reason, the kernel will not be able to see the consistent queue state, and therefore the whole scheme will not work correctly.

The final question is how to guarantee the correctness on accessing shared objects, if we allow dynamic load balance on the device. Lock is extensively used for this purpose. However, as presented in [28], a lock-based blocking method is very expensive on GPUs.

With evolving GPU technologies, now it is possible to answer above questions by exploiting the new hardware and software features. More specifically, since CUDA 2.2, a region of the host's memory can be mapped into the device's address space, and kernels can directly access this region of memory using normal load/store memory operations, provided that the device supports the **canMapHostMemory** feature. By duplicating index variables, and cleverly utilizing this feature, as we shall demonstrate in our design, the enqueue and dequeue operations can be redesigned such that the queue polling only incurs local memory accesses, and at most one remote memory access to an index variable is needed for a successful enqueue/dequeue operation. This addresses part of the first question, i.e., where to keep index variables.

The solution to the rest of the first question and the second question essentially requires mechanisms to enforce the ordering of memory access across the host-device interface. Memory fence functions are included in the new CUDA runtime. But they are only for memory accesses (made by the kernel) to the global and shared memory on the device. On the other hand, CUDA *event* can be used by the host process to asynchronously monitor the device's progress, e.g., memory accesses. Basically, an event can be inserted into a sequence of commands issued to a device. If this event is *recorded*, then all commands preceding it must have completed. Therefore, by inserting an event immediately after a memory access to

the device's memory system and waiting for its being recorded, the host process can guarantee that such memory operation has completed on the device, before it proceeds. This is equivalent to a memory fence for the host process to access the device's memory system. However, at this moment, there is no sufficient mechanism to ensure the ordering of memory accesses made by a kernel to the mapped host memory [100]. In this case, if we had the task queues residing on the host memory system, in the dequeue operation, the kernel cannot guarantee that the memory read on the task object has completed before updating the corresponding index variables. Therefore, the queue(s) can only reside in the global memory on the device. On the whole, by having the queue(s) on the device, and using both the event-based mechanism mentioned above and the device memory fence functions, we can develop correct enqueue and dequeue semantics that consist of memory accesses to both the host's memory system and the device's memory system.

With the advent of atomic functions on GPUs, such as *fetch-and-add* and *compare-and-swap*, it is possible to allow non-blocking synchronization mechanisms. This resolves the last question.

Here we summary all necessary mechanisms to enable correct, efficient host-device interactions as follows.

1. **Asynchronous concurrent execution:** overlap the host-device data transfer with kernel execution.

2. **Mapped host memory:** enable the light-weight queue polling without generating host-device traffic.

3. **Event:** asynchronously monitor the device's progress.

4. **Atomic instructions:** enable the non-blocking synchronization.

8.2.3 Notations

Before we present the queue-based design, we first introduce terminologies that will be used throughout the chapter. On a device, threads have a unique *local_id* within a TB, which is provided by the CUDA environment. In addition, each thread has a unique *thread_lane* within a warp, which indicates the logic index of this thread in a warp. This index can be computed from the *local_id* (within a TB) and the warp size. *host_write_fence()* is the event-based mechanism described above, which guarantees the correct ordering of memory stores made by the host process to the device's memory system. *block_write_fence()* is a fence function that guarantees that all device memory stores (made by the calling thread on the device) prior to this function are visible to all thread in the TB. *warp_write_fence()* is a fence function that guarantees that all device memory stores (made by the calling thread on the device) prior to this function are visible to all threads in the warp. *block_barrier()* synchronizes all threads within a TB. The above functions are either provided by CUDA or implemented by us. Please refer to Section 9.1 for implementation details.

For variables that are referred by both the host process and the kernel, we prefix them with either *h_* or *d_* to denote their actual residence on the host or the device, respectively. For variables residing in the device's memory system, we also post-fix them with *_sm* or *_gm* to denote whether they are in the shared memory or in the global memory. For those variables without any prefix or post-fix just described, they are simply host/device local variables.

8.2.4 Queue-based Design

Here we present our novel queue-based design, which allows automatic, dynamic load balancing on the device without using expensive locks. In this design, the host process sends tasks to the queue(s) without interrupting the execution of the kernel. On the device, all TBs concurrently retrieve tasks from these shared

queue(s). Since each task is executed by a single TB, we call this the *TB-level* task execution scheme.

A queue object is described by the following variables.

- *d_tasks_gm*: an array of task objects.
- *d_n_gm*: the number of tasks ready to be dequeued from this queue.
- *h_written*: the host's copy of the accumulated number of tasks written to this queue.
- *d_written_gm*: the device's copy of the accumulated number of tasks written to this queue.
- *h_consumed*: the host's copy of the accumulated number of tasks dequeued by the device.
- *d_consumed_gm*: the device's copy of the accumulated number of tasks dequeued by the device.

At the beginning of a computation, all those variables are initialized to 0s.

The complete enqueue and dequeue procedures are described in Algorithm 3 and Algorithm 4, respectively. To enqueue tasks, the host first has to check whether a queue is ready. In our scheme, we require that a queue is ready when it is empty, which is identified by $h\_consumed == h\_written$. If a queue is not ready, the host either waits until this queue becomes ready (single-queue case), or checks other queues (multi-queue case). Otherwise, the host first places tasks in *d_tasks_gm* starting from the starting location, and update *d_n_gm* to the number of tasks enqueued. Then it waits on *host_write_fence()* to make sure that the previous writes have completed on the device. After that, the host process updates *h_written*, and *d_written_gm* to inform the kernel that new tasks are available in the queue.

Algorithm 3 TB-LEVEL HOST ENQUEUE

Require: n task objects $tasks$, n_queue task queues q , each of a capacity of $size$, i next queue to insert

Ensure: host process enqueues $tasks$ into q

1: $n\_remaining \leftarrow n$
2: **if** $n\_remaining > size$ **then**
3: $n\_to\_write \leftarrow size$
4: **else**
5: $n\_to\_write \leftarrow n\_remaining$
6: **end if**
7: **repeat**
8: **if** $q[i].h\_consumed == q[i].h\_written$ **then**
9: $q[i].d\_tasks\_gm \leftarrow tasks[n - n\_remaining : n - n\_remaining + n\_to\_write - 1]$
10: $q[i].d\_n\_gm \leftarrow n\_to\_write$
11: $host\_write\_fence()$
12: $q[i].h\_written \leftarrow q[i].h\_written + n\_to\_write$
13: $q[i].d\_written\_gm \leftarrow q[i].h\_written$
14: $i \leftarrow (i + 1) \pmod{n\_queues}$
15: $n\_remaining \leftarrow n\_remaining - n\_to\_write$
16: **if** $n\_remaining > size$ **then**
17: $n\_to\_write \leftarrow size$
18: **else**
19: $n\_to\_write \leftarrow n\_remaining$
20: **end if**
21: **else**
22: $i \leftarrow (i + 1) \pmod{n\_queues}$
23: **end if**
24: **until** $n\_to\_write == 0$

Algorithm 4 TB-LEVEL DEVICE DEQUEUE

Require: n_queue task queues q, i next queue to work on
Ensure: TB fetches a task object from q into $task\_sm$

```
 1: done ← false
 2: if local_id == 0 then
 3:    repeat
 4:       if q[i].d_consumed_gm == q[i].d_written_gm then
 5:          i ← (i + 1)   (mod n_queues)
 6:       else
 7:          j ← fetch_and_add(q[i].d_n_gm, −1) − 1
 8:          if j ≥ 0 then
 9:             task_sm ← q[i].d_tasks_gm[j]
10:             block_write_fence()
11:             done ← true
12:             jj ← fetch_and_add(q[i].d_consumed_gm, 1)
13:             if jj == q[i].d_written_gm then
14:                q[i].h_consumed ← q[i].d_consumed_gm
15:                i ← (i + 1)   (mod n_queues)
16:             end if
17:          else
18:             i ← (i + 1)   (mod n_queues)
19:          end if
20:       end if
21:    until done
22: end if
23: block_barrier()
```

On the device, each TB uses a single thread, i.e., the one of $local\_id == 0$, to dequeue tasks. It first determines whether a queue is empty by checking $d\_consumed\_gm$ and $d\_written\_gm$. If a queue is empty, then it keeps checking until this queue has tasks (single-queue case) or checks other queues (multi-queue case). Otherwise, it first applies *fetch-and-add* on d_n_gm with -1. If the return value of this atomic function is greater than 0, it means there is a valid task available in the queue, and this TB can use this value as the index to retrieve a task from d_tasks_gm. The thread waits on the memory fence until the retrieval of the task is finished. It then applies *fetch-and-add* on $d\_written\_gm$ with 1, and checks whether the task just retrieved is the last task in the queue by comparing the return value of the atomic function just called with $d\_written\_gm$. If yes, this thread is responsible for updating $h\_consumed$ to inform the host process that this queue is empty. Barrier (Algorithm 4:Line 23) is used to make sure that all threads in a TB will read the same task information from $task\_sm$ after the dequeue procedure exits. The dequeue procedure is a wait-free [75] approach, in the sense that in a fixed steps, a TB either retrieves a task from a queue, or finds out that queue is empty.

To avoid data race, this scheme does not allow the host process to enqueue tasks to an non-empty queue that is possibly being accessed by the kernel. This seems to be a shortcoming of this scheme because enqueue and dequeue operations cannot be carried out concurrently on a same queue. However, simply employing multiple queues can efficiently solve this issue by overlapping enqueue with the dequeue on different queues.

To determine when to signal the kernel to terminate, the host process has to check whether all queues are empty after all computation tasks have been enqueued. If it is true, the host process enqueues B HALT tasks to the queue, one for each of the B concurrently active TBs on the device. TBs exit after getting HALT, and eventually the kernel terminates.

8.2.5 An Even Finer Task Execution Design

While the TB-level task execution scheme described in Section 8.2.4 is finer-grained than the normal CUDA scheme (where a function is executed by the entire device), it is not necessarily the optimal granularity for executing tasks. For example, if each task only exposes limited data parallelism, which can be handled by a few threads, using the TB-level task execution scheme simply wastes the computation power of other threads in the same TB. Therefore, we propose a *warp-level* task execution scheme, where tasks are fetched and executed by individual warps on the device. Since, on the GPU, the SM creates, manages, schedules, and executes threads in warps, the warp-level task execution scheme perfectly matches this architectural feature, and therefore can potentially utilize the hardware more efficiently than the TB-level scheme, and the normal CUDA programming paradigm. On the other hand, using even finer-grained schemes, such as individual threads, will not help. Since all threads within a warp share an instruction issue unit, they cannot execute different codes concurrently. In fact, the most efficient execution on GPU is that all threads of a warp take the same execution path [109].

Compared to the TB-level scheme, this warp-level scheme only involves changes on the device side. Therefore, Algorithm 3 can be reused for task sending in this warp-level scheme as well. The complete procedure for warp-level device fetching task is described in Algorithm 5.

On the device, each warp uses a single thread, i.e., the one of $warp\_lane == 0$, to fetch tasks. This thread first determines whether a queue has tasks available by checking $d\_consumed\_gm$ and $d\_written\_gm$. If a queue is empty, then it keeps checking until the host process has inserted tasks into this queue (single-queue case) or checks other queue (multi-queue case). Otherwise, it first applies *fetch-and-add* on d_n_gm with -1. If the return value of this atomic function is greater than 0, it means there is a valid task available in the queue, and this warp uses this value

Algorithm 5 WARP-LEVEL DEVICE DEQUEUE

Require: n_queue task queues q, i next queue to work on
Ensure: Warp fetches a task object from q into $task\_sm$

```
 1: done ← false
 2: if warp_lane == 0 then
 3:    repeat
 4:       if q[i].d_consumed_gm == q[i].d_written_gm then
 5:          i ← (i + 1)   (mod n_queues)
 6:       else
 7:          j ← fetch_and_add(q[i].d_n_gm, −1) − 1
 8:          if j ≥ 0 then
 9:             task_sm ← q[i].d_tasks_gm[j]
10:             warp_write_fence()
11:             done ← true
12:             jj ← fetch_and_add(q[i].d_consumed_gm, 1)
13:             if jj == q[i].d_written_gm then
14:                q[i].h_consumed ← q[i].d_consumed_gm
15:                i ← (i + 1)   (mod n_queues)
16:             end if
17:          else
18:             i ← (i + 1)   (mod n_queues)
19:          end if
20:       end if
21:    until done
22: end if
```

as the index to retrieve a task from *d_tasks_gm* (Algorithm 5: Line 9). This thread waits on the memory fence until the retrieval of the task is finished. It then checks whether the task just retrieved was the last task in the queue (Algorithm 5: Line 12-13). If yes, this thread informs the host process that this queue is empty by updating *h_consumed*. Since threads in a warp execute in a synchronized way (no explicit barriers needed), after the task fetching procedure finishes, all threads in a warp see the same value in *task_sm*. Like the TB-level scheme, this task fetching procedure of the warp-level scheme is also wait-free.

8.3 Design for Multi-GPU Systems

How to efficiently utilize single-GPU systems for general purpose scientific computing has been investigated for many applications. There are continuing efforts to facilitate programming GPU clusters. The work in [48] employs Global Arrays (GA) [104] to simplify the communications among GPUs. A memory consistency model is proposed in [97] to enable a distributed shared memory system, which consists of texture memory across multiple GPUs. Performance modeling of multi-GPU systems and GPU clusters is studied in [125]. Results show that such modeling techniques can be accurate for applications of a deterministic execution manner.

One of the major challenges of using multiple GPUs concurrently is the load balancing issue. This is particularly true when the target applications exhibit irregular, unbalanced workload, or the computation is to be carried out on heterogeneous platforms, e.g., consisting of both CPUs and GPUs, or a diversity of GPUs of varying capability. A static scheduling approach will not work since it lacks the ability to automatically adapt to the application irregularity and system heterogeneity. One possible approach is to decompose the computation into small chunks. Whenever a GPU is free, it receives a chunk for processing. While this approach potentially can provide better load balancing behavior than the static approach, the overall performance of the program is heavily affected by the granularity of the chunks.

Coarse-grained chunking may lead to load imbalance among GPUs, and therefore hurt the performance. Generally speaking, using finer-grained chunks can achieve better load balancing. When the workload in each chunk becomes smaller and smaller, a single chunk may not be able to present enough parallelism to fully utilize the GPU. However, most of the CUDA programming methods suggest that all programmer-defined functions run sequentially on the GPU. Therefore, using these fine-grained chunks could result in the underutilization of the GPUs, and degrade the overall performance. This issue cannot be satisfactorily addressed by the current GPU programming methodologies and techniques.

Our approach for solving these issues is to allow concurrent execution of fine-grained tasks on multi-GPU systems. Specifically, in our approach, the granularity of task execution is finer than what is currently supported in CUDA; the execution of a task only requires a subset of the GPU hardware resources. While some tasks are being processed by part of the GPU resources, CPU can dispatch other homogeneous/heterogeneous tasks to this GPU, and these tasks can be processed by using other part of the GPU resources. All tasks can be processed concurrently and independently, assuming there is no dependence among them.

In this section, we generalize our design for single-GPU systems and present a fine-grained task-based execution framework for multi-GPU systems, where one such system is a compute node equipped with multiple GPUs. This fine-grained approach is particularly attractive because of the following reasons. First, it provides means for achieving efficient, and dynamic load balancing on multi-GPU systems. While scheduling fine-grained tasks enables good load balancing among multiple GPUs, concurrent execution of multiple tasks on each single GPU solves the hardware underutilization issue when tasks are small. Second, since our approach allows the overlapping executions of homogeneous/heterogeneous tasks, the programmers will have the flexibility to arrange their applications with fine-grained tasks and apply

dataflow-like solutions to increase the efficiency of the program execution.

8.4 A Fine-grained Task-based Execution Framework

The multi-GPU systems discussed in this study can be viewed as illustrated in Figure 8.3. In the system shown, multiple devices are connected to the host via a PCIe bus. With the current CUDA environment, devices cannot exchange data with each other directly. Instead, data movements across devices have to be done by the host processes.

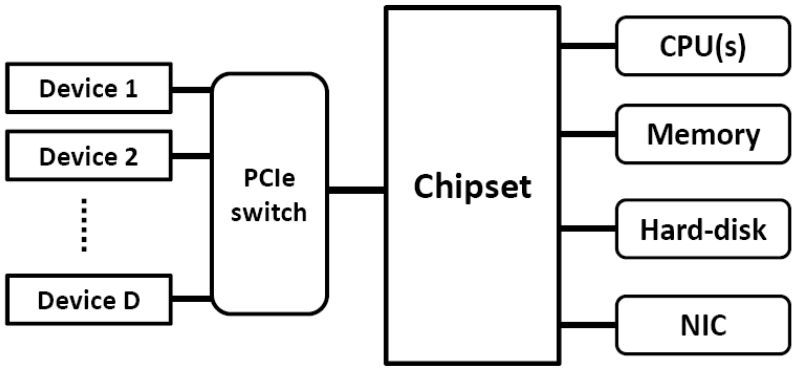


Figure 8.3: A PCIe Connected Multi-GPU System

To efficiently utilize such multi-GPU systems, we propose a fine-grained task-based execution framework, which is demonstrated in Figure 8.4.

In our framework, for each host process-device pair, one or more programmer-created *local task containers* are used to enable the host-device communications when a kernel is running on the device. These task containers can be of the form of regular queues for in-order processing, or priority queues to support tasks with different priorities[2]. Such local task containers are only accessible by the corresponding host process and the device. The computation to be carried out on a multi-GPU system

[2] In this study, we only show the designs with the regular FIFO queues.

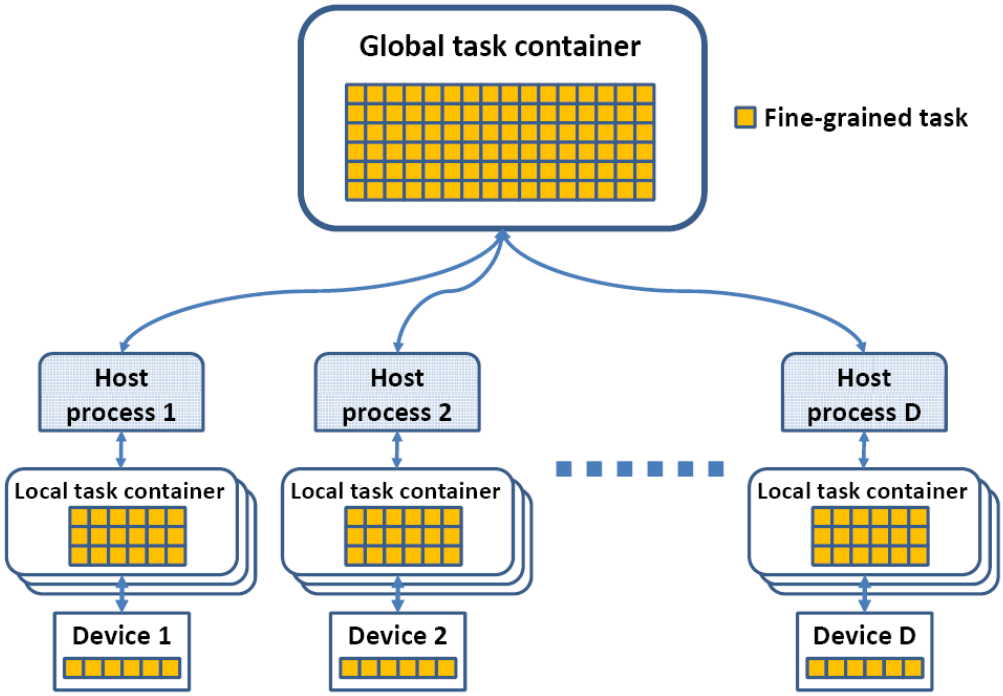


Figure 8.4: Fine-grained Task-based Execution Framework for Multi-GPU Systems

is first decomposed into many fine-grained tasks, which are kept in a programmer-created *global task container*. All individual host processes can fetch/send tasks from/to this global task container. Once a host process finds out that its own local task container has free space and some tasks in the global task container are ready to start, it moves a number of such tasks from the global task container to its own local task container, and informs the device the availability of new tasks.

On each device, a persistent kernel is launched at the beginning of the computation. This kernel fetches tasks from the local task container(s), and executes them by groups of threads, which we call *task execution units* (TEUs). Note that the processing of each task is carried out by a single TEU, which can be at a granularity finer than the entire device. Multiple tasks can be fetched and processed by different TEUs (on a same device) concurrently and independently, assuming there is no

dependence among them. Moreover, task sending (by the host process) and task fetching (by the TEUs) can happen at the same time, as illustrated in Figure 8.5. One of the challenges to implement this device-scope fine-grained execution scheme

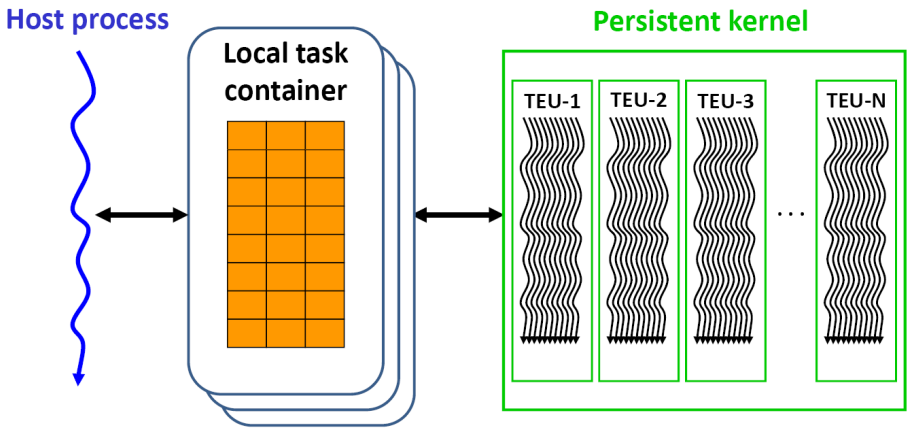


Figure 8.5: General Form of Fine-grained Execution on a Single GPU

is to provide a correct and efficient host-device communication mechanism when a kernel is running on the device. As demonstrated in Section 8.2, by judiciously utilizing the GPU's architectural features and the mechanisms provided by the CUDA environment, such as the multiple memory spaces, and the asynchronous concurrent execution, this mechanism can be achieved for host-device communications.

8.5 Dynamic Load Balancing Design for Multi-GPU Systems

Based on our fine-grained task-based framework described in Section 8.4, a design for dynamically balancing workload on multi-GPU systems is quite straightforward. The design follows the basic structure illustrated in Figure 8.4. Specifically, the work to be processed with a multi-GPU system is decomposed into fine-grained tasks, which are to be executed by individual warps. When a local task container becomes empty, the corresponding host process fills it with certain number of fine-grained tasks retrieved from the global task container. Here we assume that the dependencies among tasks have been taken care of by the host processes, and all

tasks in the local task containers can be executed independently by devices. Since all host processes share a host memory space, the orchestrations among them can be accomplished with regular programming methodologies and techniques for shared-memory systems.

8.6 Related Work

Load balance is a critical issue for parallel processing. However, in the literature, there are few studies addressing this load balancing issue on GPUs. The load imbalance issue of graphics problems was discussed in [51, 101], and authors observed that it is of fundamental importance for high performance graphics algorithm implementations on GPUs. Several static and dynamic load balancing strategies were evaluated in [28] on GPUs for an octree partitioning problem. The experimental results show that synchronization can be very expensive on GPUs, and non-blocking mechanisms or other methods that can take advantage of the GPU architectural features should be used for the purpose of dynamic load balancing. Our work differs from this study in several ways. First, in the former study, the load balancing strategies were carried out solely on the GPU; the CPU cannot interact with the GPU during the execution. Second, the former study only investigated single-GPU systems. Our work is performed on both single- and multi-GPU systems, and can be easily extended to GPU clusters.

Scheduling task execution on GPU-enabled systems and other heterogeneous platforms has been investigated in a few studies. A runtime scheduler is presented for situations where individual kernels cannot fully utilize GPUs [69]. This runtime intercepts the stream of kernel invocations to an GPU, and makes scheduling decisions and merges the workload from multiple kernels and creates a super-kernel based on them. Experiments with both micro-benchmarks and a near neighbor search program show the proposed runtime can improve the hardware utilization when multiple kernels can fit simultaneously on the hardware. However, the merge

has to be performed statically, and thus dynamic load balance cannot be guaranteed. Recently, researchers have begun to investigate how to exploit heterogeneous platforms with the concept of tasks. Merge [93] is such a programming framework proposed for heterogeneous multi-core systems. It employs a library-based method to automatically distribute computation across the underlying heterogeneous computing devices. The Merge programming model abstracts the underlying architectures and requires additional information from the programmer for dispatch decisions. A prototyping implementation shows performance gains on both a heterogeneous platform and a homogeneous platform for a set of benchmarks. STARPU [11] is another framework for task scheduling on heterogeneous platforms, in which hints, including the performance models of tasks, can be given to guide the scheduling policies. It features a uniform execution model for both CPUs and GPUs, a high-level framework to design scheduling policies and a library that automates data transfers. Another similar effort is reported in [86], where schemes for efficient automatic task distribution between CPU and GPU are proposed and tested with a real-time system. Our work is orthogonal to prior efforts in that our solution exhibits excellent dynamic load balance for GPU-enabled systems. It also enables the GPU to exchange information with the CPU during execution, which further enables these platforms to understand the runtime behavior of the underlying devices, and improve the performance of the system.

Chapter 9

IMPLEMENTATION AND EVALUATION

In this chapter, we first describe implementation issues of our framework. We then present the experimental results for both the micro-benchmarks and a molecular dynamics (MD) application.

9.1 Implementation

We implemented our fine-grained task-based execution framework on a system equipped with 1 quad-core AMD Phenom II X4 940 processors and 4 NVIDIA Tesla C1060 GPUs. The system is running 64-bit Ubuntu version 8.10, with NVIDIA driver version 190.10. CUDA Toolkit version 2.3 ,CUDA SDK version 2.3, and GCC version 4.3.2 were used in the development. The above system provides all necessary features to implement our framework.

To utilize the asynchronous concurrent execution feature, CUDA requires using different, nonzero CUDA *streams*, where a stream is basically a sequence of commands that are performed in order on the device, and the zero stream is the default stream in CUDA. So, in our implementation, we use one stream for kernel launching, another stream for performing queue operations.

While CUDA does provide a TB-scope memory fence function, $\_\_threadfence\_block()$, it does not differentiate between stores and loads. In our implementations, it was used as store fences, *block_write_fence()*. On the other hand, CUDA does not have a warp-scope memory fence function like *warp_write_fence()*.

Therefore, in our implementation, we simply use the TB-scope memory fence function, *__threadfence_block()* for this purpose. CUDA also provides a function to synchronize all threads in a TB, *__syncthreads*(), which behaves as both a regular barrier and also a memory fence in a TB. In our implementations, we took advantage of this and eliminated redundant operations. *host_write_fence()* was implemented with CUDA event methods. For two consecutive asynchronous memory operation issued by the host (to the device's global memory), a combination of an event recording and an event synchronizing inserted between these two memory operations ensures that the second operation will not start unless the first one finishes on the device.

9.2 Micro-benchmarks

Here we report benchmarking results for major components performed in our framework, such as, host-device data transfer, synchronizations, atomic instructions, and the complete enqueue/dequeue operations. Performance measurements of these individual components help us understand how our designs work with the real applications.

Host-device data transfer

Every time the host process performs an enqueue operation, or the kernel writes the mapped host memory, it involves host-device data transfers. The host-device interface equipped in our system is PCIe 2.0 $\times$ 8. We measured the time to transfer contiguous data between the host and the device across this interface using the pinned memory. Since queue operations only update objects of small sizes, i.e., tasks and index variables, we conducted the test for sizes from 8 bytes to 4KB. Figure 9.1 shows measured transfer times for transfers initialized by the host process, i.e., using the regular synchronous copies (Memcpy) with the zero stream, asynchronous copies (MemcpyAsync) with the zero stream, and asynchronous copies with a nonzero stream, and the transfers initialized by the kernel, i.e., the mapped

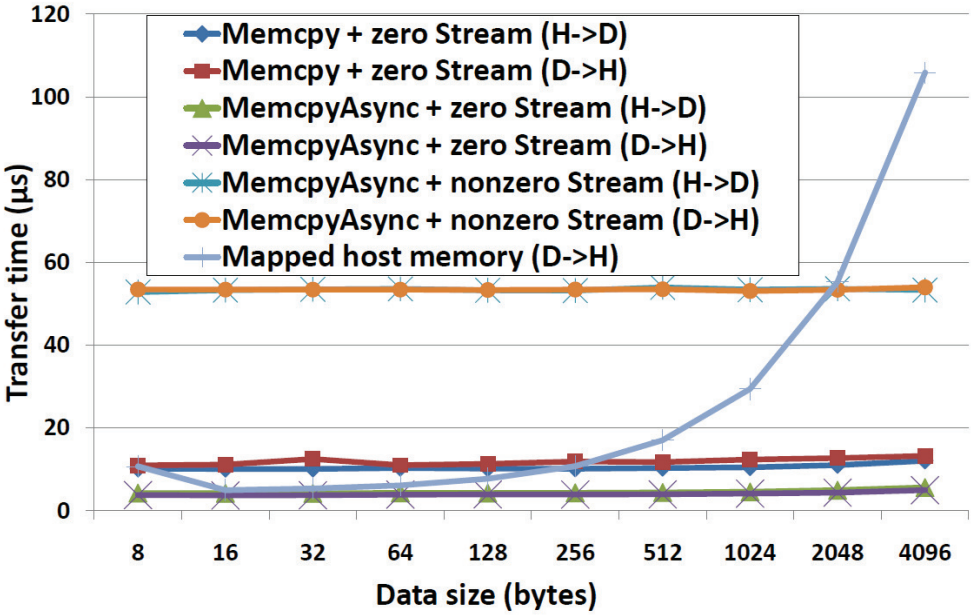


Figure 9.1: CPU-GPU Data Transfer Time

host memory, where H->D and D->H indicate the transfer from the host to the device, or the reverse, respectively. Note that one device thread was used to perform transfers from the device to the mapped host memory. From the figure, it is clear that using a nonzero stream to perform asynchronous copies is much expensive, compared to both synchronous copies and asynchronous copies performed with the zero stream, i.e., 5×-10× slower. Without exposing to the CUDA internal, we do not really understand why such operation is so costly. On the other hand, with the current CUDA programming environment, using nonzero streams is the only way to achieve the asynchronous concurrent execution.

For transfers initialized by the host process, the transfer time changes slowly in the above data range due to the high bandwidth, i.e., 4GB/s. So, if the host-device data transfer is inevitable, combining multiple data accesses into one single transaction is highly recommended. In fact, in our implementation of the enqueue

operation, we actually update d_n_gm and d_tasks_gm with a single host-device transaction.

On the other hand, since there is no mechanism for a kernel to copy a contiguous memory region, it has to perform such copy with assignments on basic data types. Therefore, the transfer time is quite linearly proportional to the size of data, compared to transfers issued by the host process.

Barrier and fence

Barrier and memory fence functions are used in our design to ensure the correctness of the operations. In this test, we made all threads (on the device) calling a specific barrier or fence function a large number of times, and measured the average completion time. For the fence function, we also measured the case that only one thread in each TB makes the call, which emulates the scenario in dequeue operations.

Figure 9.2 shows the results for these functions with a TB size of 128. We observed that the completion time of these functions tends to keep constant regardless the number of TBs launched. Especially, the fence functions are very efficient; it takes a same amount of time to complete for the case when called by a single thread in a TB (annotated with "one T/B" in the figure), and for the case when called by all threads in a TB (annotated with "all Ts/B"). Similar results were observed for various TB sizes.

Atomic instructions

Atomic functions are used in our design to guarantee correct dequeue operations on the device. In this benchmark, one thread in each TB performs a large number of *fetch-and-add* function on a device's memory address in the global memory. Experimental results show that atomic functions are being executed serially,

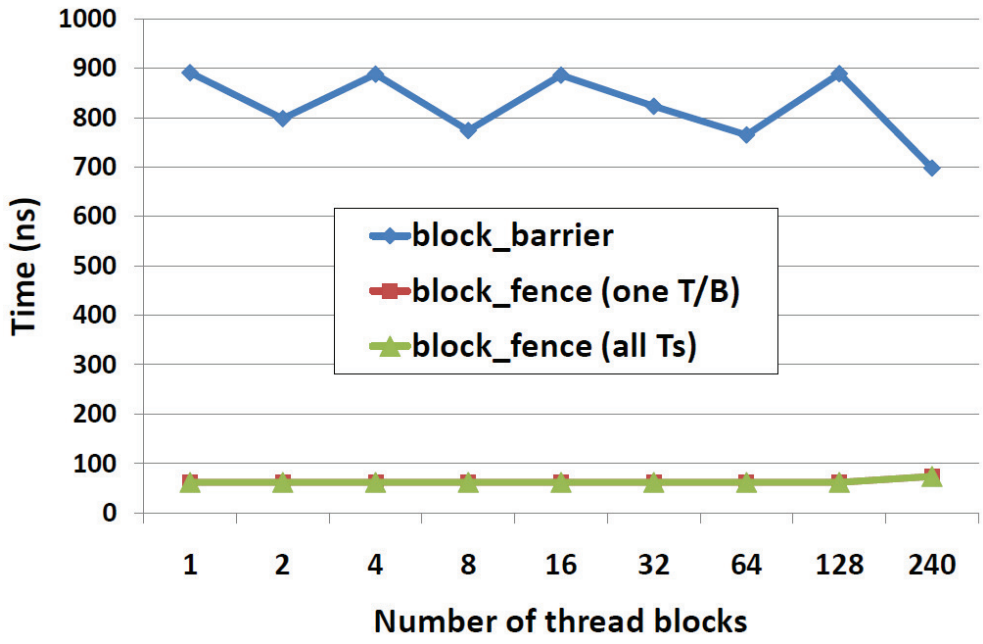


Figure 9.2: GPU Barrier and Fence Functions (128Ts/B)

and the average completion time is $327ns$. Experiments with other atomic functions show similar results.

Task queue operations

We conducted experiments to show the average overhead of each enqueue and dequeue operation for our framework. For the enqueue operation, this was measured by calling an enqueue operation many times without running a kernel on the device. In experiments, each enqueue operation places 120 tasks in the queue. For the dequeue operations, we first pre-loaded queues with a large number of tasks, and then we launched a kernel that only retrieves tasks from queues, without performing any real work. The average enqueue operation is $114.3\mu s$, and the average dequeue operation is $0.4\mu s$ when the dequeue kernel was run with 120 TBs, each of 128 threads, for both the TB-level scheme and the warp-level scheme. Comparing

these numbers with Figure 9.1, it is clear that host-device data transfers account for the major overhead in enqueue operations. For example, 2 PCIe transactions in enqueue operations need approximately $110\mu s$ to finish, which is about 95% of the overall enqueue time. While this seems a very high overhead, by overlapping enqueue operations with the computation on devices, solutions based on our framework actually outperform several alternatives, for a MD application, as shown in Section 9.3,

We also conducted experiments for enqueue operations with varied number of tasks in each operation. We observed that inserting more tasks with one operation only incurs negligible extra overhead, when a single queue can hold these tasks. On the other hand, the average dequeue time is reduced when more TBs are used on the device. For example, when increasing the number of TBs from 16 to 120, the average dequeue time decreases from $0.7\mu s$ to $0.4\mu s$, which is about the time to complete an atomic function. This indicates that our dequeue algorithm actually enables concurrent accesses to the shared queue from all TBs, with very small overhead.

9.3 Case study: Molecular Dynamics

The MD systems used in our study is synthetic unbalanced systems. Such a synthetic unbalanced system is built by following a Gaussian distribution of helium atoms in a 3D space. The system has a higher density in the center than in periphery. The density decreases from the center to the periphery following a Gaussian curve. Therefore the force contributions for the atoms at the periphery are much less than those for the atoms close to the center. An example of such system is illustrated in Figure 9.3. The force between atoms is calculated using both electrostatic potential and Lennard-Jones potential [55].

The reason for using synthetic systems is two-fold: (1) synthetic systems can isolate the load balancing issue from other complex facts exhibiting in the real life systems, and therefore facilitates the evaluations and analyses of different solutions,

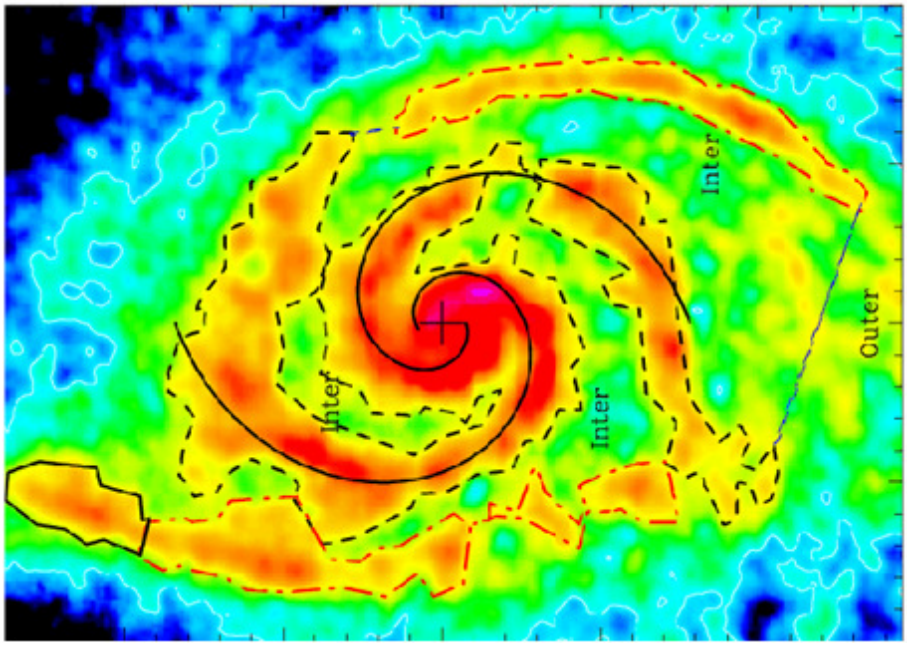


Figure 9.3: Example Synthetic MD System

(2) it is very difficult to find real life examples where a particular atom distribution is constant as the simulated system size scales up, and therefore it makes very hard to objectively evaluate different solutions with different system sizes.

For each system, the N atom positions are stored in a linear array A. Specifically, the 3D space is first decomposed in boxes of size equal to the *cutoff* radius. Then, atoms in each individual box are stored into the array contiguously. This data layout reduces the thread divergence as atoms processed in a warp are most likely close to each other in the physical system, and, with high probability, they will follow the same control path, which is the most efficient execution way on GPUs [109]. Therefore, this data layout is efficient for processing with single-GPU systems. However, due to the effect of cutoff radius, the systems built with non-uniform distributions exhibit irregular, unbalanced computation workload for different boxes. Consequently, using this data layout with multi-GPU systems can

be challenging, in terms of the load balancing and the absolute performance.

9.3.1 Solution Implementations

We implement solutions based on our fine-grained task-based execution framework, using both the TB-level scheme and the warp-level scheme. We also implement other load balance techniques based on the conventional CUDA programming method.

Solution STATIC

This solution statically divides A into P contiguous regions of equal size, where P is the number of devices used. Each device is responsible for computing forces for atoms within a region, where each TB is responsible for evaluating 128 atoms, and the number of TBs is determined by the size of the region. The computation of a time step finishes when all devices finishes their regions.

This solution reduces the thread divergence as atoms processed in a TB will follow most likely the same control path, which is the most efficient execution way on GPUs. Due to this feature this method is expected to be one of the fasted method for single GPU, however partitioning at multi-GPU level is very difficult for systems of unbalanced workload in the space. An uneven portioning has to be performed as the cost of distance calculation and force calculation has to be proportionally taken in account. This solution is designed to take advantage of single GPU computing sacrificing multi-GPU load balancing. We use it in a multi-GPU configuration equally dividing A into P contiguous regions, knowing in advance that it will have poor load balance behavior. The objective is to use it as a baseline to compare other load-balancing schemes in the multi-GPU experiments.

Solution RANDOM

This solution randomly permutes the elements in A before the simulation, and after every certain amount of time steps in the simulation. After the permutation, the workload is distributed as Solution STATIC; the array A is equally partitioned into P contiguous regions, one for each device.

By discarding the locality information of atoms, this solution ensures almost perfect load balance among multiple devices, since now each atom in the array has a (nearly) equal probability to exert a force with all other atoms in the array. This technique is used in parallel implementations of state-of-the-art biological MD programs such as CHARMM [24] and GROMOS [33]. However, when applied to the GPU codes, it introduces the problem of thread divergence inside a warp for simulating systems of non-uniform atom distributions, as now atoms with a lot of force interactions are mixed with atoms with few force interactions.

The randomization procedure is performed on the host, and we do not include its execution time into the overall computation time. Note that randomization procedure has a computational complexity $\Theta(N)$ and therefore can be fairly used in atom-decomposition MD computation which has complexity $\Theta(N^2)$.

Solution CHUNKING

This solution is a dynamic approach that uses fine-grained workload to dynamically balance load across multiple devices. Specifically, the array A is decomposed into many data chunks of equal atoms. Whenever a host process finds out that the corresponding device is free (on kernel running on the device), it assigns the force computation of atoms within a data chunk to the device, by launching a kernel with the data chunk information. The host process waits until this kernel completes the computation and the device becomes free again, then it launches another kernel with a new data chunk.

This solution is designed to take advantage of both good load balancing among multiple GPUs and thread convergence. Since a device only receives a relatively small workload after it finishes the current one, this approach potentially provides better load balancing than Solution STATIC, for unbalanced workload. Since the performance of chunking is affected by the chunk size, we use an empirically optimal size of $15,360$ atoms/chunk (120 TBs $\times$ 128 atoms) for this solution, which achieves the best absolute performance among all examined chunk sizes.

Solution TB-TASK

This solution is an approach that employs our fine-grained task-based execution framework; it is based on the TB-level scheme (presented in Section 8.2.4). In this solution, each task is the force evaluation of 128 atoms (stored contiguously in A) with all atoms in the system, and it is to be executed by a single TB. The simulation of each time step is decomposed into tasks, which are kept in the global task container. On each device, two local task containers are used to overlap the host task sending with the device task fetching. Each local task container holds up to 20 tasks. Whenever a task container of a device becomes empty, the corresponding host process tries to fetch as much as 20 tasks from the global task container at a time, and sends them to the device with a single task sending procedure. The kernel is run with 120 TBs, each of 128 threads. Note these configuration numbers are determined empirically.

Solution WARP-TASK

This solution is also built on our fine-grained task-based framework. Unlike Solution TB-TASK, this solution is based on the warp-level scheme (presented in Section 8.2.5). To accommodate this granularity change of TEUs, the granularity of each task is accordingly decreased to the force evaluation of 32 atoms (stored contiguously in A) with all atoms in the system. The kernel is run with 120 TBs,

each of 128 threads, i.e., 512 warps on a single device. Like Solution TB-TASK, these configuration numbers are determined empirically.

Note that all 5 solutions use the same device function to perform the force computation, which is based on the atom-decomposition [132] technique. Also, before timing the computation, the array A is already available on devices. In this way, we can ensure that all performance differences are only due to the load balancing mechanisms employed.

9.4 Results and Discussions

We conduct our experiments on the 4-GPU system described in Section 9.1. For each run of the simulation, we use the average runtime in the first 10 time steps as the metric for the absolute performance (the runtime differences among these 10 time steps are trivial). We first evaluate all solutions presented in Section 9.3.1 with identical input data from the unbalanced system described in Section 9.3, for both the single-GPU scenario and the multi-GPU scenario. We then investigate the effect of different atom distributions.

9.4.1 Single-GPU Scenario

Figure 9.4 shows the normalized speedup of the average runtime per time step over Solution STATIC, with respect to different system sizes, when only 1 GPU is used in the computation.

As discussed in Section 9.3.1, unlike other approaches, Solution RANDOM does not exploit the spatial locality in the system, and thus causes severe thread divergences within a TB. For example, for a 512K atoms system, the CUDA profiler reports that Solution RANDOM occurs 49% more thread divergences than Solution STATIC, and its average runtime per time step is 74% slower than Solution STATIC.

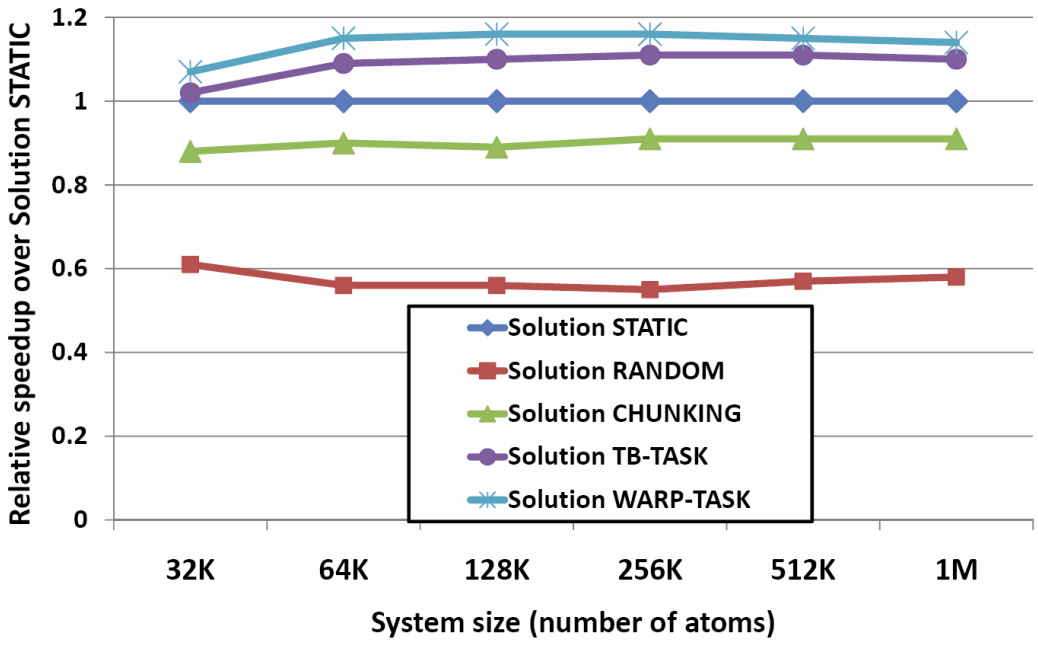


Figure 9.4: Relative Speedup over Solution STATIC versus System Size (1 GPU)

Due to the overhead of a large number of kernel invocations and subsequent synchronizations, Solution CHUNKING cannot achieve better performance than Solution STATIC on a single-GPU system, although evaluating a larger data chunk with each kernel invocation can alleviate such overhead.

Solutions based on our framework, i.e, Solution TB-TASK and Solution WARP-TASK, consistently outperform other approaches even when running on a single GPU. For example,for a 512K atoms system, the average runtime per time step is 84.1s and 80.7s for Solution TB-TASK and Solution WARP-TASK, respectively. Compared to 93.6s for Solution STATIC, the performance improvement is about 11.3% and 16.0%, respectively.

Regarding this significant performance difference, our first guess was that Solution STATIC has to launch many more TBs than solutions based on our framework, therefore incurring in a large overhead. However, we experimentally measured that the extra overhead is relatively small. For example, when using a simple kernel,

launching it with $4,000$ TBs only incurs extra $26\mu s$ overhead, compared to launching it with 120 TBs, which does not justify the huge performance difference. Therefore, the only reason lies in how efficient CUDA can schedule TBs of different workload.

To investigate this issue, we create several workload patterns to simulate unbalanced load. To do this, we set up a balanced MD system of 512K atom in which all atoms are uniformly distributed[1]. Since the computation for each atom now involves equal amount of work, TBs consisting of computation of same amount of atoms should also take a similar amount of time to finish. Based on this balanced system, we create several computations following the patterns illustrated in Figure 9.5. In the figure, P0, $\cdots$, P4, represent systems of specific workload patterns. All patterns consist of a same number of blocks. In Pattern P0, each block contains 128 atoms, which is the workload for a TB (Solution STATIC), or in a task (Solution TB-TASK). Pattern P0 is actually the balanced system, and all blocks are of equal workload. For the rest of patterns, some blocks are labelled as *nullified*. Whenever a TB reads such a block, it either exits (Solution STATIC), or fetches another task immediately (Solution TB-TASK). In Solution STATIC, the CUDA scheduler is notified that a TB has completed and another TB is scheduled. In Solution TB-TASK, the persistent TB fetches another task from the execution queue.

Figure 9.6 shows the average runtime per time step for Solution STATIC and Solution TB-TASK, for different workload patterns. To our surprise the CUDA TB scheduler does not handle properly unbalanced execution of TBs. When the workload is balanced among all data blocks, i.e., Pattern P0, Solution TB-TASK is slightly worse than Solution STATIC due to the overhead associated with queue operations. However, for Pattern P1, P3, and P4, while Solution TB-TASK achieved reduced runtime, which is proportional to the reduction of the overall workload,

[1] We use the balanced system only to understand this behavior, we then return to the unbalanced Gaussian distributed system on the next section on multi-GPUs.

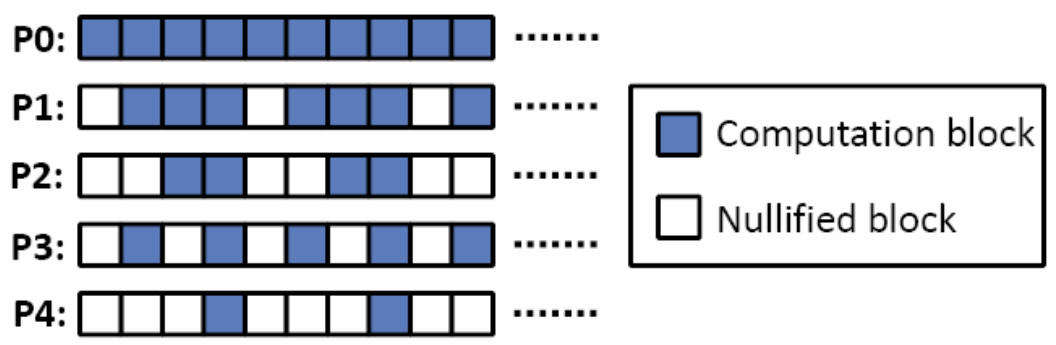


Figure 9.5: Workload Patterns

Solution STATIC failed to attain a similar reduction. For example, for Pattern P4, which implies a reduction of 75% workload over P0, Solution TB-TASK and Solution STATIC achieved runtime reduction of 74.5%, and 48.4%, respectively. Although not shown here, we also studied this issue with Solution WARP-TASK, and the results were very similar to Solution TB-TASK.

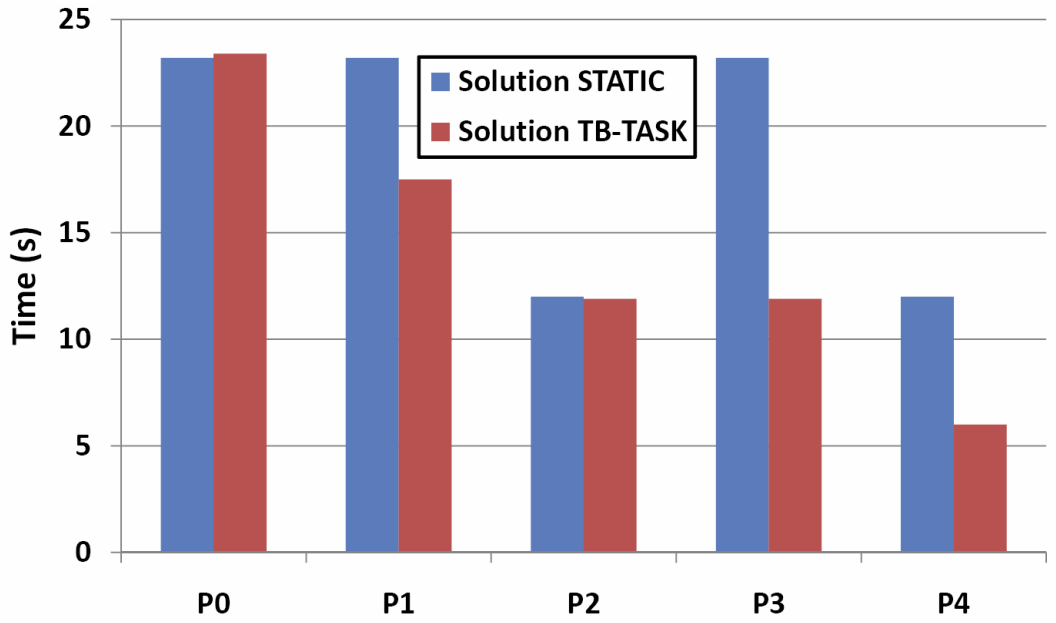


Figure 9.6: Runtime for Different Workload Patterns

To ensure that this observation is not only specific to our MD code, we conducted similar experiments with *matrixMul*, a NVIDIA's implementation of matrix multiplication included in CUDA SDK. The results also confirm our observation. This indicates that, when workload is unbalanced distributed among TBs, CUDA cannot schedule new TBs immediately when some TBs terminate, while the solutions based on our framework can utilize the hardware more efficiently.

9.4.2 Multi-GPU Scenario

Figure 9.7 shows the normalized speedup of the average runtime per time step over Solution STATIC, with respect to different system sizes, when all 4 GPUs are used in the computation. When the system size is small (32K), Solution RANDOM achieves the best performance (slightly over Solution STATIC), while other solutions incur relatively significant overhead associated multiple kernel launching (Solution CHUNKING), or queue operations (Solution TB-TASK and Solution WARP-TASK).

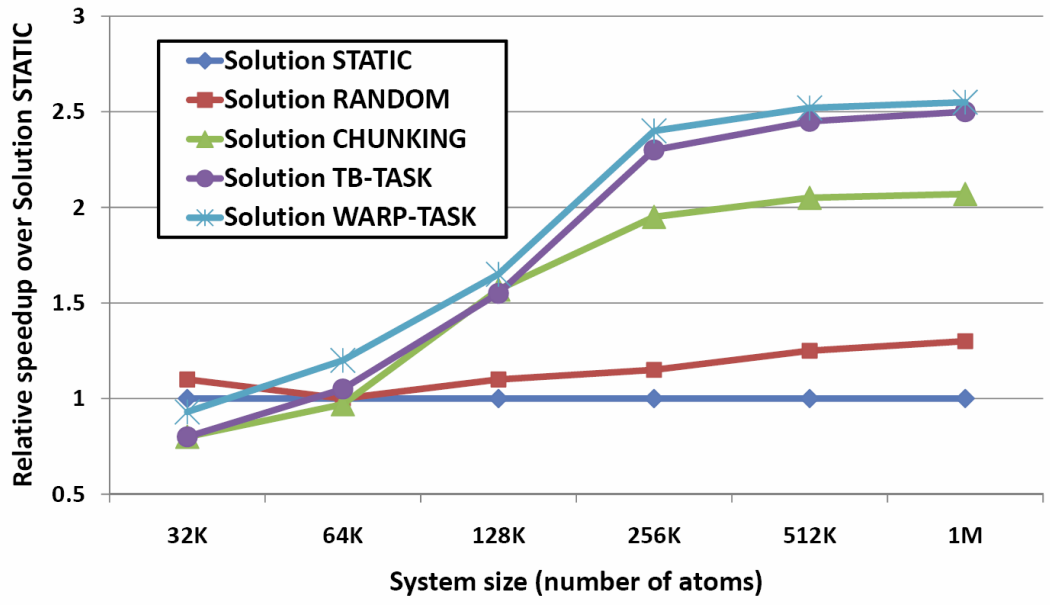


Figure 9.7: Relative Speedup over Solution STATIC versus System Size (4 GPUs)

As expected, when the system size becomes larger, we observe that solutions incorporated with dynamic load balance mechanisms remarkably outperform Solution STATIC. For example, for a system size of 512K atoms, Solution RANDOM, CHUNKING, TB-TASK, and WARP-TASK are 1.24×, 2.02×, 2.45×, and 2.53× faster than Solution STATIC, respectively. Especially, for these large system sizes, Solution TB-TASK and WARP-TASK achieve the top performance among all solutions, i.e., they are constantly about 1.2× faster than Solution CHUNKING, the next best approach.

Figure 9.8 shows the speedup with respect to the number of GPUs, for the simulation of a 512K atoms system. From the plot, we observe that, except Solution STATIC, other 4 approaches achieve nearly linearly speedup when more GPUs are used (they are so close that virtually there is only one curve visible in the figure for Solution RANDOM, CHUNKING, TB-TASK, and WARP-TASK).

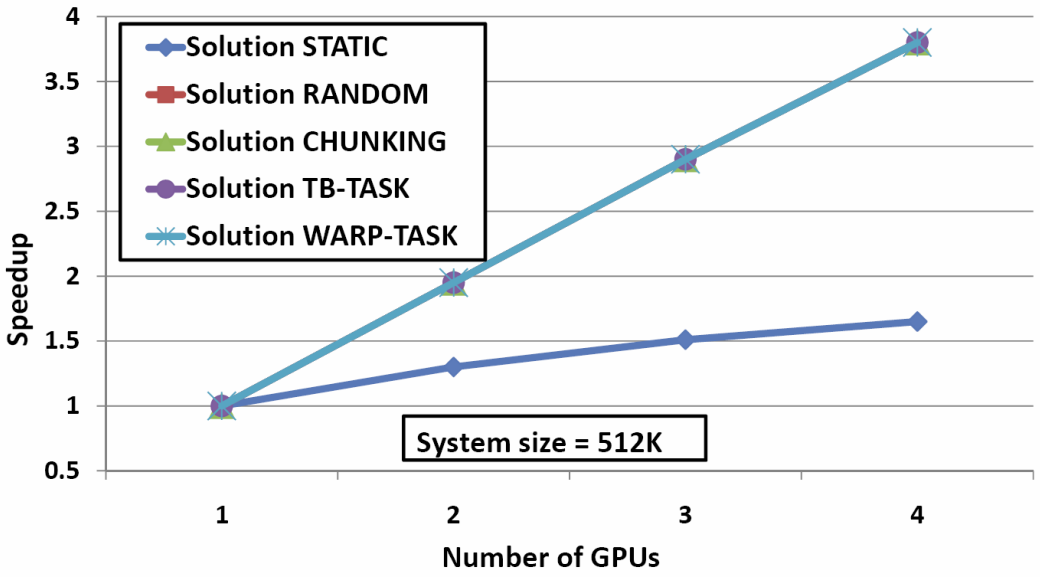


Figure 9.8: Speedup versus Number of GPUs

This observation is well explained by Figure 9.9, which shows the runtime of individual GPUs, when 4 GPUs are used in the computation of a 512K system.

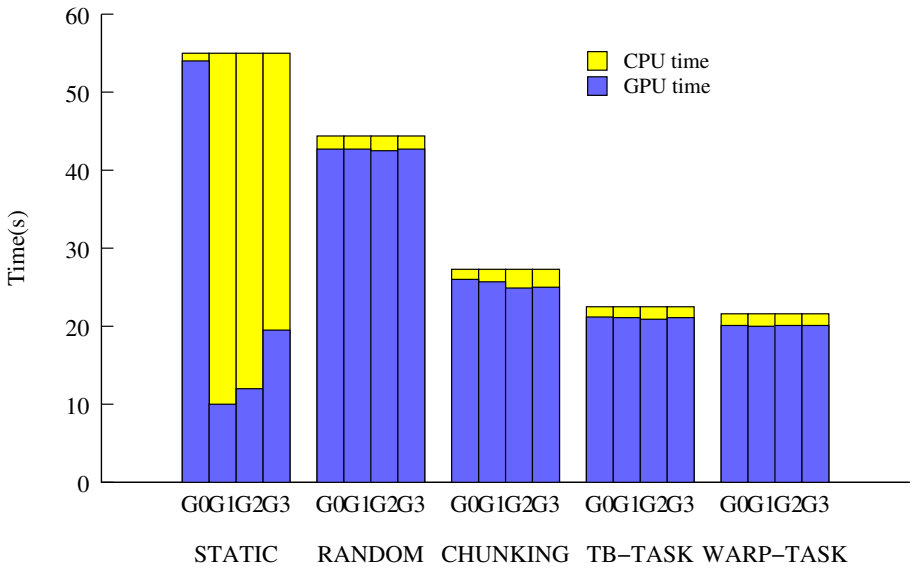


Figure 9.9: Dynamic Load on GPUs (512K Atoms)

Particularly, such timing information is presented for each individual GPU (labelled with G0-G3). In addition, we decompose the runtime into *CPU time* and *GPU time*. CPU time denotes the time spent on the corresponding host process for the host-device data transfer, position update, and communication with other host processes. GPU time denotes the time spent on the device for the force computation. In this way, the load balancing behavior is illustrated with the GPU times spent on different devices.

As illustrated, for Solution STATIC, the load among GPUs is extremely unbalanced. In contrast, other 4 approaches achieve good load balance. However, their absolute times vary. While Solution RANDOM is effective in terms of load balancing, it is 1.9x slower than Solution TB-TASK for large systems, i.e., 256K and up. As explained earlier, this is because it does not exploit the spatial locality, and therefore significantly increases the overall runtime.

Solution CHUNKING balances the load among GPUs by assigning fine-grained data chunks with different kernel invocations. However, it does not solve

the load imbalance issue within each data chunk; in a kernel invocation (for a data chunk), some TBs may need a longer time to finish than others, due to the unbalanced computation workload among them. Also it involves the overhead of kernel invocations and following synchronizations. One may argue that using larger data chunks can reduce such overhead. We investigated the effect of different sizes of data chunks, and discovered that using small data chunks, i.e., each of 15,360 (120x128) atoms, actually achieved the best performance; using larger data chunks introduced load imbalance among devices, and using smaller data chunks simply underutilized the computation power of devices.

In contrast, the solutions based on our fine-grained task-based framework both exploit the spatial locality, and achieves dynamic load balancing on individual devices, and among devices. Also, it is very easy to integrate our queue-based solution with existing CUDA code. For example, given a MD CUDA code and our framework module, a first version of a fine-grained task-based MD code was obtained within 2 hours.

On the other hand, since Solution WARP-TASK employs the execution scheme that optimally matches the GPU's architectural feature, we expected that it could outperform Solution TB-TASK remarkably, in terms of absolute performance and load balancing, We do see that Solution WARP-TASK achieves better load balancing than Solution TB-TASK. However, regarding the absolute performance, it only exhibits limited improvements over Solution TB-TASK, i.e., around 5%, which is below our initial expectation. A further examination of the force computation function reveals that, due to the specific algorithm used in our MD simulation, Solution WARP-TASK implies many more memory operations (of the same order of magnitude of N, the number of atoms in the system) to the array A, than Solution TB-TASK does. These extra memory operations offset the majority of the benefits of using a warp-level solution.

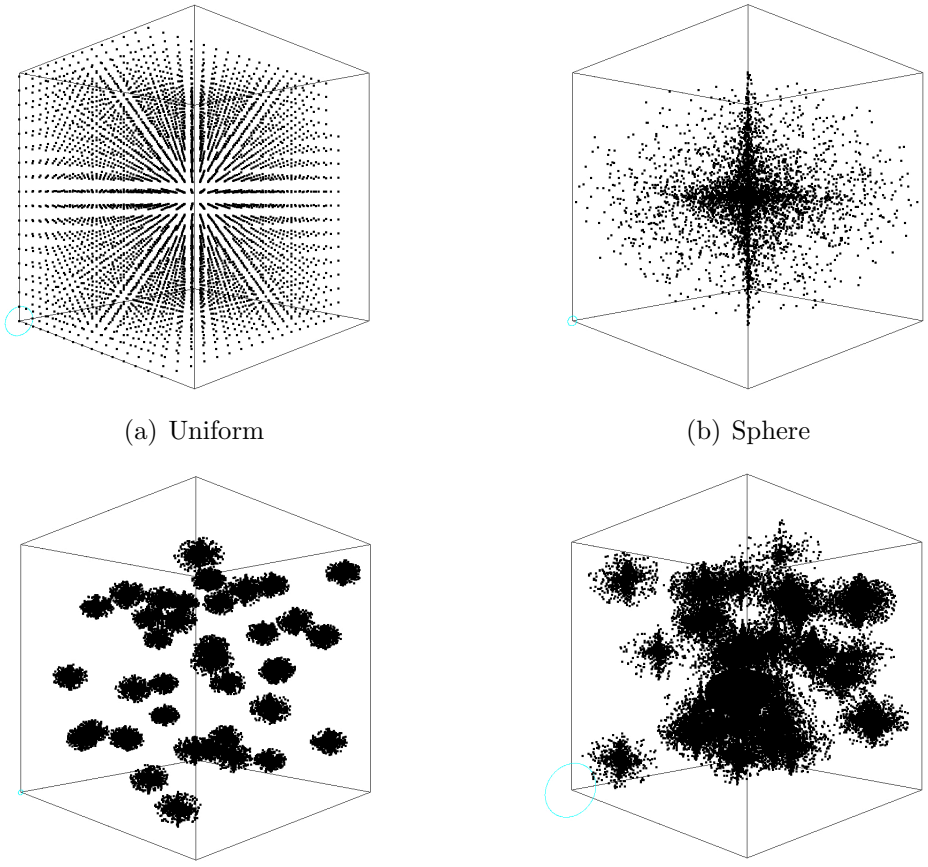
9.4.3 Effect of Different Distributions

So far, we have demonstrated that our fine-grained task-based execution framework can improve the performance of a MD application, for a particular atom distribution. But, will this framework also work for systems of other atom distributions? In this section, we evaluate the dynamic load balancing solutions based on our framework for the MD application with different workload distributions. In our experiments, we use synthetic systems of helium atoms, which are built by following 4 different atom distributions in a 3D space.

- **Uniform** distribution arranges atoms uniformly distributed in the system.
- **Sphere** distribution has a higher density in the center than in periphery [2]. The density decreases from the center to the periphery following a Gaussian curve.
- **Equal-sized cluster** distribution first partitions the system into clusters of equal number of atoms, where the centers of clusters are randomly generated. Then each cluster is built by following Sphere distribution.
- **Random-sized cluster** distribution also generates clusters of atoms. Unlike the Equal-size cluster distribution, for each cluster, both the center and the number of atoms in this cluster are randomly generated for the Random-size cluster distribution.

[2] Although the systems used in Section 9.4.1 and Section 9.4.2 share certain similarities with the systems built with Sphere distribution, they have different physical characteristics, e.g., atom layout, average distance among atoms, temperature, pressure, etc. Therefore, the results for the systems of Sphere distributions shown in this section cannot be simply compared to the results reported previously.

Figure 9.10 shows example systems of these distributions. It is clear that systems built with last three atom distributions have irregular computational load in the space.



(a) Uniform

(b) Sphere

(c) Equal-size cluster

(d) Random-size cluster

Figure 9.10: Example Synthetic Systems of Different Atom Distributions

We conduct the experiments on the 4-GPU system, where all 4 GPUs are used in the simulations.

We first investigate how different solutions behave for systems of a particular size, i.e., 256K-atom. Figure 9.11, 9.12, 9.13, and 9.14 show the average runtime per time step for all solutions (without Solution RANDOM) on each individual

GPU (labelled with G0-G3). Again, we decompose the runtime into *CPU time* and *GPU time*, and thus the load balancing behavior is illustrated with the GPU times spent on different devices. Solution RANDOM does achieve excellent load balancing among GPUs, however, it is usually much slower than other solutions. Therefore we will only describe its behaviors in text.

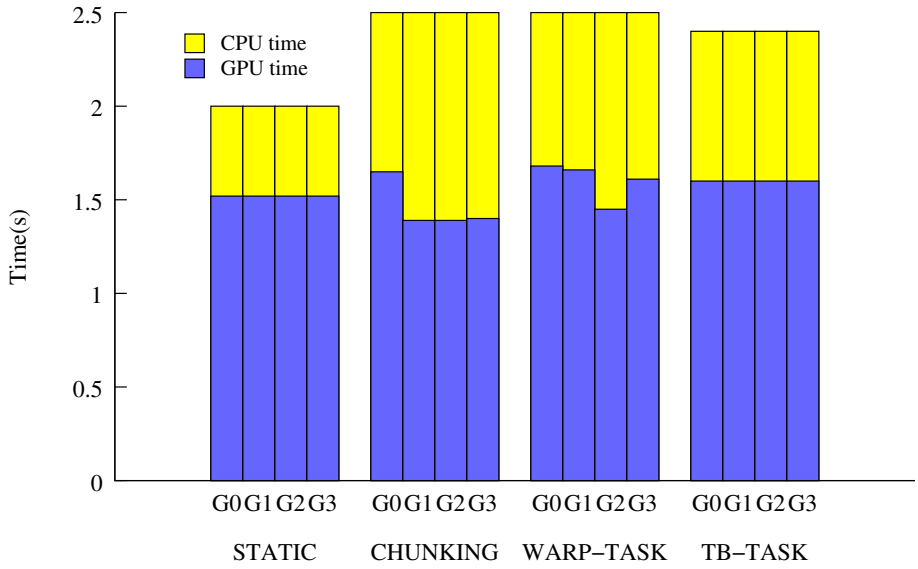


Figure 9.11: Runtime of Uniform Distribution

From these figures, it is clear that, in our experiments, the systems built with non-uniform atom distributions require much more computation time than the system built with the uniform atom distribution. This is because that with our particular Uniform distribution, there are only a few atoms in the space determined by the cutoff radius. However, for other distributions, in average, each atom may interact with up to hundreds of other atoms. Given the force computation is the most expensive part in the MD simulation, the number of force computations involved in various systems causes huge differences among them, in terms of the absolute performance.

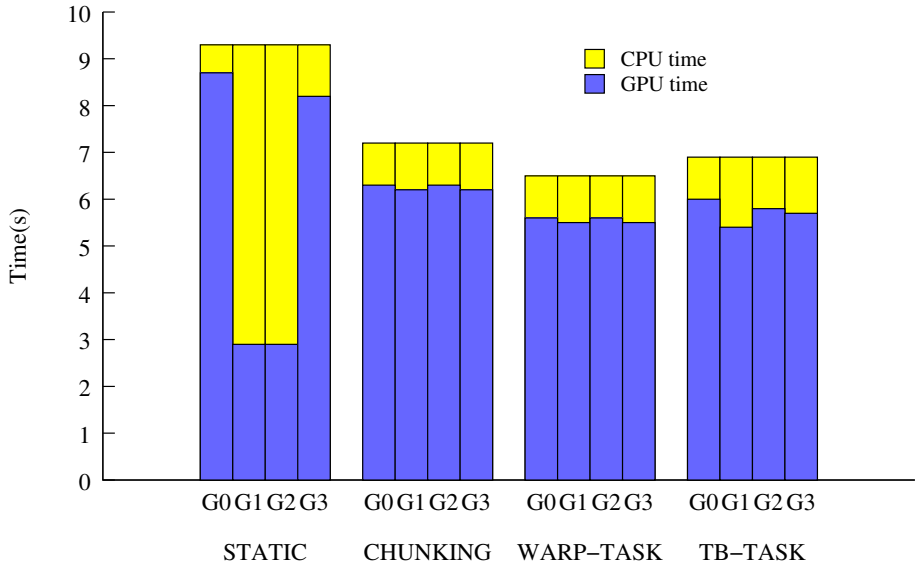


Figure 9.12: Runtime of Sphere Distribution

For the Uniform distribution, Solution STATIC achieves the best absolute performance and load balancing. It virtually balances the workload among GPUs perfectly with few overhead, while dynamic solutions suffer from the additional overhead due to the runtime scheduling. As we can see from Figure 9.11, this additional overhead is quite noticeable when the GPU time is small. However, for non-uniform distributed workload, dynamic solutions show their strengths, in terms of load balancing. Especially, for Solution WARP-TASK, the difference of GPU time among GPUs is within 3%, while such difference is up to 9% for Solution CHUNKING. Regarding the absolute performance, our fine-grained task-based solutions achieve up to 1.9× speedup over Solution STATIC. Both Solution WARP-TASK and Solution TB-TASK outperform Solution CHUNKING. Particularly, for Solution WARP-TASK, we see improvements of 11%-22% over Solution CHUNKING for different non-uniform distributions.

Not shown in the figure, Solution RANDOM also balances the workload among GPUs perfectly. However, because mixed atom data cause serious thread

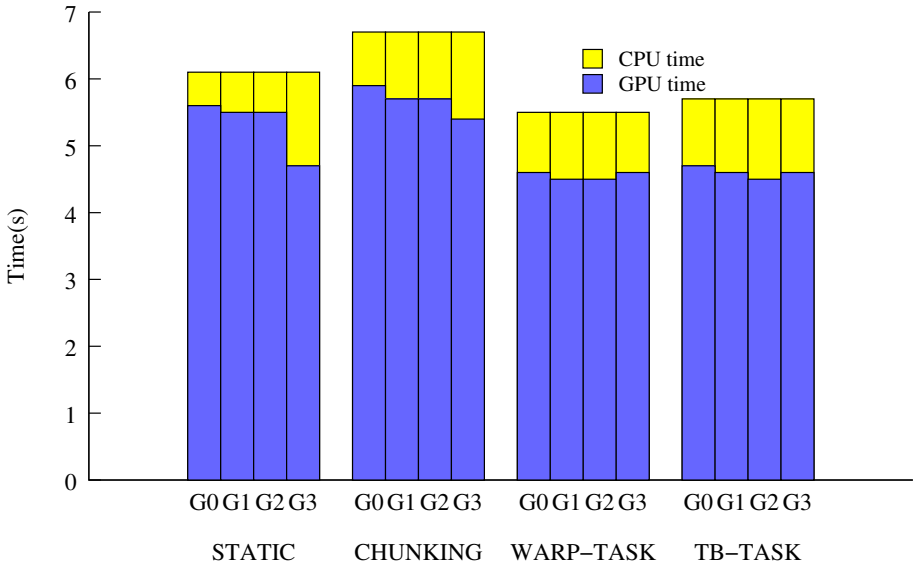


Figure 9.13: Runtime of Equal-size Cluster Distribution

divergence on each device, this solution is the slowest among all solutions, e.g., 6.9x slower than Solution STATIC for the Equal-size cluster distribution.

Figure 9.15, 9.16, 9.17, and 9.18 show the relative speedup of the average runtime per time step of all solutions (using 4 GPUs) over Solution STATIC, with respect to system sizes. Again, Solution RANDOM is not shown here due to its low performance. For the Uniform distribution, Solution STATIC still achieves the best absolute performance. However, other dynamic solutions reach comparable performance for large system sizes. This is because that the additional runtime scheduling overhead becomes relatively trivial, compared to the GPU time, when the system size increases. For other non-uniform distributions, our fine-grained task-based solutions achieve much better performance than Solution STATIC and Solution CHUNKING when large systems (i.e., 128K-atom and up) are used[3]. For all non-uniform distributions, Solution WARP-TASK constantly outperforms Solution

[3] Except for the Equal-size cluster distribution at the size of 128K-atom, where all dynamic solutions are worse than Solution STATIC.

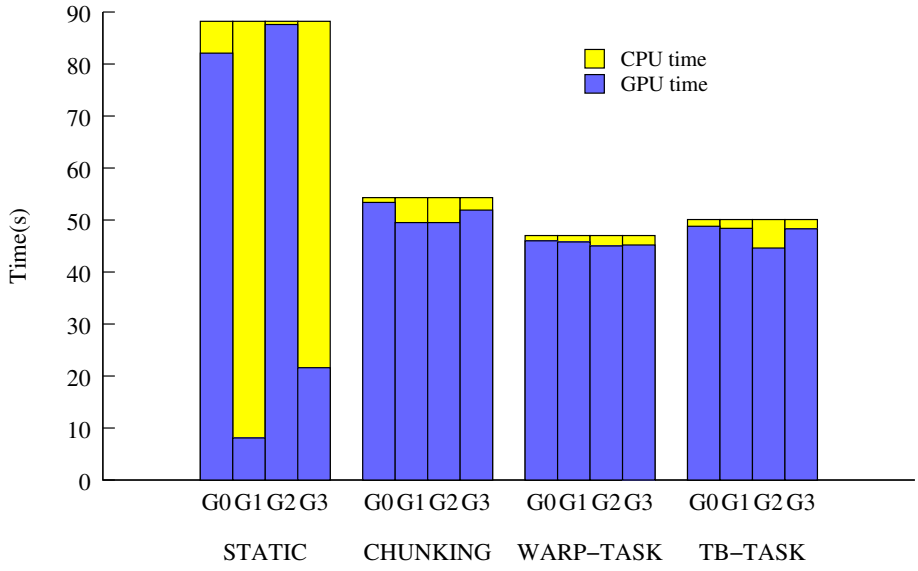


Figure 9.14: Runtime of Random-size Cluster Distribution

TB-TASK, for system sizes up to 512K-atom. However, the performance improvement becomes less significant when the system size increases. This in fact confirms our previous reasoning on why Solution WARP-TASK only exhibits limited benefits over Solution TB-TASK; when the system becomes large, the execution of those extra memory operations in Solution WARP-TASK will constitute a considerable portion of the overall runtime. In fact, when the system size reaches 1024K-atom, Solution TB-TASK achieves a similar or even better performance than Solution WARP-TASK. Note that the issue of extra memory operations is not directly related to our fine-grained task-based approach, but due to the particular algorithm used in our experiments. For algorithms/applications that can be parallelized without introducing significant amount of extra overhead, we expect that our warp-level execution scheme can generally outdo the TB-level execution scheme.

9.5 Summary

In this chapter, we first describe the platform used in our experiments and implementation issues of our fine-grained task-based framework. We then present the

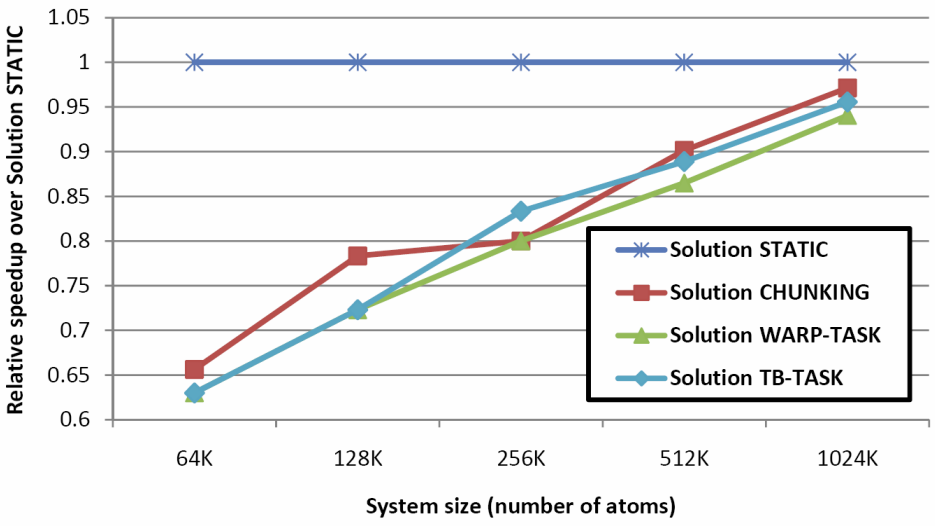


Figure 9.15: Relative Speedup: Uniform Distribution

benchmarking results for basic operations used in our framework. As a case study, a MD application with a particular unbalanced atom distribution is used to evaluate the effectiveness of our design. Experimental results with a single-GPU configuration show that our scheme can utilize the hardware more efficiently than the CUDA scheduler, for unbalanced problems. For multi-GPU configurations, our solution achieves nearly linear speedup, load balance, and significant performance improvement over alternative implementations based on the canonical CUDA paradigm. To investigate the effect of atom distributions to our framework, we also experiment with systems built with different distributions. The results show that, for non-uniform distributed workload, our solutions achieve better performance, in terms of both dynamic load balance and absolute performance, than other alternative approaches. Also, performance analyses reveal that, when utilizing this framework, the interaction between the task execution granularity and the particular algorithm can lead to significant impact upon the overall performance.

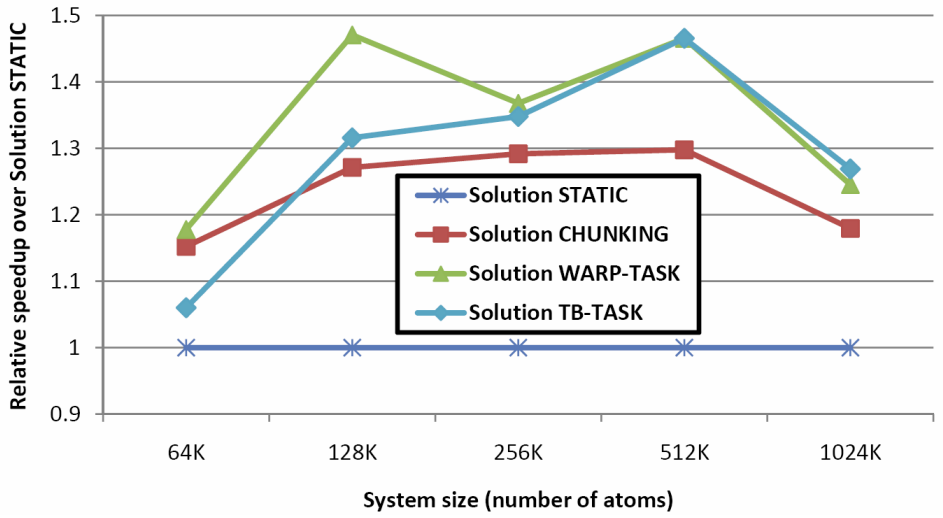


Figure 9.16: Relative Speedup: Sphere Distribution

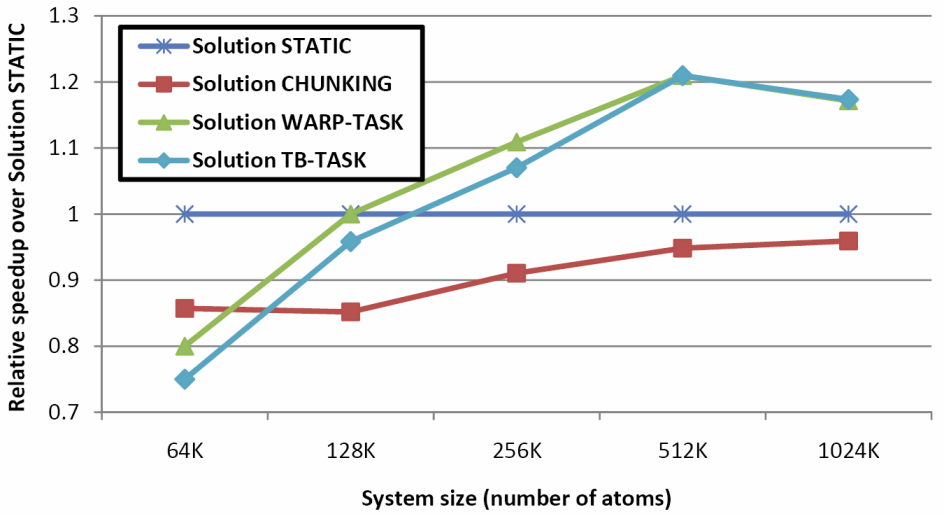


Figure 9.17: Relative Speedup: Equal-size Cluster Distribution

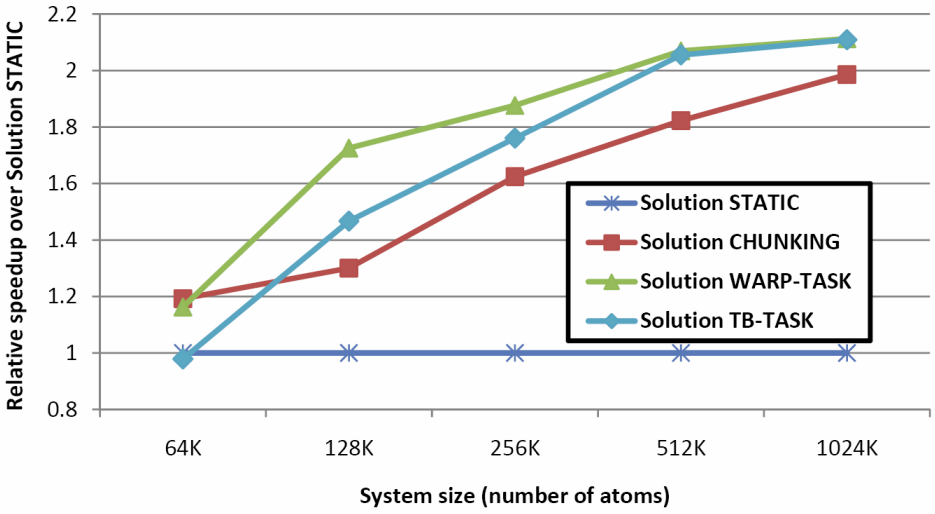


Figure 9.18: Relative Speedup: Random-size Cluster Distribution

Chapter 10

CONCLUSION

10.1 Summary

Due to the *Power wall*, the *Memory Wall*, and the *ILP Wall*, major processor manufacturers have turned to multi-core/many-core designs, in order to continue benefiting from the Moore's law. However, programming many-core systems is a new area for the software community; the majority of software community was very used to the idea of sequential programming, or at most working with very limited number of threads. The goal of this dissertation therefore is to use case studies to understand issues in designing and developing scalable, high-performance scientific computing algorithms for many-core architectures, get in-depth experience on programming and optimizing applications on those architectures, and then provide insights into implementation effort and performance behavior of optimizations and algorithmic properties for many-core architectures.

In this dissertation, we investigate the following two problem/architecture combinations as case studies,

- **Optimizing the Fast Fourier Transform for IBM Cyclops-64**

  Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFT is of great use in many scientific and engineering domains.

  We design and implement scalable high-performance parallel 1D and 2D FFT algorithms for the C64 architecture. We analyze the optimization challenges

and opportunities for FFT problems, and identify domain-specific features of the target problems and match them well with some key many-core architecture features. The impacts of various optimization techniques and effectiveness of the target architecture are addressed quantitatively.

We propose a performance model that estimates the performance of parallel FFT algorithms for an abstract many-core architecture, which captures generic features and parameters of several real many-core architectures. We derive the performance model based on cost functions for three main components of an execution: the memory accesses, the computation, and the synchronization. We evaluate our performance model on the C64 architecture. Experimental results from both simulations and the executions on the real hardware have verified the effectiveness of our performance model; our model can predict the performance trend accurately.

- **Exploring Fine-grained Task-based Execution on Graphics Processing Unit-enabled Systems**

  The computational power provided by many-core graphics processing units (GPUs) has been exploited in many applications. The programming techniques currently employed on these GPUs are not sufficient to address problems exhibiting irregular, and unbalanced workload. The problem is exacerbated when trying to effectively exploit multiple GPUs concurrently, which are commonly available in many modern systems.

  To solve the above problem, we propose a fine-grained, task-based execution framework for GPU-enabled systems. The framework allows computation tasks to be executed at a finer granularity than what is supported in existing GPU APIs such as NVIDIA CUDA. This fine-grained approach provides means for achieving efficient, and dynamic load balancing on GPU-enabled systems.

While scheduling fine-grained tasks enables good load balancing among multiple GPUs, concurrent execution of multiple tasks on each single GPU solves the hardware underutilization issue when tasks are small. Also, since this approach allows the overlapping executions of homogeneous/heterogeneous tasks, the programmers will have the flexibility to arrange their applications with fine-grained tasks and apply dataflow-like solutions to increase the efficiency of the program execution.

We implement our framework with CUDA. We evaluate the performance of the basic operations of this implementation with micro-benchmarks. We evaluate the solutions based on our framework with a MD application. Experimental results with a single-GPU configuration show that our solutions can utilize the hardware more efficiently than the CUDA scheduler, for unbalanced problems. For multi-GPU configurations, our solutions achieve nearly linear speedup, load balance, and significant performance improvement over alternative implementations based on the canonical CUDA paradigm.

10.2 Future Work

While the FFT optimization techniques presented in Chapter 4 could be helpful for developing other applications on C64-like many-core architectures, there are some important issues that can be considered as natural extensions to the current work. Firstly, although the absolute performance obtained on C64 is impressive, the efficiency of the current 1D FFT implementation on C64 is only 25% (20.72Gflops/80Gflops), compared to 76.39% (5.5Gflops/7.2Gflops) on Pentium 4 processor [49, 79]. More work has to be done to analyzes the program behavior on C64 and further improve the performance. Secondly, one of the architecture features of C64 has not been fully explored is the fast SPM associated with the corresponding thread unit. One possible way to employ SPM is to use it as an *extended* register

file, since it has very low access latency (2 cycles for load, 1 cycle for write). As we discussed in Chapter 4, using large work units may introduce serious register spilling, and then degrade the overall performance. Preliminary experiments, with explicit use of the SPM as a buffer to keep the intermediate computing results, showed promising results [157]. Another issue is to study larger FFT problem sizes when data cannot be fully stored in on-chip memories. In this case, data movement in the memory hierarchy, and computation have to be orchestrated carefully to overlap the communication with the computation.

The FFT performance model presented in Chapter 5 is an attempt to quantitatively analyze the interaction between existing algorithms and the emerging many-core architectures. The model can be further improved in several dimension as discussed below. As we mentioned in Section 5.3, the analysis of off-chip GM accesses is a natural extension. This is particularly important for the study of explicit data movement between levels of the memory hierarchy, which is used in many high performance FFT algorithms. It will be interesting to include analyses of such data movement, and thus verify the effectiveness of the existing FFT algorithms for many-core architectures. This performance model can be incorporated into an FFT computational framework, as a search engine to find suitable algorithms and optimal parameters for a given FFT problem. For example, as shown in Section 5.4, the performance model could identify the optimal number of PEs to be used for a given problem. Unlike an empirical search approach, by examining the properties of the algorithms and the architecture parameters, this performance model can potentially provide faster and more accurate solutions. Last, although our analysis presented in Chapter 5 is focused on the FFT algorithms, it will be interesting to investigate how the general methodology can be applied to other problems of statically defined communication and computation patterns, like matrix operations.

In our current design of the GPU task-based execution framework, to ensure the dependencies among tasks, we have to manually schedule the execution of tasks on the CPU side, according to their dependencies. An efficient mechanism to automatically enforce dependencies among tasks will greatly facilitate the design and development of fine-grained data-driven or event-driven applications. Another possible feature is to enable GPU to dispatch tasks to the CPU. The task could be remote data accesses, which cannot be performed by the GPU, or certain computations, which are too expensive on the GPU. These feature could further increase the applicability of our framework. Other future work includes extending the current design for GPU clusters, which have been introduced to several scientific sites. In this case, MPI, GA, or other alternatives should be integrated into our framework to take care of the distributed memory configuration. The recently announced NVIDIA Fermi architecture supports concurrent execution of multiple kernels. It would be interesting to evaluate the effectiveness of our framework with Fermi for imbalanced workload.

BIBLIOGRAPHY

[1] Folding@home. http://folding.stanford.edu.

[2] OpenMP. http://www.openmp.org.

[3] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[4] IEEE Std 1003.1-2004. The Open Group Base Specifications Issue 6, section 2.9. IEEE and The Open Group, 2004.

[5] R.C. Agarwal and J.W. Cooley. Vectorized Mixed Radix Discrete Fourier Transform Algorithms. In *Proceedings of the IEEE*, volume 75, pages 1283–1292, 1987.

[6] AMD. ATI Stream. http://www.amd.com.

[7] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, 2008.

[8] ARM. ARM11 MPCore Processor. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer Parry Husbands, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[10] M. Ashworth and A. G. Lyne. A segmented FFT algorithm for vector computers. *Parallel Computing*, 6:217–224, 1988.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009*, pages 863–874, Delft, Netherlands, 2009.

[12] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15:61–74, 1990.

[13] D. H. Bailey. FFTs in external or hierarchical memory. In *Proceedings of the Supercomputing 89*, pages 234–242, 1989.

[14] David H. Bailey. A high-performance fast Fourier transform algorithm for the Cray2. *Journal of Supercomputing*, 1:43–60, 1987.

[15] Ruud van der Pas Barbara Chapman, Gabriele Jost. *Using OpenMP: portable shared memory parallel programming.* MIT Press, 2008.

[16] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. Operating system issues for petascale systems. *SIGOPS Oper. Syst. Rev.*, 40(2):29–33, 2006.

[17] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[18] Robert G. Belleman, Jeroen Bdorf, and Simon F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13(2):103 – 112, 2008.

[19] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3):43 – 56, 1995.

[20] G.A. Bird, editor. *Molecular gas dynamics and the direct simulation of gas flows : GA Bird Oxford engineering science series: 42.* Oxford University Press, 1995.

[21] Shekhar Y. Borkar, Hans Mulder, Pradeep Dubey, Stephen S. Pawlowski, Kevin C. Kahn, Justin R. Rattner, and David J. Kuck. Platform 2015: Intel processor and platform evolution for the next decade, 2005.

[22] Michael Boyer, D. T., S. A., and K. S. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *IPDPS 2009*, pages 1–12, 2009.

[23] W. Briggs, L. Hart, R. Sweet, and A. O'Gallagher. Multiprocessor FFT methods. *SIAM J. Sci. Stat. Comput*, 8:27–42, January 1987.

[24] B.R. Brooks and Hodoscek M. Parallelization of Charmm for MIMD Machines. *Chemical Design Automation News*, 7(16):16–22, 1992.

[25] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *in the Proceedings of Supercomputing'00*, pages 51–51, 2000.

[26] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *in the Proceedings of HPCA*, pages 349–359, 2000.

[27] David A. Carlson. Using local memory to boost the performance of FFT algorithms on the CRAY-2 supercomputer. *J. Supercomput.*, 4:345–356, 1990.

[28] Daniel Cederman and Philippas T. On Dynamic Load Balancing on Graphics Processors. In *GH 2008*, pages 57–64, 2008.

[29] Soumen Chakrabarti, James Demmel, and Katherine Yelick. Modeling the benefits of mixed data and task parallelism. In *SPAA'95*, pages 74–83, New York, NY, USA, 1995. ACM.

[30] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-Power CMOS Digital Design. *Solid-State Circuits, IEEE Journal of*, 27:473–484, 1992.

[31] Shuai Che, Michael B., Jiayuan M., David T., Jeremy W. S., and Kevin S. A performance study of general-purpose applications on graphics processors using cuda. *JPDC*, 68(10):1370–1380, 2008.

[32] Edmond Chow and David Hysom. Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical report, Lawrence Livermore National Laboratory, May 2001.

[33] T.W. Clark, McCammon J.A., and Scott L.R. Parallel Molecular Dynamics. In *SIAM PP'91*, pages 338–344, March 1991.

[34] ClearSpeed. ClearSpeed CSX700. http://www.clearspeed.com/products/cs_advance_e700/.

[35] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.

[36] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[37] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[38] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

[39] Bill Dally. Computer architecture in the many-core era. In *Key note in the 24th Intl. Conf. on Comput. Design (ICCD 2006)*, 2006.

[40] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Workshop on Modeling, Benchmarking, and Simulation*, Madison, Wisconsin, June 2005.

[41] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In *Proceedings of the 20th International Symposium on High Performance Computing Systems and Applications (HPCS'06)*, May 2006.

[42] Juan del Cuvillon, Weirong Zhu, Ziang Hu, and Guang R. Gao. TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. In *Fifth Workshop on Massively Parallel Processing*, page 265, Denver, Colorado, USA, April 2005.

[43] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, Yorktown Heights, NY, 2007. IBM Confidential.

[44] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part II. memory organization, the A-switch, and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, NY, 2007. IBM Confidential.

[45] Erich Elsen, Mike Houston, V. Vishal, Eric Darve, Pat Hanrahan, and Vijay Pande. N-body simulation on gpus. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 188, New York, NY, USA, 2006. ACM.

[46] F. Ercolessi. A molecular dynamics primer. http://www.fisica.uniud.it/ ercolessi/md/.

[47] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC'04*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Zhe Fan, Feng Qiu, and Arie E. Kaufman. Zippy: A framework for computation and visualization on a gpu cluster. *Comput. Graph. Forum*, 27(2):341–350, 2008.

[49] FFTW. FFT Benchmark Results. http://www.fftw.org/speed/.

[50] R. Fine, G. Dimmler, and C. Levinthal. Fastrun: A special purpose, hardwired computer for molecular simulation. *Proteins: Structure, Function, and Bioinformatics*, 11:242253, 1991.

[51] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[52] Basilio B. Fraguela, Yevgen Voronenko, and Markus Puschel. Automatic tuning of discrete fourier transforms driven by analytical modeling. In *PACT'09: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[53] Franz Franchetti, Yevgen Voronenko, and Markus Puschel. FFT program generation for shared memory: SMP and multicore. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 51–51, 2006.

[54] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14(3):437 – 449, 2006.

[55] Daan Frenkel and Berend Smit, editors. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, Inc., Orlando, FL, USA, 1996.

[56] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[57] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[58] Angelopoulos G. and Pitas I. Parallel implementation of 2-d FFT algorithms on a hypercube. In *Proc. Parallel Computing Action, Workshop ISPRA*, 1990.

[59] Pat Gelsinger. Keynote at Intel Developer Forum, Spring 2004. http://www.intel.com/pressroom/archive/speeches/gelsinger20040219.htm.

[60] GNU. GCC: the GNU compiler collection. http://gcc.gnu.org.

[61] GNU. The GNU Binutils. http://sources.redhat.com/binutils/.

[62] Dominik Gödeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H.M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33(10-11):685 – 699, 2007.

[63] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[64] Naga K. Govindaraju, Avneesh Sud, Sung-Eui Yoon, and Dinesh Manocha. Interactive visibility culling in complex environments using occlusion-switches. In *I3D'03*, pages 103–112, New York, NY, USA, 2003. ACM.

[65] David J. Griffiths. *Introduction to Electrodynamics, 3/E.* Addison-Wesley, 1999.

[66] David Jeffery Griffiths. *Introduction to quantum mechanics.* Pearson Prentice Hall, 2005.

[67] Liang Gu and Xiaoming Li. DFT Performance Prediction in FFTW. In *LCPC 2009: Languages and Compilers for Parallel Computing*, pages 140–156, Newark, DE, 2009.

[68] Liang Gu, Xiaoming Li, and Jakob Siegel. An empirically tuned 2d and 3d fft library on cuda gpu. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 305–314, New York, NY, USA, 2010. ACM.

[69] M. Guevara, C. Gregg, and Skadron K. Enabling task parallelism in the cuda scheduler. In *PEMA 2009*, 2009.

[70] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[71] Pawan Harish and Narayanan P.J. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[72] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th edition.* Morgan Kauffman, San Francisco, CA, 2007.

[73] John L. Hennessy and David A. Patterson. *Computer organization and design (3rd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[74] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *the Proceedings of Supercomputing'00*, pages 50–50, 2000.

[75] Maurice Herlihy. Wait-free synchronization. *ACM TPLS.*, 13(1):124–149, 1991.

[76] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. Clawhmmer: A streaming hmmer-search implementatio. In *SC'05*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[77] Ziang Hu, Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Optimization of dense matrix multiplication on IBM Cyclops-64: Challenges and experiences. In *Euro-Par*, pages 134–144, 2006.

[78] IBM. The CELL project at IBM research. http://www.research.ibm.com/cell/.

[79] Intel. Intel Microprocessor export compliance metrics. http://www.intel.com.

[80] Intel. Intel develops tera-scale research chips. http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm, September 2006.

[81] Intel. Single-chip Cloud Computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm, 2009.

[82] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, NASA Ames Research Center, 1999.

[83] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9:449–500, 1990.

[84] S. L. Johnsson and R. L. Krawitz. Cooley-tukey FFT on the connection machine. *Parallel Computing*, 18(11):1201–1221, 1992.

[85] S.L. Johnsson and D. Cohen. Computational arrays for the discrete Fourier transform. 1981.

[86] Mark Joselli, Marcelo Z., Esteban C., Anselmo M., Aura C., Regina L., Luis V., Bruno F., Marcos d' O., and Cesar P. Automatic Dynamic Task Distribution between CPU and GPU for Real-Time Systems. In *CSE'08*, pages 48–55, 2008.

[87] Khronos. OpenCL. http://www.khronos.org.

[88] P. Kongetira. A 32-way multithreaded SPARC processor. Hot Chips 16, 2004.

[89] G. Kurian, J. Miller, J. Psota, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *PACT 2010*, Vienna, Austria, 2010.

[90] Rubin H. Landau, Manuel José Páez, and Cristian C. Bordeianu. *A survey of computational physics: introductory computational science*. Princeton University Press, 2008.

[91] John Lennard-Jones. On the determination of molecular fields. In *Proceedings of the Royal Society of London.*, volume 106, pages 441–462, 1924.

[92] Yan Li, Li Zhao, Haibo Lin, Chow Alex Chunghen, and Diamond Jeffrey R. A performance model for fast fourier transform. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[93] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. M. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[94] Charles Van Loan. *Computational framework for the fast Fourier transform.* SIAM, Philadelphia, 1992.

[95] T. Maruyama. SPARC64 VI: Fujitsu's next generation processor. in Mircroprocessor Forum 2003, 2003.

[96] J. Andrew McCammon, Bruce R. Gelin, and Martin Karplus. Dynamics of folded proteins. *Nature*, 267:585 – 590, 1977.

[97] Adam Moerschell and John D. Owens. Distributed texture memory in a multi-gpu environment. In *GH'06*, pages 31–38, New York, NY, USA, 2006. ACM.

[98] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(18), April 19, 1965.

[99] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[100] Tim Murray. Personal communication, June 2009.

[101] M. Mller, C.and Strengert and T. Ertl. Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Computing*, 33(6):406 – 419, 2007. Parallel Graphics and Visualization.

[102] Newlib. Newlib. http://sources.redhat.com/newlib/.

[103] John Nickolls, Ian Buck, Michael G., and Kevin S. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[104] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. HPCA*, 20(2):203–231, 2006.

[105] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Transactions on Computers*, 36(5):581–591, 1987.

[106] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka. Bandwidth intensive 3-d FFT kernel for GPUs using CUDA. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[107] Akira Nukada, Yasuhiko Ogata, Toshio Endo, and Satoshi Matsuoka. Bandwidth intensive 3-d fft kernel for gpus using cuda. In *SC'08*, pages 1–11, Piscataway, NJ, USA, 2008.

[108] Nvidia. CUDA. http://www.nvidia.com.

[109] Nvidia. NVIDIA CUDA Programming Guide 2.3, 2009.

[110] National Oceanic and Atmospheric Administration. NOAA‘s Powerful New Supercomputers Boost U.S. Weather Forecasts. http://www.noaanews.noaa.gov/stories2009/20090908_computer.html, 2009.

[111] Yasuhiko Ogata, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *IPDPS*, pages 1–10, 2008.

[112] Tomohiro Okuyama, Fumihiko I., and Kenichi H. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *ISPA'08*, pages 284–291, 2008.

[113] Stephen Olivier, Jan Prins, Jeff Derby, and Ken Vu. Porting the gromacs molecular dynamics code to the cell processor. *Parallel and Distributed Processing Symposium, International*, 0:370, 2007.

[114] Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A. Kendall. General portable SHMEM library for high performance computing. In ACM, editor, *SC2000: High Performance Networking and Computing*, pages 148–149, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, November 2000. ACM Press and IEEE Computer Society Press.

[115] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[116] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[117] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *SC'08*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[118] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, 1995.

[119] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005.

[120] Shankar Ramaswamy, Sachin S., and Prithviraj B. A framework for exploiting task and data parallelism on distributed memory multicomputers. *TPDS.*, 8(11):1098–1116, 1997.

[121] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *CF'08: Proceedings of the 5th conference on Computing frontiers*, pages 273–282, New York, NY, USA, 2008. ACM.

[122] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and W. M. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP'08*, pages 73–82, 2008.

[123] D. MacDonald S. L. Johnsson, R.L. Krawitz and R. Frye. A radix 2 FFT on the connection machine. In *Proceedings of Supercomputing 89*, pages 809–819, 1989.

[124] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of openmp compiler for an smp cluster. In *In EWOMP 99*, pages 32–39, 1999.

[125] Dana Schaa and David Kaeli. Exploring the multiple-gpu design space. In *IPDPS'09*, pages 1–12, Washington, DC, USA, 2009.

[126] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh V. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[127] O. Shacham, Z. Asgar, H. Chen, A. Firoozshahian, R. Hameed, C. Kozyrakis, W. Qadeer, S. Richardson, A. Solomatnikov, D. Stark, M. Wachs, and M. Horowitz. Smart Memories Polymorphic Chip Multiprocessor. In *Design Automation Conference*.

[128] Hongzhang Shan, Jaswinder P. Singh, Leonid Oliker, and Rupak Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, 29(2):167–186, 2003.

[129] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 1–12, New York, NY, USA, 2007. ACM.

[130] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *International Parallel Processing Symposium*, pages 92–101, 1991.

[131] Lorna Smith and Mark Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2-3/2001):83–98, 2001.

[132] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62:229–248, 1991.

[133] W. Smith and T. R. Forester. DL_POLY_2.0: A general-purpose parallel molecular dynamics simulation package. *Journal of Molecular Graphics*, 14(3):136 – 141, 1996.

[134] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005.

[135] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.

[136] M. Strengert, M. Magallón, D. Weiskopf, Stefan Guthe, and T. Ertl. Large volume visualization of compressed time-dependent datasets on gpu clusters. *Parallel Comput.*, 31(2):205–219, 2005.

[137] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5(1-2):197–210, 1987.

[138] D. Sylvester and K. Keutzer. Microarchitectures for systems on a chip in small process geometries. In *the IEEE*, pages 467–489, 2001.

[139] Makoto Taiji, Noriyuki Futatsugi, Tetsu Narumi, Atsushi Suenaga, Yousuke Ohno, Naoki Takada, and Akihiko Konagaya. Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations. In *in Supercomputing*. ACM Press, 2003.

[140] Guangming Tan and Guang R. Gao. A study of parallel betweenness centrality algorithm on a many-core architecture, 2007. CAPSL Technical Memo 76.

[141] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanovic. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Design Automation Conference*, Anaheim, CA, 2010.

[142] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An

exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[143] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads*. Ph.D. dissertation, McGill, May 1999.

[144] Parimala Thulasiraman, Kevin B. Theobald, Ashfaq A. Khokhar, and Guang R. Gao. Multithreaded algorithms for the fast fourier transform. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 176–185, 2000.

[145] Tilera. Tilera Manycore Processors. http://www.tilera.com/products/processors, 2010.

[146] S. Toyoda, H. Miyagawa, K. Kitamura, T. Amisaki, E. Hashimoto, H. Ikeda, A. Kusumi, and N. Miyakawa. Development of md engine: High-speed accelerator with parallel processor design for molecular dynamics simulations. *Journal of Computational Chemistry*, pages 185–199, 1999.

[147] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[148] J. A. van Meel, A. Arnold, D. Frenkel, Portegies, and R. G. Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[149] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of 2007 International Solid-State Circuits Conference*, Feb. 2007.

[150] Ioannis E. Venetis and Guang R. Gao. Optimizing the LU benchmark for the Cyclops-64 architecture, 2007. CAPSL Technical Memo 75.

[151] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159(1):98, Jul 1967.

[152] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC 2008*, pages 1–11, 2008.

[153] Žark Cvetanović. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM J. Res. Dev.*, 31(4):435–451, 1987.

[154] WIKIPEDIA. Computational science. http://en.wikipedia.org/wiki/Computational_science.

[155] S. Williams, J. Shalf, L. Oliker, P. Husbands, S. Kamil, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20, 2006.

[156] Alexander Wolfe. Intel clears up post-tejas confusion. http://www.crn.com/it-channel/18842588, May 2004.

[157] Liping Xue, Long Chen, Ziang Hu, and Guang R. Gao. Performance tuning of the fast fourier transform on a multi-core architecture. In *MULTIPROG'08: the 1st Workshop on Programmability Issues for Multi-Core Computers*, Goteborg, Sweden, 2008.

[158] Juekuan Yang, Yujuan Wang, and Yunfei Chen. Gpu accelerated molecular dynamics simulation of thermal conductivities. *J. Comput. Phys.*, 221(2):799–804, 2007.

[159] Katherine Yelick. Programming models for petascale to exascale. In *Key note in the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, 2008.

[160] Yuan Zhang, Weirong Zhu, Fei Chen, Ziang Hu, and Guang R. Gao. Sequential consistency revisited: The sufficient conditions and method to reason consistency model of a multiprocessor-on-a chip architecture. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN2005)*, page 12, Innsbruck, Austria, 2005.