OPTIMIZING THE FAST FOURIER TRANSFORM ON A MANY-CORE ARCHITECTURE

by

Long Chen

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical & Computer Engineering

Winter 2008

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

OPTIMIZING THE FAST FOURIER TRANSFORM ON A MANY-CORE ARCHITECTURE

by

Long Chen

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Gonzalo R. Arce, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____

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

Approved: _____

Carolyn A. Thoroughgood, Ph.D.
Vice Provost for Research and Graduate Studies

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 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, Juan del Cuvillo, Andrew Russo, Geoffery Gerfin, Junmin Lin, Guangming Tan, Fei Chen for providing insightful comments and helpful advice on my research. Their kind assistance and friendship have also made my life in the U.S. easy and colorful.

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 a M.S. This thesis, thereupon, is dedicated to them for their infinite love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The rapid revolution in microprocessor chip architecture due to multi-core and many-core technology is presenting an unprecedented challenge to the application developers as well as system software designers: how to exploit the performance potential due to such architectures effectively and efficiently?

In this thesis, an in-depth study on optimizing the Fast Fourier Transform (FFT) on a many-core architecture is presented. The IBM Cyclops-64 (C64) chip architecture, the case study in this thesis, is a many-core chip architecture consisting of 160 thread units, associated memory banks and an interconnection network that connects them together in a shared memory organization.

The study presented in this thesis demonstrates how many-core architectures, like the C64, could be used to achieve a scalable high-performance implementation of FFT both in 1D and 2D cases. The thesis also analyzes the optimization challenges and opportunities, including problem decomposition, load balancing, work distribution, and data-reuse, together with the exploiting of the C64 architecture features such as the massive parallelism, explicit multi-level memory hierarchy and large register files.

The main contributions of this thesis include: 1) the development of scalable high-performance parallel FFT implementation on C64; 2) the study demonstrates that successful optimization for C64-like many-core architectures requires a careful analysis that can identify certain domain-specific features of a target application (e.g. FFT) and match them well with some key many-core architecture features; 3)

the optimization procedure, assisted with the hand-tuned process, provides quantitative evidence on the importance of each optimization identified in 2) and valuable experience toward establishing an effective programming methodology for C64-like many-core architectures.

Chapter 1

INTRODUCTION

As the traditional uni-processor architectures are no longer 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 architecture has higher power-efficiency, improved heat dissipation, better memory latency tolerance, and many other benefits. Many researchers think that this architecture is going to become the mainstream for the parallel computing in the future.

1.1 Introduction to 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 [60]. 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 uni-processor has followed Moore's law for 40 years. This made it relatively easy to improve the performance of the traditional software. Most users just simply relied on the increasing capabilities and speed of single processors to get free performance improvement.

However, this frequency increase could no longer be sustained due to the power consumption, heat dissipation, memory latency, diminishing returns on finding more instruction level parallelism (ILP), and other issues. Due to the above fundamental limits, the era of applying Moore's law to the traditional uni-processor designs appears to be coming to an end.

Instead of devoting the entire die to a single and complex processor, the many-core chip architecture design integrates a number of simple processors/cores on a single die. Provided that hundreds of millions, even billions, of transistors can be fabricated into a single chip die, it is believed that these architectures can extend the benefits of Moore's law.

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

- Since the power consumption of a single core drops significantly with the reduction in frequency, many-core architectures can provide a power-efficient way to achieve performance by running multiple cores with moderate clock rate [16, 71].

- 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.

- Many-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.

- Many-core chip architectures naturally support thread-level parallelism, which is expected to be exploited in future applications and multiprocessor-aware operating systems and environments [42].

- 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 [73].

While the dual-core and quad-core microprocessors, which are called multi-cores, start dominating 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 into a single chip [11, 23]. For example, Intel has announced its 80-core teraflops research chip [77]. Another example is the IBM Cyclops-64 chip architecture, which supports 160 hardware thread units in one chip. Researchers also predicted that 1000-core chip would be achieved when 30nm technology is available [7].

1.2 Motivation

Many-core architectures are becoming increasingly attractive platforms for high performance computing. For the first time, many-core architectures demand that the mainstream software community engage in parallel processing, which until now was reserved for the rarefied field of supercomputing. 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.

1.2.1 Parallel Programming Approaches

There are many existing parallel programming approaches. Some of them are in common use, such as explicit thread programming, data parallel, message passing, hybrid, etc. However, many-core architectures bring unprecedented challenges for parallel programming, and it seems those existing parallel programming approaches may not be sufficient or as effective in the many-core era.

The explicit thread programming approaches, such as POSIX threads (Pthreads) [3], usually provide a rich set of thread control routines for programmers. By devoting sufficient time and effort, programmers may be able to parallelize the problem and achieve good performance. Since these approaches offer no encapsulation or modularity, however, programming hundreds, even thousands, threads definitely would be a nightmare for the vast majority of programmers.

Another way is to simply treat many-core chips as conventional symmetric multiprocessors (SMPs), thus OpenMP [1] could become a promising choice. However, OpenMP works at a fairly coarse-grained level and it does not guarantee to make the most efficient use of shared memory systems. 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 Origin2000 [22], while it is 4 GB/s for Cyclops-64 [28]. 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 should cease 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. Furthermore, high performance computing systems usually 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) [13, 14, 44, 67]. Also, it is not clear how to smoothly make the interaction between different programming models [18, 69].

On the other hand, Message Passing Interface (MPI) [2], the current dominant programming model for parallel scientific programmings, 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. Also, programmers have to explicitly deal with decomposing data, mapping tasks, and performing synchronization, the massively increasing number of cores may introduce additional challenges to programmers. Nevertheless, MPI might have a place between chips in a many-chip system, as it combines both data transfer and synchronization in a single message.

Because of the above issues, while researchers believe that many-core processors are going to become the mainstream in the future, they 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. Furthermore, there are few resources on experiences of application development on real many-core architectures.

1.2.2 Case Study: Fast Fourier Transform

In this thesis, we report our experience in the implementation and optimization of the Fast Fourier Transform (FFT) on the IBM Cyclops-64 (C64) many-core architecture. This thesis demonstrates how many-core architectures, like C64, could

be employed to achieve a scalable high performance implementation of FFT both in 1D and 2D cases.

FFT 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 [21, 57], etc. Due to such importance, it has been studied extensively on various architectures [57]. FFT was chosen because it represents commonly used techniques in a wide variety of applications and has performance characteristics typical of many parallel scientific applications. In addition, it has small code segments whose behavior we can understand and directly track to specific architectural characteristics. Algorithms like FFT are both computation-intensive and memory-intensive due to the large amount of data involved in the underlying applications. Moreover, some applications require performing these algorithms on large data sets in real time. Thus a high performance computing engine is highly desirable for this domain. On the other hand, most FFT-like algorithms have inherent parallelism [41, 62]. Therefore, it seems that these algorithms can potentially take advantage of the emerging many-core architectures to achieve better performance.

1.2.3 Cyclops-64 Architecture

The C64 chip, the "node" in the IBM petaflop supercomputer that may employ thousands of such chips, is used in our study. The C64 chip employs the many-core design by integrating 160 thread units (cores), 80 double-precision floating-point units (FPUs), 160 32KB SRAM banks, and a 96×96 crossbar on a single chip. This large number of cores provide enormous computation power to exploit the inherent parallelism of FFT. All thread units within a chip are connected by a 16-bit signal bus, which provides a means to efficiently implement barriers. This is critical for problems that involve intensive synchronization operations. Furthermore, the C64

instruction set architecture (ISA) features a large number of exotic instructions, for example, Bit Gather, which greatly facilitate the index computing in FFT. Other components include an A-switch and B-switch that are used to form the 3D mesh for a C64 system with multiple chips, 4 DRAM controllers, GigaBit Ethernet controller and other I/O devices. A C64 chip provides massively on-chip parallelism, massive on-chip memory bandwidth, a large register file for each thread unit and an explicit multiple level memory hierarchy without data cache.

To fully exploit the architectural features and understand issues in developing high-performance application on C64, we employed the TiNy-Threads library [26], an efficient C64 thread virtual machine and its programming interface, with a familiar Pthreads like API. While studying and comparing other programming approaches, such as OpenMP and MPI, on C64 will be meaningful, they are out of the scope of this thesis. We hope that our experience of the FFT study on C64 may benefit application/system software developers on other many-core platforms.

1.3 Related Research

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 [34, 51, 66] 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 [52] and mesh architectures [68].

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 [72]. Two different scheduling strategies for single-vector FFTs and three different approaches to the multiple FFTs are discussed in [12]. 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 [9, 63, 79]. Moreover, by using the Kronecker notation, the work in [50] 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 [10]. The work in [15] shows how to use local memory to compute the FFT efficiently on CRAY-2. The issue of data re-use is also discussed in [4, 8]. Further, an excellent review of various sequential and parallel DFT algorithms proposed in the literature until 1991 appeared in [57]. Two dataflow-based multithreaded FFTs [76] are presented to exploit the features of EARTH [75], a fine-grain dataflow architecture.

FFTW [33] 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 [32]. 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 [70] 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 formulas 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 [31]. The generated parallel FFT codes using OpenMP as well as Pthreads are evaluated on several CMP/SMP architectures.

Performance evaluation and analysis of several scientific computing kernels, including FFT, on the IBM Cell architecture [46], a heterogeneous multi-core architecture, are reported in [80]. Results demonstrate the tremendous potential of the

Cell architecture for scientific computations in terms of both raw performance and power efficiency.

1.4 Main Contributions

In this thesis, optimizing the FFT on Cyclops-64, a many-core architecture, is presented. This thesis also analyzes the optimization challenges and opportunities, including problem decomposition, load balancing, work distribution, and data-reuse, together with the exploitation of the C64 architecture features such as the explicit multi-level memory hierarchy without data cache and large register files. The main contributions are listed below.

- A scalable high-performance parallel FFT implementation is designed and developed on C64, for both 1D FFT and 2D FFT cases. For both cases, we demonstrate that the performance of this implementation scales nearly linearly up to 128 threads, when the problem size is large enough [1]. The absolute performance measurements are 20.72Gflops and 20.00Gflops for the 1D FFT and 2D FFT, respectively.

- For the various techniques applied in the optimization process, detailed analysis and discussions are presented. During our study of 1D and 2D FFT on C64, we found out that domain-specific knowledge is very important, some time critical. To achieve better performance, we carefully analyzed the FFT algorithm features, identified 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. We proposed optimization methods to address these issues and demonstrated how we explored the corresponding C64 architecture features. We set up the experiments based

[1] While the input data should be large enough to fully utilize the parallelism provided by the C64 architecture, they should fit into the on-chip SRAM.

on the analysis, and found out the optimal parameters that matched the C64 architecture features.

- The impacts of various optimization techniques and the effectiveness of the target architecture are addressed quantitatively. The optimization procedure itself provides valuable information for optimizing other applications on C64-like architectures. The procedure, together with the experimental results, also provide valuable experience toward establishing an effective programming methodology for those many-core architectures.

Some of the above optimizations had to be performed manually before the C64 compiler could automatically generate more efficient code. Our study provided valuable guidance to the compiler designers on what should be done in C64 compiler optimizations. For instance, the C64 compiler was enhanced by adding the memory segment aware instruction scheduling, such that the FFT code generated by the compiler is very close to the tediously hand-tunned code.

1.5 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 presents the background knowledge of many-core architectures, parallel programming approaches, and FFT. Chapter 3 first presents the C64 chip architecture and its major features, it then introduces the experimental infrastructure that is employed for the study. Chapter 4 presents how we identify a set of important issues by analyzing the FFT algorithm features, and propose optimization methods to address these issues and match them with some key C64 architecture features. Chapter 5 concludes the thesis with some possible issues to be developed further.

Chapter 2

BACKGROUND

This chapter introduces the background knowledge and related work for this study. Section 2.1 discusses the limits of the traditional uni-processor designs, the current designs of the multi-core architectures, and the design of many-core architectures. Section 2.2 gives a brief introduction to the existing parallel programming approaches. Section 2.3 presents a short introduction to the FFT algorithm.

2.1 From Uni-processor to Multi-core and Many-core

Since Moore's law was coined in 1965, the CPU frequency of the traditional uni-processor has followed it for 40 years. This made it relatively easy to improve performance of the traditional software. Most users just simply received performance improvement by running the existing codes on a faster machine. However, this frequency increase could no longer be sustained due to the following problems.

First, and perhaps the most important, the increasing power density is an unsolvable problem for conventional uni-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 [36]. Due to this so-called "Power Wall", several of the next generation processors, such as the Tejas Pentium 4 processor from Intel, were canceled or redefined [81].

Second, 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.

Next, advances in IC technology allow the 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 nullify the speed increases of the transistors.

Finally, most uni-processors are designed to exploit ILP in programs. ILP approaches could overlap the execution of instructions and improve performance without affecting the standard uni-processor programming model. While exploiting ILP was the primary focus of processor designs for a long time, ILP can be quite limited, or hard to exploit, in many applications [43]. This is known as the "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 uni-processor designs appears to be coming to an end. On the other hand, by integrating a number of simple processors (cores) on a single die. the multi-core/many-core architectures are believed to be able to extend the benefits of Moore's law.

With the above arguments, the entire microprocessor industry has dramatically shifted to the multi-core/many-core technology [49, 6, 11, 19, 46, 55, 58].

2.1.1 Two Types of Multi-Core Architecture Designs

Because multi-core architecture is a large and diverse field, and much of the field is in its youth, researchers from both the industry and academia are actively

exploiting the design space of multi-core chip design. Roughly, these designs can be classified into two types [35, 54].

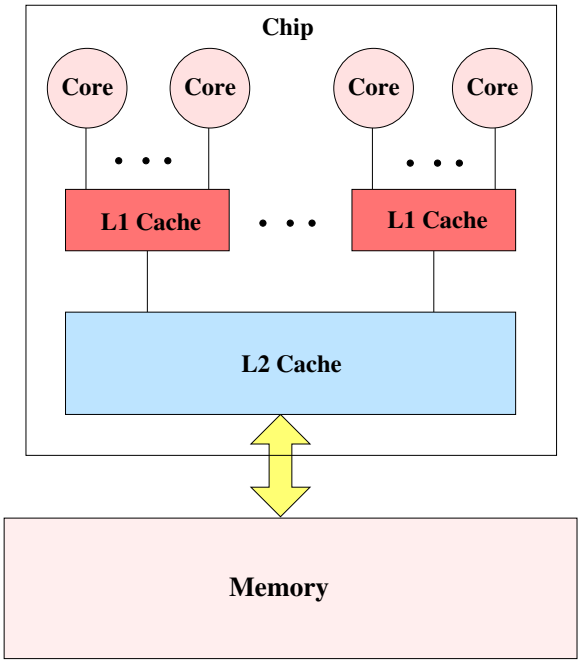


Figure 2.1: Hierarchical Multi-core Design With Heavy Cores and Multiple Level Cache

Courtesy: Weirong Zhu

- The first type simply reuses the existing uni-processor designs and "glues" these "heavy" cores into a single chip with only minor changes, usually an outermost-level shared cache. Figure 2.1 shows a such multi-core design, which consists of a number of heavy cores that communicate through cache. This is the approach that is currently taken by major microprocessor manufacturers. For example, Intel Core Duo Processors [17], AMD Opteron dual-Core processors [5], IBM Power5 dual-Core processors [53], and many others that are now available on the market, can be attributed to this type of multi-core design.

- Although the first type seems to be natural and simple, some other multi-core designs take a different approach by completely redesigning the chip architecture to explore the parallel architecture design space and search for the most suitable chip architecture models. Figure 2.2 shows a multi-core design with many simple cores and on-chip memory modules connected through an on-chip interconnection network. The IBM Cell processor [39, 40], IBM Cyclops-64 chip [27, 28, 29], Intel Tera-Scale researcher chip [49], and some others are now exploring this type of multi-core design.

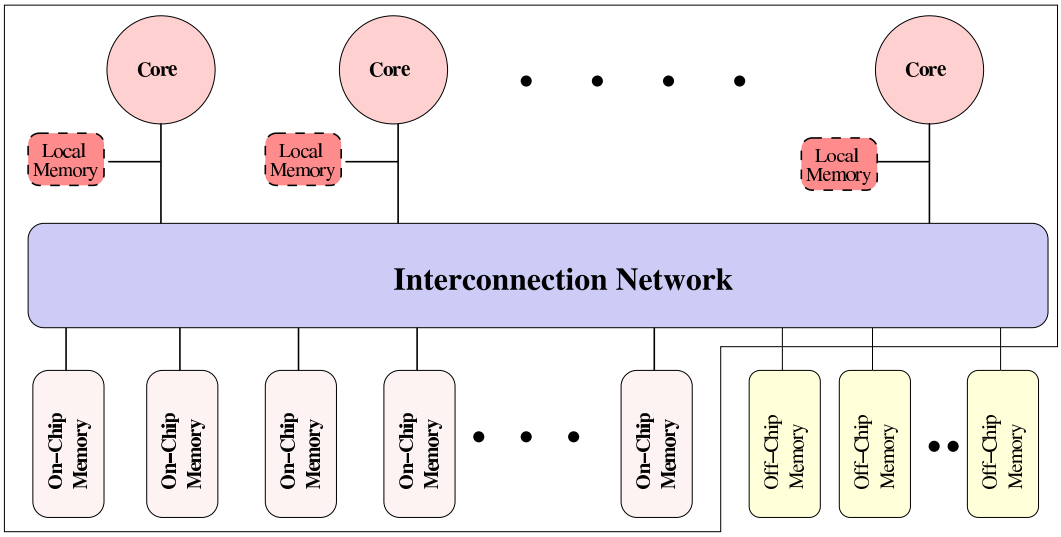


Figure 2.2: Multi-core Design with Many Simple Cores and on-chip Memory Modules

Courtesy: Weirong Zhu

While the multi-core chips, namely dual-core and quad-core microprocessors, start dominating the market of servers and personal computers, both the industry and academia are exploiting the design space of the future multi-core architectures by integrating an even larger number of cores (32s and beyond) into a single chip, which is called many-core [11, 23]. For example, Intel has announced its 80-core teraflops research chip [77]. Another example is the IBM C64 chip architecture, which

supports 160 hardware thread units/cores in one chip. Researchers also predicted that 1000-core chip would be achieved when 30nm technology is available [7].

The IBM C64 chip architecture, the target architecture employed in this thesis, is a homogeneous many-core architecture that belongs to the second type of multi-core/many-core designs. With such a chip design, massive on-chip parallelism are provided through a large number of on-chip cores.

2.2 Parallel Programming Approaches

As we discussed in Section 2.1, microprocessor manufacturers are embarking on the many-core roadmap. On the other side, the software community is faced with a severe dilemma. Until now, most software has been developed with a single processor in mind and it needs to be parallelized to take advantage of the new breed of many-core computers. As a result, progress in how to easily harness the computing power of many-core architectures is in great demand.

The majority of the software community is familiar with 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 cannot exploit the power of multiple processors.

As a matter of fact, people have been doing parallel programming for many years on vector machines, clusters, SMPs, etc. Different approaches have been proposed and utilized. It would be meaningful to review some here.

2.2.1 Explicit Thread Programming

Explicit thread programming approaches include Pthreads, Sun Solaris threads, Windows threads, and other native threading APIs. Explicit thread programming is a low-level mechanism for managing concurrent computations that share a single address space. It is primarily designed to express the natural concurrency that is present in most applications, and to improve the performance and response time. 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.

However, because explicit threading is an inherently low-level API that mostly requires multiple steps to perform simple threading tasks, it requires considerable effort from the programmer's side. Due to this issue, developers have been increasingly looking for other, simpler, alternatives.

2.2.2 OpenMP

Jointly defined by a group of major computer hardware and software vendors, OpenMP 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. OpenMP is a compiler-based threading approach and it gives programmers a simple and flexible interface for developing parallel applications on shared-memory systems.

OpenMP simplifies parallel application development by hiding many of the details of thread management and thread communication behind a simplified programming interface. Developers specify parallel regions of code by adding pragmas to the source code. By judicious use of these pragmas, a single-threaded program can be made multithreaded, without recourse to APIs or environment variables.

In addition, these pragmas communicate other information such as properties of variables and simple synchronization.

OpenMP uses a *fork-join* model of parallel execution. The programmer must specify the start and end of each parallel region. All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered. Then the master thread creates (fork) a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate (join), leaving only the master thread.

For applications characterized by easy data decomposition, clean functional decomposition, and simple needs for locking and mutual exclusion, OpenMP is a powerful, simple, and sufficient means of threading programs. Because of this, OpenMP has become a very successful model for developing shared-memory parallel applications.

However, it is important to recognize that OpenMP does has several weaknesses. Firstly, OpenMP is not meant for non-shared memory systems by itself. This excludes the possibility of directly utilizing OpenMP on a large number of parallel systems. Although there have been several efforts to run OpenMP programs on distributed memory systems [47, 56, 59, 64], it clear that more study is necessary to address the issues on program porting and performance. Secondly, OpenMP does not analyze code correctness, and so it cannot detect this dependency. As a result, it will generate code that generates an incorrect result. Likewise, data races and other threading problems can lead to generation of code that does not work correctly. Programmers do have to make their code thread-safe by themselves. Thirdly, OpenMP works at a fairly coarse-grained level and it does not guarantee to make the most efficient use of shared memory.

2.2.3 Message Passing

The 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. First, the message passing model fits well on parallel supercomputers and clusters of workstations, which are composed of separate processors 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. Second, message passing offers a full set of functions for expressing parallel algorithms, providing the control is not found in data-parallel and compiler-based models. Third, by giving the programmer explicit control of data locality, message passing usually can achieve an effective use of the memory hierarchy of modern architectures, thus achieving a satisfactory performance. The message passing programming model has been effectively standardized by MPI. Due to its portability and stability, MPI has become the *de facto* standard for writing parallel programs on distributed systems.

Unfortunately message passing is generally a difficult method to program. It puts a burden on the programmer to manage their distributed data structure. These data structures have to be explicitly partitioned so the entire application must be parallelized in order to work with the partitioned data structures. There is no incremental path to parallelizing an application. Also, message passing has a fundamental issue that the latency of software passing message is typically high, which makes it hard to achieve low overhead on small messages.

2.3 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.

2.3.1 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 2.1.

$$X(k) = \sum_{n=0}^{N-1} x(n)\omega_N^{kn}, \quad 0 \leq k \leq N-1 \tag{2.1}$$

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

$$x(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(k)\omega_N^{-kn}, \quad 0 \leq n \leq N-1 \tag{2.2}$$

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:

$$\omega_N^{k+N/2} = -\omega_N^k \tag{2.3}$$

$$\omega_N^{k+N} = \omega_N^k \tag{2.4}$$

where Equations 2.3 and 2.4 are known as the *symmetry* property, and the *periodicity* property, respectively.

2.3.2 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:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)\omega_N^{kn}, \qquad k = 0, 1, \cdots, N-1 \\ &= \sum_{n \in even} x(n)\omega_N^{kn} + \sum_{n \in odd} x(n)\omega_N^{kn} \\ &= \sum_{m=0}^{N/2-1} x(2m)\omega_N^{2mk} + \sum_{m=0}^{N/2-1} x(2m+1)\omega_N^{k(2m+1)} \end{aligned} \tag{2.5}$$

With the substitution of $\omega_N^2 = \omega_{N/2}$, , Equation 2.5 can be expressed as,

$$\begin{aligned} X(k) &= \sum_{m=0}^{N/2-1} f_1(m)\omega_{N/2}^{km} + \omega_N^k \sum_{m=0}^{N/2-1} f_2(m)\omega_{N/2}^{km} \\ &= F_1(k) + \omega_N^k F_2(k), \qquad k = 0, 1, \cdots, N-1 \end{aligned} \tag{2.6}$$

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 2.4, 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 2.3, the original DFT could be expressed as,

$$\begin{aligned} X(k) &= F_1(k) + \omega_N^k F_2(k), \quad k = 0, 1, \cdots, N/2-1 \\ X(k+\frac{N}{2}) &= F_1(k) - \omega_N^k F_2(k), \quad k = 0, 1, \cdots, N/2-1 \end{aligned} \tag{2.7}$$

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 $2(N/2)^2 + N/2 = N^2 + N/2$ 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 above is usually referred to as the radix-2 Cooley-Tukey FFT algorithm [20], and the specific computation is known as a butterfly operation, which is shown in Figure 2.3. Generally speaking, the implementation of the above re-

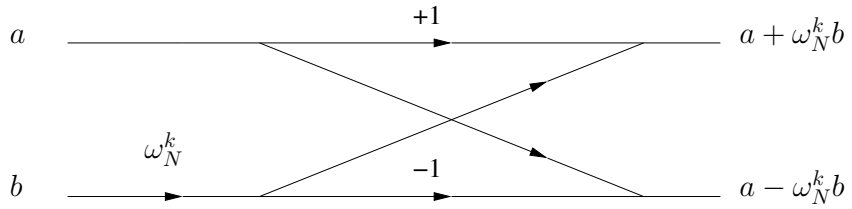


Figure 2.3: Cooley-Tukey Butterfly Operation

cursive FFT algorithm introduces non-negligible recursion overhead, thus it is not

favored. Another approach is to employ the iterative implementation. The iterative algorithm subdivides the resulting subproblems 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 2.4 and Figure 2.5, which illustrate the decimation of the eight-point sequence.

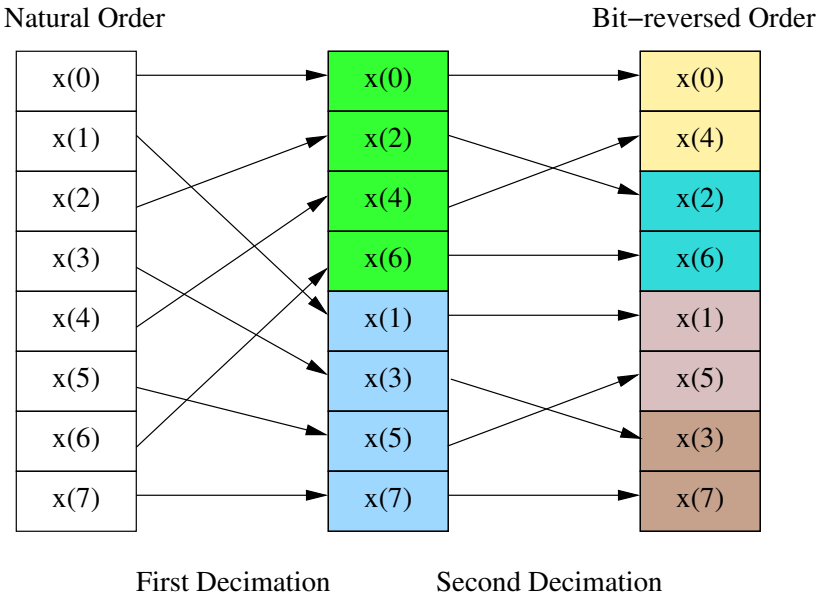


Figure 2.4: Bit Reversal of 8-point Data

In the Cooley-Tukey algorithm, this permutation is performed before the butterfly computations. Figure 2.6 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

$(n_2n_1n_0)$		$(n_0n_2n_1)$		$(n_0n_1n_2)$
(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 2.5: Binary Representation of the Bit Reversal

the computation is decomposed through 3 stages of butterfly operations. We use the iterative approach in this thesis.

2.3.3 Multidimensional 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 multidimensional DFT of a multidimensional array $x(n_1, n_2, \cdots, n_d)$ that is a function 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 multidimensional 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

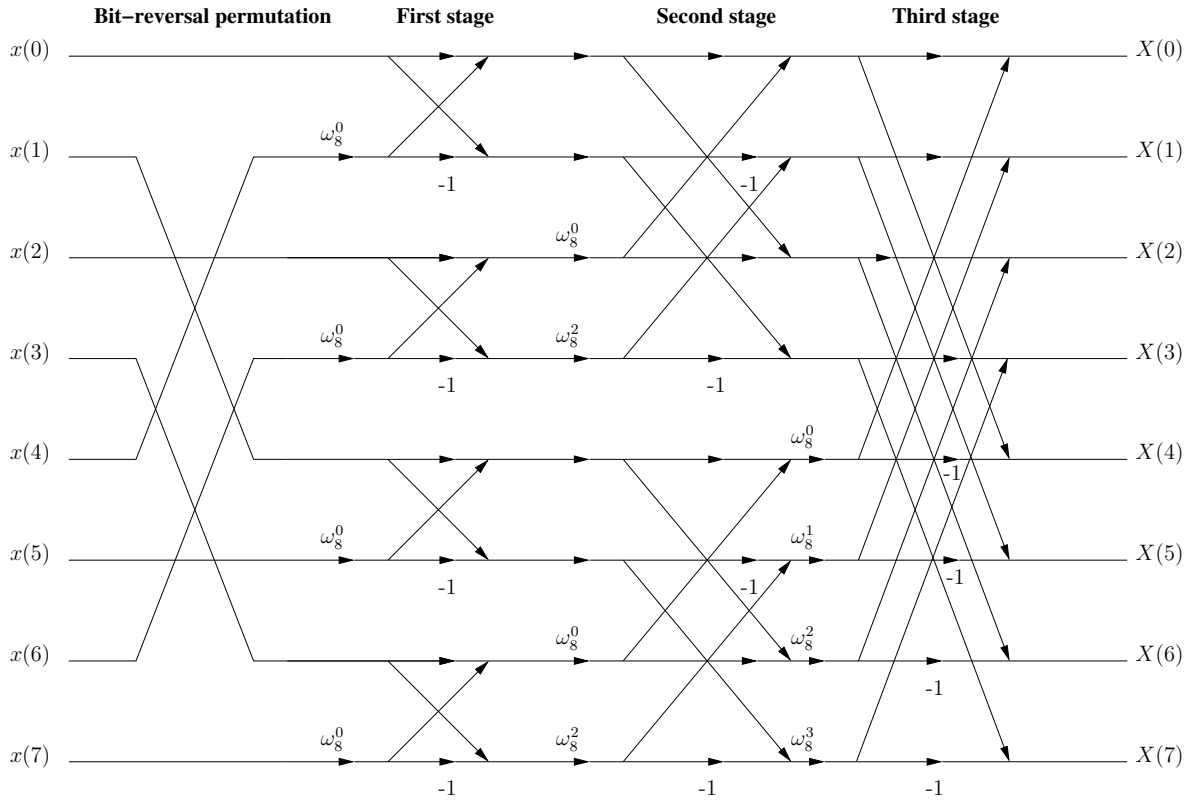


Figure 2.6: 8-point Cooley-Tukey FFT Example

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 the data, interleaved with data transpose steps. This method is called the multidimensional 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.

Chapter 3

CYCLOPS-64 CHIP ARCHITECTURE

The chapter presents the platform used in this thesis. Section 3.1 introduces the IBM C64 chip architecture and its major features. Section 3.2 presents a brief overview of the C64 system software toolchain.

3.1 Introduction to Cyclops-64 Architecture

The C64 chip, shown in figure 3.1, is the core computation engine of the C64 supercomputer system, which consists of thousands of such chips connected through a 3D-mesh network. One C64 chip features massive parallelism with 80 64-bit processors, each consisting 1 double-precision floating point unit (FPU) and 2 thread units (TUs). Each TU is a single-issue, in-order RISC processor operating at a moderate clock rate (500MHz). It has 64 64-bit registers and 32 KB SRAM. Other on-chip components include 16 shared instruction caches (ICs), 4 DRAM controllers, A-Switch, B-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 C64 architecture has a segmented memory space, including the scratchpad (SP) memory, 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 SP memory (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 TUs on the chip of a uniform latency, the SPM is regarded as the fast local memory

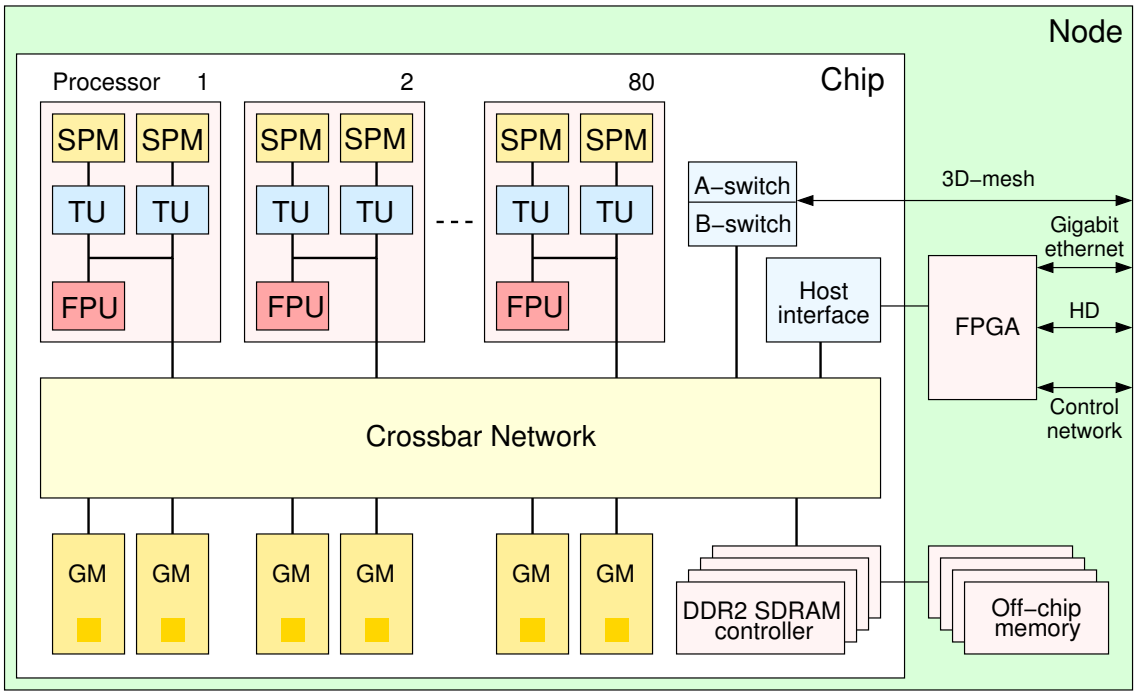
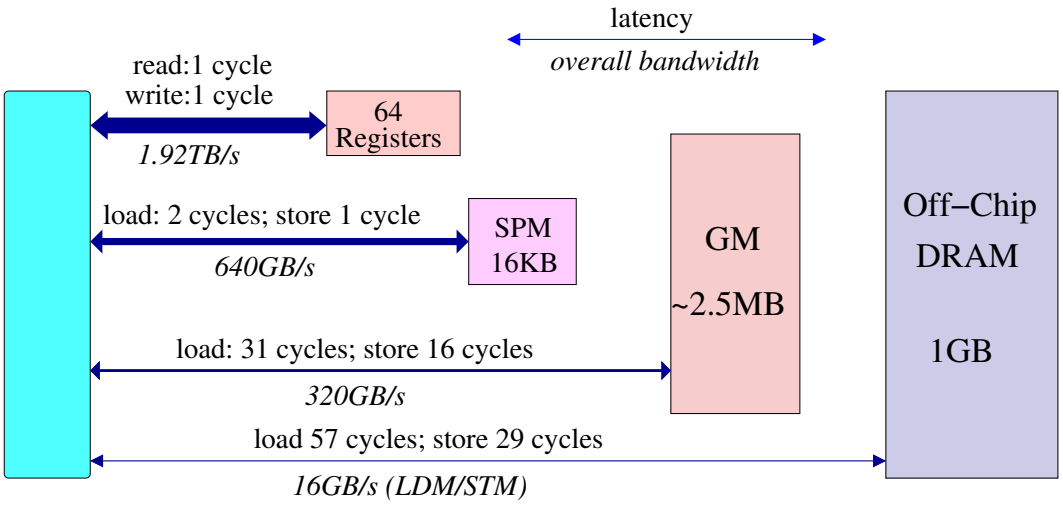


Figure 3.1: Cyclops-64 Chip Architecture

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

of the corresponding thread unit. Figure 3.2 shows the latency and bandwidth for accessing different segments in the C64 memory hierarchy.

The large number of cores on C64 chips provide enormous computation power to exploit the inherent parallelism of FFT. The large register files further increase this power by providing tremendous bandwidth. Moreover, all TUs within a chip are connected by a 16-bit signal bus, which provides a means to efficiently implement barriers. This is critical for problems that need fast inter-thread synchronization, for instance, the FFT. Furthermore, the C64 instruction set architecture (ISA) features a large number of atomic in-memory instructions and exotic instructions, which greatly facilitate the computing of the FFT. For example, the Bit Gather instructions are intensively used for the bit-reversal permutation. All these architecture



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

Figure 3.2: Cyclops-64 Memory Hierarchy

features greatly facilitate the development of FFT-like applications.

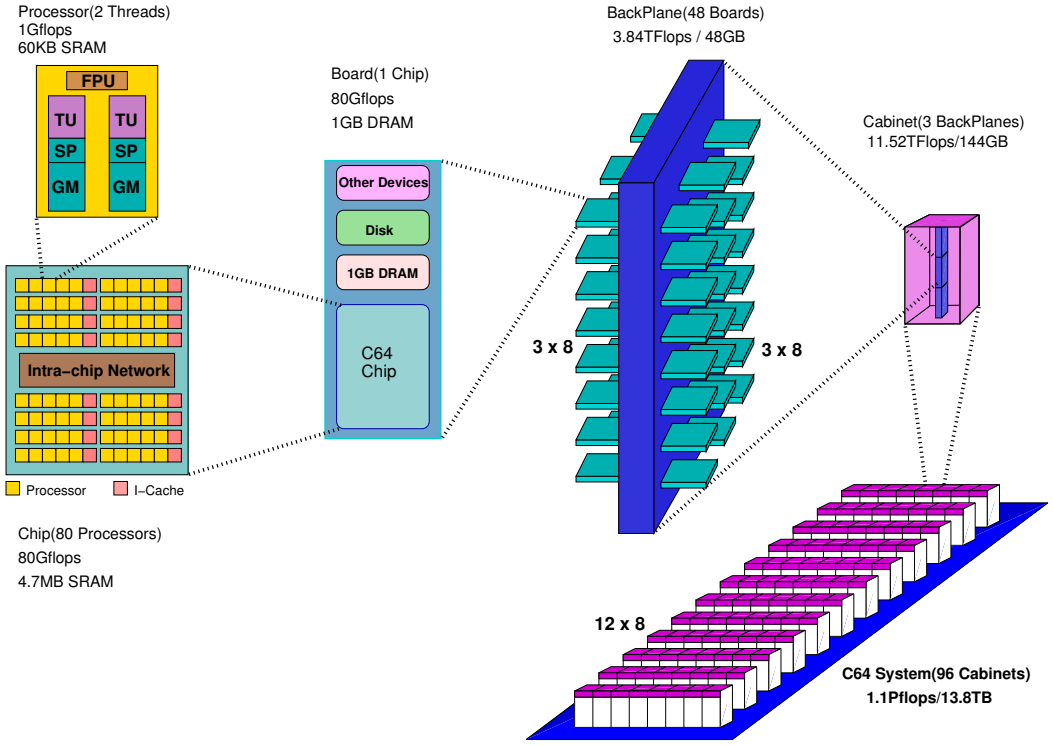
Finally, Figure 3.3 illustrates an instance of a C64 supercomputer architecture with $24 \times 24 \times 24$ logically arranged C64 nodes in the 3D-mesh configuration.

3.2 Cyclops-64 System Software Toolchain

Figure 3.4 shows the C64 system software toolchain [25], which is used for software and application development on the C64 system. The toolchain consists of following basic components:

- **Binary utilities (binutils):** assembler, linker, and objdump, etc. The binutils is ported from GNU binutils-2.11.2 [38].

- **GNU CC compilers:** C and Fortran compilers, which are ported from GCC-3.2.3/GCC-4.0.0/GCC-4.1.1 suite [37]. To fully exploit the explicitly addressable multi-layered memory hierarchy of C64, the compiler, assembler, and



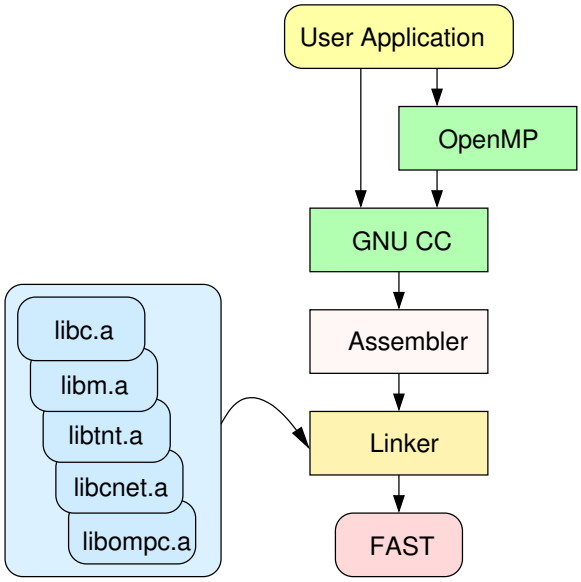
Courtesy: This figure was first created by Juan del Cuvillo and then successively revised by Weirong Zhu and the author of this thesis. Information in the figure is provided by Monty Denneau.

Figure 3.3: Cyclops-64 Supercomputer

linker are enhanced to support segmented memory spaces that are not contiguous. In other words, multiple sections of code and data can be allocated on different memory regions.

- **FAST simulator:** Before the actual C64 chip is available, the development and research of system software and scientific and engineering applications are conducted on an execution-driven, binary-compatible simulator of a multi-chip multithreaded C64 system, which is called Functionally Accurate Simulator Toolset (FAST) [24].

  FAST is a full-system simulator of a multi-chip multithreaded C64 system. It accurately reproduces the functional behavior of hardware components such as



Courtesy: Juan del Cuvillo.

Figure 3.4: Cyclops-64 System Software Toolchain

thread units, on-chip and off-chip memory banks, and the 3D-mesh network. Although it is not cycle accurate, it is useful for performance estimation [24]. Table 3.1 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) [26]. The TVM includes the TNT non-preemptive thread model, memory model, and synchronization model. In the TNT thread model, thread execution is non-preemptive and software threads map directly to hardware thread units. 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 [26]. The TNT library provides user and library developers an efficient Pthreads-like API for thread level parallel programming purposes.

Table 3.1: 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

- **Standard C and math libraries:** the libraries are derived from those in newlib-1.10.0 [61]. 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 [28] to provide user-level remote memory read/write functionality.

- **BNET communication library:** The BNET communication library is used to manage the B-switch communication hardware [29] to provide fast chip-to-chip communication.

- **SHMEM:** The SHMEM [65] shared memory access library, which is built on CNET, is developed to support high-level shared memory programming models 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 4

OPTIMIZATIONS AND DISCUSSION

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 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 3.2), 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.

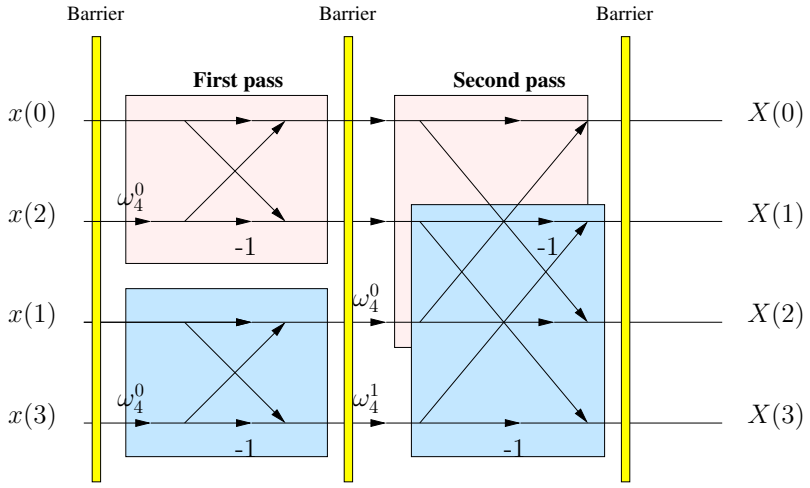


Figure 4.1: Example of 2-point Work Unit

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

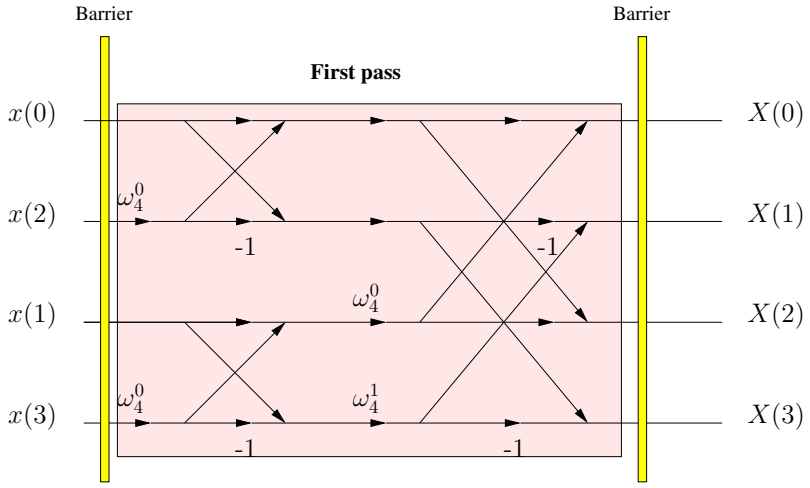


Figure 4.3: Example of 4-point Work Unit

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 2.6, 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

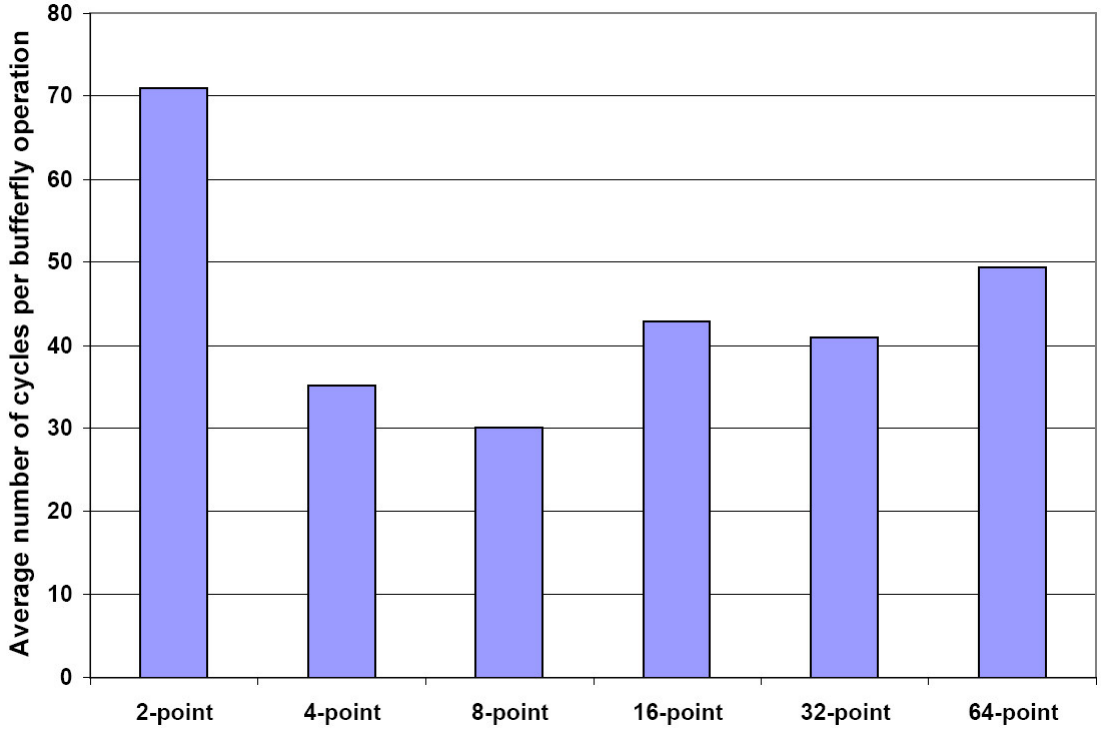


Figure 4.4: Number of Cycles per Butterfly Operation versus the Size of Work Unit

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 2.6 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 at 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.

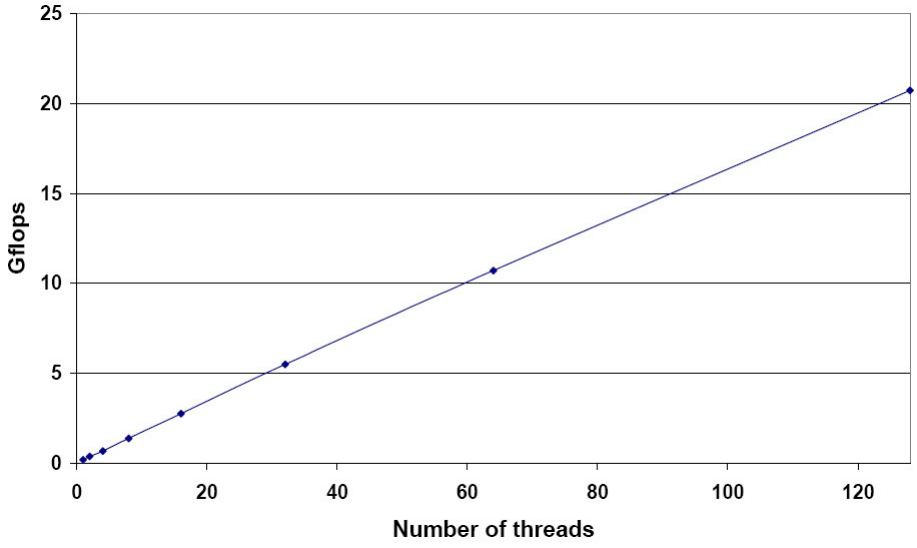


Figure 4.5: Performance of the Optimized 1D FFT Implementation

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 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.

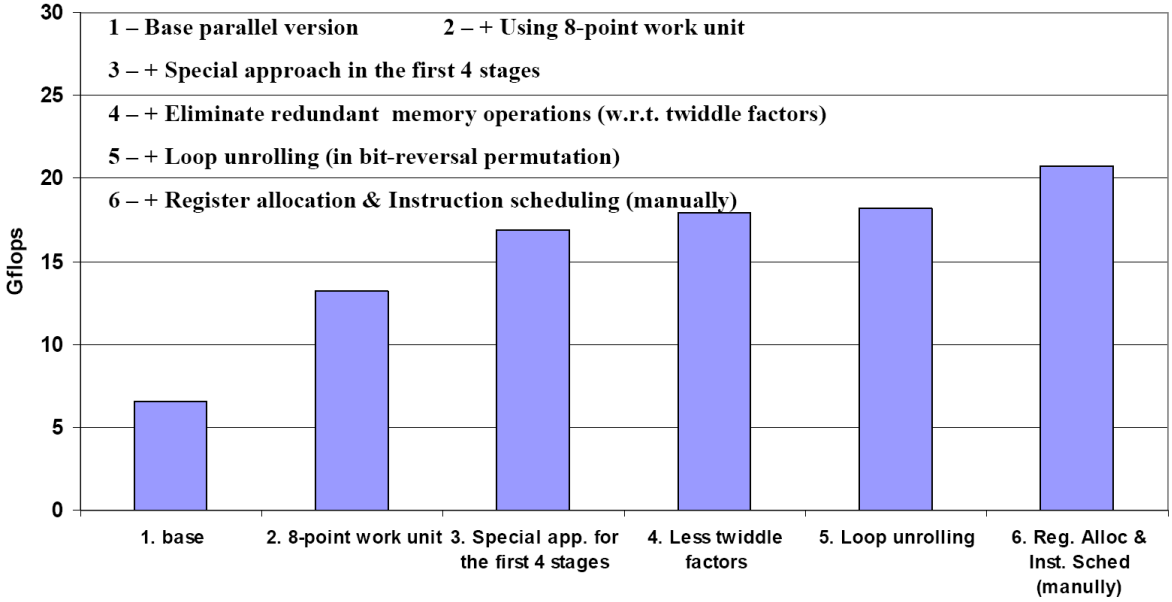


Figure 4.6: Effect of Optimization Techniques of 1D FFT Implementation (without the Memory Hierarchy Aware Compilation)

4.2 2D FFT

As mentioned in Chapter 2, 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.

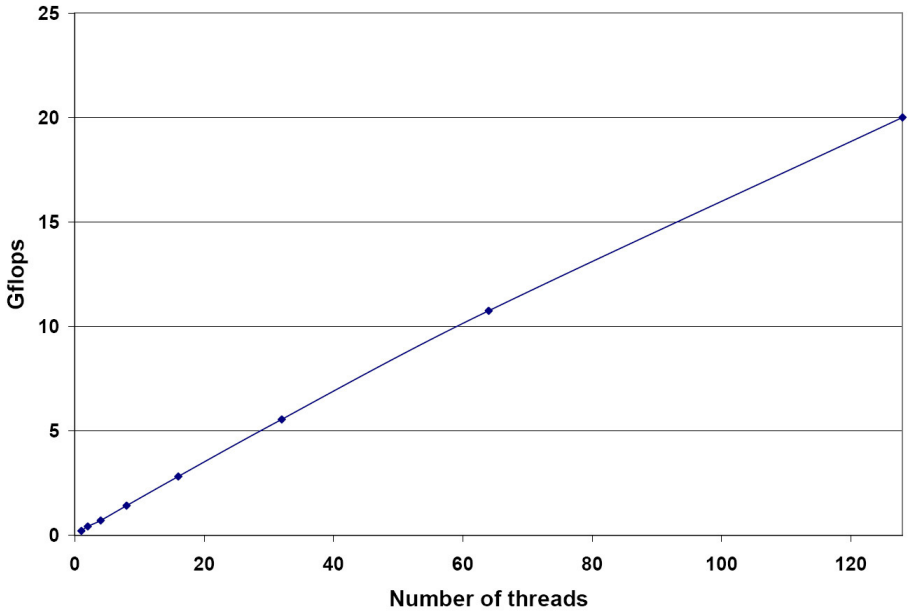


Figure 4.7: Performance of the Optimized 2D FFT Implementation

Chapter 5

CONCLUSIONS AND FUTURE WORK

In this thesis, 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 [30], 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 thesis, together with other software development experiences on C64 [45, 74, 78], clearly highlight the benefits and advantages of employing many-core architectures and serves as a basis for future research.

While the optimization techniques presented in this thesis 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 [30, 48]. 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. 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.

BIBLIOGRAPHY

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

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

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

[4] 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.

[5] AMD. AMD Multi-Core. http://multicore.amd.com/.

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

[7] 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.

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

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

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] Lin Chao. Preface: Meet the Intel Core Duo processor. *Intel Technology Journal*, 10(2):iii–iv, May 2006.

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

[19] ClearSpeed. ClearSpeed CSX600. http://www.clearspeed.com/products/si.php.

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

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

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

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

[24] 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 (MoBS2005), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005)*, Madison, Wisconsin, June 2005.

[25] 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.

[26] 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, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 265, Denver, Colorado, USA, April 2005.

[27] 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.

[28] 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.

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

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

[31] 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.

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

[33] 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".

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

[35] Guang R. Gao. Programming and compiling for TiNy threads (TNT) – experience with Cyclops-64 architecture, Dec 2006. Invited talk at High Performance Computing Workshop on Programming Languages, Sandia National Labs.

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

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

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

[39] Michael Gschwind. Chip multiprocessing and the Cell broadband engine. In *in Proc. of the 3rd Conf. on Computing frontiers*, 2006.

[40] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[41] Anshul Gupta and Vipin Kumar. On the scalability of FFT on parallel computers. In *FMPSC: Frontiers of Massively Parallel Scientific Computation*. National Aeronautics and Space Administration NASA, IEEE Computer Society Press, 1990.

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

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

[44] 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.

[45] 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.

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

[47] Intel. Cluster OpenMP for Intel Compilers for Linux. http://www.intel.com.

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

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

[50] 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.

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

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

[53] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[54] Peter M. Kogge. Past predictions, the present, and the future trends, Dec 2006. Invited talk at High Performance Computing Workshop on Programming Languages, Sandia National Labs.

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

[56] Barbara M. Chapman Lei Huang and Zhenying Liu. Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. *Parallel Computing*, 31(10-12):1114–1139, 2005.

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

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

[59] S. Min, A. Basumallik, and R. Eigenmann. Optimizing OpenMP programs on software distributed shared memory systems. *International Journal of Parallel Programming*, 31(3):225–249, 2003.

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

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

[62] Huy Nguyen and Lizy Kurian John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *International Conference on Supercomputing*, pages 11–20, 1999.

[63] 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.

[64] Inho Park and Seon Wook Kim. Study of OpenMP applications on the InfiniBand-based software distributed shared-memory system. *Parallel Computing*, 31(10-12):1099–1113, 2005.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] SPIRAL. SPIRAL website. http://www.spiral.net.

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

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

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

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

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

[76] 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.

[77] 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.

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

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

[80] 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.

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