EFFICIENT MAPPING OF FAST FOURIER TRANSFORM ON THE CYCLOPS-64 MULTITHREADED ARCHITECTURE

by

Liping Xue

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

Summer 2007

© 2007 Liping Xue
All Rights Reserved
EFFICIENT MAPPING OF FAST FOURIER
TRANSFORM ON THE CYCLOPS-64
MULTITHREADED ARCHITECTURE

by

Liping Xue

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 and Computer Engineering

Approved:
Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved:
Carolyn A. Thoroughgood, Ph.D.
Vice Provost for Research and Graduate Studies
I would like to thank Prof. Gao for giving me the opportunity to study in CAPSL. Without his help, I would not have been able to come here and start my research.

I thank Dr. Hu. He is the person that gave me the opportunity to join the Cyclops-64 team and take part in such an interesting project. He shared his system software development experience with me and helped me a great deal in writing this thesis.

I thank Long Chen, who answered me a lot of question on the base implementation of FFT on C64 and also gave me many invaluable suggestions when I was developing new optimizations for FFT on C64.

I thank Juan del Cuvillo and Fei Chen, who have installed the C64 software toolchain, and answered me a lot of questions about the toolchain.

I thank Weirong Zhu, Guangming Tan and Chen Chen. Weirong helped me a lot on the usage of C64 software toolchain. Guangming and Chen’s feedback on my search-based algorithm were very helpful.
Lastly, I thank everybody in CAPSL who helped me in the last year. They are very smart, warmhearted people. It has been a wonderful experience to work together with them.
DEDICATION

To my parents and husband.
TABLE OF CONTENTS

LIST OF FIGURES .................................................. viii
ABSTRACT .............................................................. x

Chapter

1 INTRODUCTION ...................................................... 1
  1.1 Background ....................................................... 1
  1.2 Problem Statement .............................................. 2
  1.3 Contributions ................................................... 3
  1.4 Glossary .......................................................... 4

2 FAST FOURIER TRANSFORMATION ............................... 5
  2.1 FFT Definition ................................................... 5
  2.2 Cooley-Tukey Algorithm ....................................... 6
  2.3 Recursive Implementation for Cooley-Tukey Algorithm ........ 6
  2.4 Iterative Implementation for Cooley-Tukey Algorithm ........ 8

3 CYCLOPS-64 ARCHITECTURE ................................... 10

4 OPTIMIZATIONS AND DISCUSSIONS ............................. 12
  4.1 1D FFT .......................................................... 12
    4.1.1 Previous Implementation ................................ 13
    4.1.2 The Effect of Number of Threads ....................... 16
    4.1.3 Search-based Scheme ..................................... 17
    4.1.4 Exploration of Larger Work Unit ....................... 22
      4.1.4.1 Motivating Example ................................ 22
      4.1.4.2 Kernel Function for 16-point Work Unit .......... 23
# LIST OF FIGURES

2.1 Butterfly operation in radix-2 Cooley-Tukey algorithm .......................... 8  
2.2 8-point radix-2 Cooley-Tukey example .............................................. 9  
3.1 C64 Chip/Architecture ........................................................................... 10  
4.1 Scalability for 1D FFT on C64 ............................................................... 17  
4.2 Search Plans for 1D FFT ........................................................................ 24  
4.3 Optimize Kernel for 16-point Working Unit .......................................... 26  
4.4 Optimize Kernel for 32-point Working Unit .......................................... 26  
4.5 Kernel Evaluation I ................................................................................. 28  
4.6 Kernel Evaluation II ................................................................................ 29  
4.7 Search Plans with Kernel Function for 16-point and 32-point Working Unit ................................................................. 29  
4.8 *Major – reversal* Distribution Scheme with 4 Threads ......................... 32  
4.9 Scalability for 2D FFT on C64 ................................................................. 34  
4.10 Search Plan for 2D FFT ......................................................................... 35  
5.1 Cyclops-64 software toolchain ................................................................. 39  
5.2 Performance Evaluation for 1D FFT on C64 .......................................... 41  
5.3 Performance Evaluation for 2D FFT on C64 .......................................... 41
5.4 FFTW3 Performance on G5 and Pentium IV ... 43
The emerging multi-core architectures unveil opportunities of massive on-chip parallelism through hardware support, but also present great challenges to application developers and system software designers. In this paper, we report our experience of optimizing the Fast Fourier Transform (FFT) on IBM Cyclops-64 (C64) architecture, a novel multi-core architecture consisting of 160 threads, explicit memory-hierarchy and interconnection network to provide massive on-chip parallelism. C64 does not have data cache and thus cannot take advantage of cache-oblivious algorithm. In addition, current implementation of cache-oblivious method is entirely based on cache memory hierarchy that does not lend itself to the construction of an accurate performance model - which is critical in the performance optimization of data movement through a Cyclops-64 style explicit memory hierarchy. Therefore, to make a cache-oblivious FFT working efficiently on C64 is probably non-trivial.

This thesis takes a different path. The main contribution of this thesis includes: 1) an iterative search approach has been proposed and implemented for the C64 architecture taking advantage of the explicit memory hierarchy. Our approach fully exploits the opportunity provided by explicitly-addressable on-chip memory hierarchy (without caches) and constructs an accurate/deterministic performance model analytically. This model is used to rapidly calculate the performance of different “FFT computation plans” iteratively. Such performance numbers will be productively used by our search based optimization procedure; 2) a new technique for optimizing the scratch-pad memory space utilization has been proposed that
can judiciously explore live-range splitting methods and a significant performance
gain has been achieved as evidenced by our experiments; 3) an implementation of
the proposed methods has been implemented on the C64 software and simulation
toolchain, and a detailed scalability study of FFT on C64 architecture has been
conducted. Furthermore, an in-depth analysis has been provided to illustrate the
choice of optimal (vs. maximum) number of threads under each situation based on
tradeoffs between the computation power and the synchronization overhead. The
experimental results have demonstrated up to 25.5% speedup over a best known
non-search based method.
Chapter 1

INTRODUCTION

In this chapter, we give the background introduction of this thesis. This chapter is organized as follows: In sec 1.1, a brief introduction about multi-core architecture especially the Cyclops-64 multi-core architecture is presented. Problem statements are given in sec 1.2. The main contributions of this thesis are presented in sec 1.3.

1.1 Background

While IC manufacturing technology continues to improve, some physical limitations can cause significant heat dissipation problem. From the application perspective, although some techniques like superscalar and pipelining are suitable to exploit instruction-level parallelism (ILP) in some applications, these techniques are not suitable for applications without enough ILP. But those applications may have many thread level parallelism (TLP) which is suitable for the multi-core architecture. The increased available space and physical limitations due to the improvement of manufacturing technology and the highly demand for increased TLP forces the creation of multi-core. Multi-core architecture presents big challenges to the application developers and system software designers on how to exploit the TLP provided by the multi-core chips.

While a lot of researchers believe that multi-core architectures will become the mainstream in the future, there are only a few studies about the application development on those advanced architectures have been reported.
In this thesis, we report our studies on the implementation and tuning of the Fast Fourier Transform (FFT) on IBM Cyclops-64(C64) multi-core architecture. The experiment platform in our study is the C64 chip, the computing engine for high performance computation. Each C64 chip is composed of 160 thread units, 80 floating point units, 160 32KB SRAM banks and a crossbar with a large number of ports (96 × 96). Other components include 4 off-chip DRAM controllers, A-switch which is used to form 3D mesh network to connect multiple C64 chips, GigaBit Ethernet controller and I/O devices. C64 chip provides massive on-chip parallelism, massive on-chip bandwidth, and multiple level memory hierarchy. C64 chip does not have data cache. The explicit memory hierarchy of C64 is a big challenge to the software development on C64. Application developers need to exploit the locality in the application.

1.2 Problem Statement

FFT is one of the most important DSP algorithms which is widely used in different areas and thus has been extensively studied on different platforms. Most DSP algorithms such as FFT are usually used in the real-time system. High performance computing engine is highly desired in this application domain. On the other hand, most DSP algorithms have some degree of parallelism [1, 2], and thus can take the advantage of multi-core architecture to achieve better performance. In this thesis, we are interested in the following questions:

• Does FFT running on C64 has different behavior from that running on a general purpose single core machine?

• Can FFT take the advantage of multi-core architecture to achieve better performance?

• How to implement, analyze and tune FFT on a multi-core architecture like C64?
1.3 Contributions

Long et.al. [3] studied different size of work units and found the optimal work unit according to the experiment result. Based on their observation, they proposed a fixed approach to implement FFT on C64 architecture.

Our study is based on the previous work of Long et.al[3]. The scheme proposed in [3] has several disadvantages:

- This fixed approach is less flexible than the search-based approach. Although the optimal work unit is used in the implementation, for some input size, best performance cannot guarantee.

- The performance of FFT on C64 is determined by two factors: computation overhead and synchronization overhead. The computation overhead lies on kernel functions used in the implementation, while, the synchronization overhead lies on the number of barriers involved in the computation process. Their scheme tried to reduce the computation overhead by choosing the optimal work unit, but they neglected the synchronization overhead which is determined by the number of barriers.

- The optimal work unit mentioned in [3] is not really “optimal” for C64 architecture.

The main contributions of this paper include:

- propose and implement a search-based approach to find the best implementation for FFT on C64 architecture. This search approach on C64 architecture with explicit memory hierarchy is different from previous studies such as FFTW[4] and SPIRAL[5], which target architectures with cache-based memory hierarchy.

- propose a new technique for efficiently scratch pad memory space utilization to optimize FFT on C64.
• conduct a scalability study of FFT on C64 architecture and an in-depth analysis to illustrate the choice of the optimal number of threads for each specific input size.

The rest of this thesis is organized as follows. In chapter 2, a brief introduction of FFT algorithm is given. In chapter 3, the Cyclops64 architecture and its major features are presented. In chapter 4, our study about optimizations and analysis of tuning both 1D and 2D FFT on C64 architecture, which includes a search scheme to find the best plan, the scalability study on C64, and using scratch pad memory to optimize FFT to achieve better performance are reported. In chapter 5, performance of both 1D and 2D FFT on C64 are evaluated. In chapter 6, related works are summarized. In chapter 7, conclusion is presented and future works are discussed.

1.4 Glossary

The important terms used in this thesis are put in the appendix A for the convenience of readers.
Chapter 2

FAST FOURIER TRANSFORMATION

In this chapter, a brief introduction about FFT has been presented. This section is organized as follows: a FFT definition is reviewed in sec 2.1, and then the most widely used Cooley-Tukey algorithm[6] for FFT computation is reviewed in sec 2.2. Finally, the recursive and iterative implementations for Cooley-Tukey algorithm are discussed in sec 2.3 and sec 2.4 respectively.

2.1 FFT Definition

Let $X$ be a sequence of $n$ complex numbers. The discrete Fourier transform (DFT) of $X$ is defined by the following equation:

$$Y[i] = \sum_{j=0}^{n-1} X[j] \omega_n^{-ij}$$

where $\omega_n = e^{2\pi \sqrt{-1}/n}$ is a primitive root of unity, and $0 \leq i \leq n$.

The backward DFT is defined by the following equation.

$$Y[i] = \sum_{j=0}^{n-1} X[j] \omega_n^{ij}$$

Directly evaluating these equations needs $O(n^2)$ arithmetical operations. The FFT algorithm is an efficient algorithm to compute the discrete Fourier transform and its inverse in $O(n \log_2 n)$ operations. FFT is widely used in many areas, including the digital signal processing, image processing and other domains.
2.2 Cooley-Tukey Algorithm

There are many variants of FFT algorithms. But the most common FFT algorithm is the Cooley-Tukey algorithm\[6\]. This algorithm recursively break down a DFT with size \( N = N1N2 \) into two smaller DFT with size \( N1 \) and \( N2 \). The most well-known use of Cooley-Tukey algorithm is to divide the transform into two pieces of the equal size at each step, which is called radix-2 Cooley-Tukey algorithm and therefore limited to power of two sizes.

Let us consider the \( N = 2^t \) point DFT, \( x(n) \), this radix-2 algorithm divides the \( N \)-point data sequence into two \( N/2 \)-point data sequences, \( f1(n) \) and \( f2(n) \) according to the even-indexed and odd-indexed points of \( x(n) \) respectively. Then the \( N \)-point DFT can be computed as the two \( N/2 \)-point DFTs:

\[
X(k) = F1(k) + \omega_N^k F2(k), 0 \leq k \leq \frac{N}{2} - 1
\]

\[
X(k + \frac{N}{2}) = F1(k) - \omega_N^k F2(k), 0 \leq k \leq \frac{N}{2} - 1
\]

where \( \omega_N^k \) are called twiddle factors, \( F1(k) \) and \( F2(k) \) are the DFTs of \( N/2 \)-point DFT of \( f1(n) \) and \( f2(n) \) respectively. \( F1(k) \) and \( F2(k) \) can be computed recursively to obtain the final solution of the original problem. The complexity of this algorithm is \( O(N\log_2 N) \). The above computation is usually referred to “butterfly” operation in the radix-2 Cooley-Tukey algorithm, which is shown in Figure 2.1. The multidimensional FFT problem can be solved by alternatively performing 1D FFT on each dimension.

Although the basic idea of Cooley-Tukey algorithm is recursive, it can be implemented either recursively or iteratively. The recursive and iterative implementations will be briefly described in the following sections.

2.3 Recursive Implementation for Cooley-Tukey Algorithm

As we have already described, for the given FFT with size \( N = N1N2 \), this algorithm recursively computes \( N2 \) FFT transforms of size \( N1 \), multiplies the results by the twiddle factors, and then computes \( N1 \) FFT transforms of size \( N2 \).
FFTW[4] implements this cache-oblivious Cooley-Tukey algorithm. Typically, a cache-oblivious algorithm works by a recursively divide and conquer fashion, where the problem is divided into smaller and smaller subproblems. Eventually, the subproblem size will fit into the cache regardless the cache size. For example, a cache-oblivious FFT algorithm recursively breaks down a DFT with size $N = N_1 N_2$ into two subproblems with smaller size $N_1$ and $N_2$ and computes these two subproblems recursively. Eventually, the subproblem size is smaller enough to fit into the cache and can be computed efficiently.

FFTW implements a library of highly optimized blocks of C code called codelets. A codelet might compute a DFT of a fixed size. There are two flavors of codelets. Normal codelets compute the DFT of fixed size and used as the base case of recursion. Twiddle codelets are similar as normal codelets, but in addition they need multiply the input by the twiddle factors. FFTW is composed of two components executor and planner. The executor is the component that actually computes the FFT transform by interpreting a plan, which is a data structure that specifies the factorization of $N$ and which codelets are used in computing this transform. The planner find the best plan by measuring the actual runtime of different plans. For example, the following is the high-level description of a possible plan for a FFT transform with size $N = 128$:

```
DIVIDE-AND-CONQUER(128,4)
DIVIDE-AND-CONQUER(32,8)
SOLVE(4)
```

For the above plan, the executor first computes 4 transforms of size 32 recursively and then it uses the twiddle codelet with size 4 to combine the results of subproblems. Similarly, for the problem with size 32, the executor first computes 8 transforms of size 4 by using a normal codelet, which is specified in the last line of the above plan and then combines the subproblems by using a twiddle codelet with size 8.
2.4 Iterative Implementation for Cooley-Tukey Algorithm

There are also many practical implementations that use an iterative algorithm to avoid the recursion overhead. Our implementation in this thesis is an iterative algorithm. The detailed implementation of this iterative algorithm will be discussed in chapter 4. For this iterative algorithm, the input data need to be reordered before the butterfly computation, which is called *bit-reversal permutation*. Figure 2.2 shows an example of 8-point FFT using an iterative Cooley-Tukey algorithm. Before the computation stages, the bit-reversal permutation is applied to the input 8-point data. This computation needs 3 stages computation, and each stage includes 4 butterfly operations.

![Butterfly operation in radix-2 Cooley-Tukey algorithm](image)

**Figure 2.1:** Butterfly operation in radix-2 Cooley-Tukey algorithm
Figure 2.2: 8-point radix-2 Cooley-Tukey example
Chapter 3

CYCLOPS-64 ARCHITECTURE

In this chapter, we briefly introduce the Cyclop-64 architecture (C64).

The C64 chip, shown in Figure 3.1, is designed to be the computation engine for running high performance applications. A C64 supercomputer system consists of thousands of C64 chip which is connected through a 3-D mesh network.

![Figure 3.1: C64 Chip/Architecture](image)

The C64 chips favors massive parallelism by integrating 80 64-bit processors, 160 embedded SRAM banks and interconnection networks in one silicon chip. Each
64-bit processor includes 2 thread units (TUs) and 1 floating point unit (FPU). Each thread unit is a single-issue, in-order 64-bit RISC processor running at 500Mhz clock rate. It has 64 64-bit registers and 32KB SRAM. Other on-chip components include 16 shared instruction caches, 4 off-chip DRAM controllers, A-Switch etc.

All the on-chip components are connected to an on-chip pipelined crossbar network with huge number of ports (96 × 96), which can provide 4GB/s bandwidth per port per direction, 384 GB/s per direction in total.

Besides the crossbar network, all the thread units within a chip are connected by a 16-bit signal bus, which can be used to implement barriers efficiently. C64 instruction set architecture (ISA) features a rich set of hardware supported in-memory atomic instructions which facilitate the thread-level parallelism with fast inter-thread synchronizations.

The C64 memory hierarchy consists of three level memories, the scratch-pad (SP) memory, on-chip global interleaved memory (GM), and off-chip DRAM. C64 does not have data cache. Instead the SRAM memory are partitioned into two parts: SP and GM. Each thread unit has its own SP, which is the fast temporary storage to exploit the locality by software (2 cycles for load, 1 cycle for store). The GM is shared by all thread units on the chip with uniform access latency.
Chapter 4

OPTIMIZATIONS AND DISCUSSIONS

In this chapter, we discuss our experiences of tuning FFT on the C64 architecture. This chapter is organized as follows: Optimizations for 1D FFT including the review of previous implementation, the scalability study, the search-based scheme, and exploration of larger work unit are reported in sec 4.1. Optimizations for 2D FFT are reported in 4.2.

In the following experiments, we assume the input data are double-precision complex numbers and can fit into the on-chip GM. The twiddle factors are pre-computed and stored in GM. All the experiments are conducted by using FAST simulator[7], which is a functionally-accurate simulator that models the memory hierarchy of C64 architecture, including the latencies and bandwidth for each memory segment.

4.1 1D FFT

In this section, we discuss our experience of optimizing 1D FFT on the C64 architecture. In sec 4.1.1, the FFT implementation mentioned in [3] is reviewed. The scalability analysis of 1D FFT on C64 architecture is reported in sec 4.1.2. A search-based approach to search the best implementation is discussed in sec 4.1.3. In sec 4.1.4, a technique of optimizing kernel functions for larger work unit is proposed and remarkable speedup can be obtained by using these optimized kernel functions for larger work unit.
4.1.1 Previous Implementation

In [3], the authors defined work unit as “an arbitrarily defined piece of the work that is the smallest unit of concurrency that the parallel program can exploit”. In the following part, we use the same definition of work unit. 8-point work unit implies 3 stages computation. After investigating work unit of 2-point, 4-point, 8-point and 16-point, [3] found that 8-point work unit was the optimal work unit for C64 architecture.

For each work unit, they implemented several small procedures named kernel functions which perform a sequence of butterfly operations on the work unit. In the following, these kernel functions are briefly introduced and detailed implementations of these kernel functions are shown in the appendix B.

- 2-point work unit

2-point work unit implies 1 stage computation. Four different kernel functions, named \( r2v1 \), \( r2v2 \), \( r2v4 \) and \( r2v8 \) respectively, have been implemented for the 2-point work unit.

The basic one, which is called \( r2v1 \), is the butterfly operation described in [3]. Kernel \( r2v1 \) is composed of the following steps:

1. read 2-point data and twiddle factors from GM
2. perform a butterfly operations upon them
3. store the 2-point result back to GM

Kernel function \( r2v1 \) consists of 6 load operations, 10 floating point operations and 4 store operations, not considering the integer operations for computing the indexes. It need 4 registers for input data, 2 registers for indexes computation and 2 registers for twiddle factors and the total number of registers needed would be 8.
Kernel function $r2v2$ works on 2 groups of 2-point data and can be thought as a vector version of $r2v1$. Similarly, kernel function $r2v4$ works on 4 groups of 2-point data and $r2v8$ works on 8 groups of 2-point data.

- **4-point work unit**

4-point work unit implies 2 stages computation. Two different kernel functions, named $r4v1$ and $r4v2$ respectively, have been implemented for the 4-point work unit.

The basic one, which is called $r4v1$, is composed of the following steps:

1. read 4-point data, named $a,b,c,d$, and twiddle factors from GM
2. perform a butterfly operation on data pair $(a,b)$ and $(c,d)$ respectively
3. perform a butterfly operation on data pair $(a,c)$ and $(b,d)$ respectively
4. store the 4-point result back to GM

Kernel function $r4v1$ consists of 16 load operations, 40 floating point operations and 8 store operations. It need 8 registers for input data, 4 registers for the indexes computation and another 8 registers for twiddle factors. The total number of registers needed would be 20.

Kernel function $r4v2$ works on 2 groups of 4-point data and can be thought as a vector version of $r4v1$.

- **8-point work unit**

8-point work unit implies 3 stages computation. Only one kernel function $r8v1$ has been implemented for the 8-point work unit.

Kernel function $r8v1$, is composed of the following steps:

1. read 8-point data, named $a,b,c,d,e,f,g,h$, and twiddle factors from GM
2. perform a butterfly operation on data pair \((a, b), (c, d), (e, f)\) and \((g, h)\) respectively.

3. perform a butterfly operation on data pair \((a, c), (b, d), (e, g)\) and \((f, h)\) respectively.

4. perform a butterfly operation on data pair \((a, e), (b, f), (c, g)\) and \((d, h)\) respectively.

5. store the 8-point result back to GM.

Kernel function \(r8v1\) consists of 40 load operations, 120 floating point operations and 16 store operations. It needs 16 registers for input data, 8 registers for indexes computation and another 24 registers for twiddle factors. The total number of registers needed would be 48, which is less than the available registers of C64.

- **16-point work unit**

16-point work unit implies 4 stages computation. A general kernel function \(r16v1\) needs 32 registers for input data, 16 registers for indexes computation and another 64 registers for twiddle factors. The total number of registers needed would be 112, which is exceed the available registers of C64 and thus will introduce serious register spilling.

Therefore, a general kernel function for 16-point work unit is not applicable for C64 due to the high register pressure. But a specialized kernel function for the 16-point work unit which can only be applied in the first 4 stages computation is applicable for C64.

This specialized kernel function, \(r16v1-first\), only has 8 distinct twiddle factors. [3] defined these 8 twiddle factors as *macros* in the program. By doing this, the total number of registers needed is reduced to 41, which can fit into the C64 register file.
1D FFT with n-point data includes \( \lg_2 n \) stages computation. [3] proposed a scheme for a given FFT computation with n-point data, which can be summarized as follows:

- For the first 4 stages, a specialized kernel function for the 16-point work unit is applied.
- For the remaining \((\lg_2 n - 4)\) stages, a kernel function for the 8-point work unit is repeated applied until the last \((\lg_2 n - 4 - \frac{\lg_2 n - 4}{3})\) stage(s) left.
- For the last 1 or 2 stage(s), a kernel function for the 2-point work unit or 4-point work unit is applied respectively.

In order to reduce the number of branch instructions, for the last 1 or 2 stage(s), [3] chose to use kernel function \(r2v8\) and \(r4v2\) respectively. During each stage of this FFT computation, the work unit was assigned to threads in a round-robin way to achieve load balance.

### 4.1.2 The Effect of Number of Threads

[3] showed that linear speedup can be obtained for \(2^{16}\) 1D FFT when the number of threads increases. Can we always get more speedup if more threads are used? In order to answer this question, we do the following experiment.

We run 1D FFT code mentioned in [3] with different input sizes and different number of threads. Figure 4.1 shows the experiment result. The horizontal axis in Figure 4.1 is the number of running threads, and the vertical axis is the speedup over the base case, which is running with 1 thread.

Figure 4.1 shows the scalability for n-point FFT. The results indicate that for the given n-point data, the best performance can be obtained by running with an optimal number of threads \(p\). When the n-point FFT is running with less than \(p\) threads, the more threads are used, the better performance can achieve. When
the n-point FFT is running with more than \( p \) threads, the performance starts to degrade. We also find out that, this optimal number of threads \( p \) is proportional to the size of input data. The larger the input data, the more threads are needed.

This result also shows a new challenge for tuning applications on multi-core architectures. Compared to the single core architecture, the tuning procedure need to determine what is the optimal number of threads.

![Figure 4.1: Scalability for 1D FFT on C64](image)

### 4.1.3 Search-based Scheme

Table 4.1 lists all the kernel functions used in the experiment. A *plan* is defined to be a sequence of kernel functions used to perform the FFT computation. For example, for a 4-point FFT, “r4v1” and “r2v1+r2v1” are two possible plans. A barrier is needed after each kernel function call to synchronize. In the above example for a 4-point FFT, if we use plan “r4v1”, we only need one barrier, otherwise, an extra barrier is needed between these two occurrences of kernel function *r2v1*. 
<table>
<thead>
<tr>
<th>KernelIndex</th>
<th>KernelName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r2v1</td>
<td>2-point work unit, 1 stage computation, working on 1 group data</td>
</tr>
<tr>
<td>2</td>
<td>r2v2</td>
<td>2-point work unit, 1 stage computation, working on 2 groups data</td>
</tr>
<tr>
<td>3</td>
<td>r2v4</td>
<td>2-point work unit, 1 stage computation, working on 4 groups data</td>
</tr>
<tr>
<td>4</td>
<td>r2v8</td>
<td>2-point work unit, 1 stage computation, working on 8 groups data</td>
</tr>
<tr>
<td>5</td>
<td>r4v1</td>
<td>4-point work unit, 2 stages computation, working on 1 group data</td>
</tr>
<tr>
<td>6</td>
<td>r4v2</td>
<td>4-point work unit, 2 stages computation, working on 2 group data</td>
</tr>
<tr>
<td>7</td>
<td>r8v1</td>
<td>8-point work unit, 3 stages computation, working on 1 group data</td>
</tr>
<tr>
<td>9</td>
<td>r16v1-first</td>
<td>16-point work unit, 4 stages computation, working on 1 group data</td>
</tr>
</tbody>
</table>

**Table 4.1: Kernel Function and Descriptions**

The disadvantage of the scheme proposed in [3] is that it “fixed” everything. In that scheme, 8-point work unit was used as the “optimal” work unit for C64. Kernel function \( r8v1 \) is repeated used until last 1 or 2 stage(s) left. For the last 1 or 2 stage(s), this scheme always uses kernel function \( r2v8 \) or \( r4v2 \) respectively to reduce the overhead of branch instructions.

Is this scheme really “optimal” for C64 architecture? Can we get a better plan by a search based scheme? We designed and implemented a search based scheme. All the experiments are running with the FAST simulator[7]. The search scheme can be described as follows:

Let \( F(i) \) denote the smallest cycles to finish the \( i \) stages computation. For a given \( n \)-point FFT with \( m \) stages computation \( (m = lg_2 n) \), our goal is to find a plan to complete these \( m \) stages computation within the smallest cycles \( F(m) \), which can be calculated according to the following equation.
\[ F(m) = \min\{F(m - S(k)) + T(k)\}, k \in Z \]

where \( Z \) is the set of all kernel functions in Table 4.1, \( T(k) \) denotes the total cycles of running kernel function \( k \) plus a barrier on the given n-point data, \( S(k) \) denotes the number of computation stages of kernel function \( k \). Dynamic programming has been used to calculate \( F(m) \). Assume plan \( p \) is the best plan for \( m \) stages computation, and the last kernel function used in plan \( p \) is \( k \). Since kernel function \( k \) can finish \( S(k) \) stages computation, the previous \( m - S(k) \) stages computation have already been finished before kernel function \( k \) is called. The smallest cycles to finish the previous \( m - S(k) \) stages computation is \( F(m - S(k)) \).

If plan \( p \) is the best plan for \( m \) stages computation, then the subplan \( p' \) of \( p \) should be the best plan to compute previous \( m - S(k) \) stages. In the following, this search-based approach as a two step algorithm is presented:

**Algorithm 1 Search\((m, Z)\)**

1: \( F(0) = 0 \)
2: for \( i = 1 \) to \( m \) do
3: \( \text{curr} = \text{MaxInt} \)
4: for each kernel \( k \) in \( Z \) do
5: \( \text{if } (i - S(k) \geq 0) \text{ then} \)
6: \( \text{if } (F(i - S(k)) + T(k) < \text{curr}) \text{ then} \)
7: \( \text{curr} = F(i - S(k)) + T(k) \)
8: \( \text{update BestPlan}[i] \)
9: \( \text{end if} \)
10: \( \text{end if} \)
11: \( \text{end for} \)
12: \( \text{end for} \)

**Step 1:** For the given n-point data, running each kernel function \( k \) in Table 4.1 plus a barrier with the optimal number of threads to get \( T(k) \). Compute \( S(k) \) for each kernel function in Table 4.1. For example, for the kernel function \( r2v1 \), it includes 1 stage computation, so we have \( S(r2v1) = 1 \).

**Step 2:** Run the search algorithm to search the best plan. A data structure
BestPlan is used to store the best plan. BestPlan[i] denotes the best plan for the first i stages computation. Variable curr stores the current smallest cycles for computing i stages computation. The initial value of curr is set to the maximum integer. The pseudocode of this search algorithm is shown in algorithm 1.

Table 4.2 shows the experiment result. The first column is the input size. An input with dimension n includes 2^n point data. The second column shows the optimal number of threads for this specific input size, which is determined in sec 4.1.2. In our search algorithm, we fix the number of threads for the specific input size. It means that, for the specific input size n, we first find the optimal number of running threads p, and then search different plans running with optimal p threads. The third column shows the plan based on the scheme described in [3], which is called the "base plan". The last column is the "best plan" that we get through the search algorithm. We run both the "base plan" and the "best plan" with the optimal number of threads p. Figure 4.2 shows the best plan’s speedup over the "base plan".

From Figure 4.2, we find out that, the performance of the "best plan" are always not worse than that of the "base plan" for the given n-point FFT. The "base plan" can achieve the same performance as that of the "best plan" when log_2 n – 4 can be equally divided by 3. For those log_2 n – 4 cannot be equally divided by 3, the "best plan" can achieve a better performance than the "base plan" for five input sizes. As we have already discussed, if log_2 n – 4 cannot be equally divided by 3, the "base plan" applied the kernel function r2v8 and r4v2 for the last 1 or 2 stage(s) computation respectively.

For the smaller input sizes (less than 2^9), "best plan" has a distinct speedup over the "base plan" for input size 2^5, 2^6 and 2^8, which means that the fixed plan is far from the optimal one and shows the importance of the search approach.

Comparing the "base plan" and the "best plan" shown in Table 4.2, one
observation is that the “best plan” uses kernel function $r2v4$ instead of $r2v8$. Intuitively, we consider that the kernel function $r2v4$ is better than $r2v8$. Therefore, we introduce a value, $avc_{pb}$ which means average cycles per butterfly, to evaluate the efficiency of kernel functions.

For the given kernel function, let $nc$ be the total cycles to execute this kernel function, $nb$ be the total bufferfly operations for this kernel function, the $avc_{pb}$ value for the kernel function is defined as $\frac{nc}{nb}$. The smaller the $avc_{pb}$, the higher the efficiency of the kernel function.

Figure 4.5 shows $avc_{pb}$ for different kernel functions. We find out that kernel function $r16$–first has the lowest $avc_{pb}$ value which implies the highest efficiency. Besides that, kernel $r8v1$ has the second lowest $avc_{pb}$ value. For the 2-point work unit, kernel $r2v8$ has a higher $avc_{pb}$ value than that of kernel $r2v4$, which implies that kernel $r2v4$ is better than $r2v8$.

Therefore, for the input n-point data FFT, if $\log_2 n - 4$ can be divided exactly by 3, the scheme proposed in [3] can achieve the best performance. However, if $\log_2 n - 4$ cannot be divided exactly by 3, the scheme proposed in [3] used a kernel function for the 2-point work unit or 4-point work unit for the last 1 or 2 stages respectively. In order to reduce the number of branch instructions, it used kernel function $r2v8$ and $r4v2$ in the implementation. Although the number of branch instructions has been reduced, it cannot guarantee the best performance. The reason is that the kernel function $r2v8$ has a larger $avc_{pb}$ value than that of $r2v4$, which implies using $r2v4$ can get better performance.

As shown in Figure 4.2, the ‘best plan” has a distinct speedup upto 25.5% over the “base plan” for the smaller input size, such as $2^5$, $2^6$ and $2^8$. But for the larger input size, there is no big difference between the “best plan” and the “base plan” for most cases. Only for two input sizes ($2^{11}$ and $2^{14}$), the “best plan” has around 2% speedup over the “base plan”.

21
The search scheme described above is different from previous works like FFTW\cite{4} and SPIRAL\cite{5} because in our scheme, the performance numbers of different “FFT plans” are calculated iteratively. FFTW\cite{4} generates different “FFT plans” and then run these plans on the target machine to select the “best plan”. The execution time of different “FFT plans” of FFTW cannot be calculated like we have done. The reason is that, on a cache-based architecture, the performance of “codelets” (like “kernel functions” in our scheme) is not additive and even not monotonic. The performance of a codelet is depend on how the previous codelet stores the data in the memory. For example, if the data are stored in the cache, the second codelet can run faster.

In summary, both FFTW\cite{4} and SPIRAL\cite{5} perform online search, while our search scheme is performed offline. The reason is that, both FFTW and SPIRAL are targeting architectures with cache-based memory hierarchy, whose performance is more unpredictable due to the cache behavior. While C64 is an architecture that does not have data cache, whose performance is more predictable. Therefore, offline search is applicable for C64 architecture. Offline search is important when a new processor is under development and only software simulator can be used to evaluate the performance. In this case, online search is impractical as the execution time would be forever. Offline search is also useful for heterogeneous architectures like IBM Cell Processor. The online search engine may not be able to run on SPU and making FFTW and/or SPIRAL working on such platform is non-trivial.

### 4.1.4 Exploration of Larger Work Unit

#### 4.1.4.1 Motivating Example

For the given n-point data FFT, the performance is determined by two factors: computation overhead and synchronization overhead. The computation overhead is determined by the efficiency of kernel functions used in the plan. The synchronization overhead is determined by the number of barriers in the plan.
<table>
<thead>
<tr>
<th>Dim</th>
<th>#Threads</th>
<th>Base Plan</th>
<th>Best Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>r16v1-first</td>
<td>r16v1-first</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>r16v1-first+r2v8</td>
<td>r4v1+r8v1</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>r16v1-first+r4v2</td>
<td>r8v1+r8v1</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>r16v1-first+r8v1</td>
<td>r16v1-first+r8v1</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>r16v1-first+r8v1+r2v8</td>
<td>r16v1-first+r4v2+r4v2</td>
</tr>
<tr>
<td>9</td>
<td>64</td>
<td>r16v1-first+r8v1+r4v2</td>
<td>r16v1-first+r8v1+r4v2</td>
</tr>
<tr>
<td>10</td>
<td>128</td>
<td>r16v1-first+r8v1+r8v1</td>
<td>r16v1-first+r8v1+r8v1</td>
</tr>
<tr>
<td>11</td>
<td>128</td>
<td>r16v1-first+r8v1+r8v1+r2v8</td>
<td>r16v1-first+r8v1+r8v1+r2v4</td>
</tr>
<tr>
<td>12</td>
<td>138</td>
<td>r16v1-first+r8v1+r8v1+r4v2</td>
<td>r16v1-first+r8v1+r8v1+r4v2</td>
</tr>
<tr>
<td>13</td>
<td>138</td>
<td>r16v1-first+r8v1+r8v1+r8v1</td>
<td>r16v1-first+r8v1+r8v1+r8v1</td>
</tr>
<tr>
<td>14</td>
<td>138</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r2v8</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r2v4</td>
</tr>
<tr>
<td>15</td>
<td>138</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r4v2</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r4v2</td>
</tr>
<tr>
<td>16</td>
<td>138</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r8v1</td>
<td>r16v1-first+r8v1+r8v1+r8v1+r8v1</td>
</tr>
</tbody>
</table>

**Table 4.2: Plan Comparison**

Let us take a FFT with \(2^{11}\) point data as an example. The “best plan” for \(2^{11}\) point FFT shown in Table 4.2 can be summarized as follows:

1. use \(r16\)-first (kernel function for 16-point work unit) to do the first 4 stages computation and execute a barrier
2. use \(r8v1\) (kernel for 8-point work unit) to do the next 3 stages computation and execute a barrier
3. use \(r8v1\) to do the next 3 stages computation and execute a barrier
4. use kernel \(r2v4\) to do the last 1 stage computation and execute a barrier

This plan needs 4 barriers. Can we reduce the number of barriers? The answer is yes. One barrier can be eliminated by using a kernel function for the 16-point work unit in the last 4 stages.

### 4.1.4.2 Kernel Function for 16-point Work Unit

In [3], a specialized kernel \(r16v1\)-first was used for the first 4 stages computation. [3] pointed out that a general kernel function for the 16-point work unit...
is not good for C64 because the number of registers needed exceeds the maximum available registers. The excessive register spill incurs performance degradation.

Actually, the performance of a general kernel function for the 16-point work unit can be improved by eliminating unnecessary indexes of twiddle factors and subsequent memory operations. As described in [3], in a 8-point work unit, only 1, 2 and 4 distinct twiddle factors are needed for the first, second and third stage respectively. Similarly, in a 16-point work unit, only 1, 2, 4 and 8 distinct twiddle factors are needed for the first, second, third and fourth stage respectively.

Besides that, the performance of a general kernel function for the 16-point work unit can be improved by using SPM in C64. SPM is the fast memory in C64 with very low latency (2 cycles for load and 1 cycle for store). Figure 4.3 illustrates our idea. The idea is that:

1. group the 16-point data into two groups of 8-point data; for each group, read the 8-point data from SRAM, do 3-stage computation on the 8-point data and store the intermediate result back to the SPM
2. regroup the 16-point data into eight groups of 2-point data; for each group, load the 2-point data from SPM, do 1-stage computation and store the result back to SRAM.

We call this optimized general kernel function for the 16-point work unit \textit{r16v1}. By using this \textit{r16v1}, we get a new plan for $2^{11}$ point FFT. This plan can be described as follows:

1. use \textit{r16-first} to do the first 4 stages computation and execute a barrier
2. use \textit{r16v1} to do the next 4 stages computation and execute a barrier
3. use \textit{r8v1} to do the next 3 stages computation and execute a barrier.

One barrier can be eliminated by using this new plan and thus around 7% speedup is achieved compared with the “best plan” in Table 4.2 when running with 128 threads. Although the \textit{avc-pb} value for \textit{r16v1} which is shown in 4.6 is a little bit larger than that of \textit{r8v1} and \textit{r2v4}, a better performance can be achieved by using \textit{r16v1} since it can reduce the number of barriers needed.

The overhead of barriers is proportional to the number of running threads. When the input data size increases, the number of running threads need to be increased to match the input size to achieve good performance. Since the barrier overhead is proportional to the number of running threads, for the large input data size, using the kernel function for 16-point work unit may improve the performance.

4.1.4.3 Kernel Function for 32-point Work Unit

Inspired by the speedup achieved by using the 16-point work unit, we decide to study a larger work unit, the 32-point work unit to expect a better performance.

For the 32-point work unit, only 1,2,4,8 and 16 distinct twiddle factors are needed for the first, second, third, fourth and fifth stage respectively, unnecessary indexes to the twiddle factors and memory operations can be eliminated.
Figure 4.3: Optimize Kernel for 16-point Working Unit

Figure 4.4: Optimize Kernel for 32-point Working Unit
Besides that, we can also use SPM to store the intermediate results to improve the performance. The idea is illustrated in Figure 4.4, which can be described as follows:

1. group the 32-point data into four groups of 8-point data; for each group, read the 8-point data from SRAM, do 3-stage computation on the 8-point data and store the intermediate result back to the SPM

2. regroup the 32-point data into eight groups of 4-point data; for each group, load the 4-point data from SPM, do 2-stage computation and store the result back to SRAM.

We call this kernel function for the 32-point work unit \textit{r32v1}. The result surprised us is that, as shown in Figure 4.6, the \textit{avc_pb} value for \textit{r32v1} is smaller than that of \textit{r16v1}, which means that \textit{r32v1} has higher efficiency than that of \textit{r16v1}.

By storing the intermediate result into SPM, a fast memory, a long live-range is split into several smaller live-ranges and thus interferences between live ranges are reduced. This optimization can reduce the register pressure and improve the performance. The overhead of load/store instructions introduced by live-range splitting is small since those memory operations are performed on SPM, a fast storage with low latency.

We run the search algorithm described in 4.1.3 with new added kernel functions \textit{r16v1} and \textit{r32v1}. Table 4.3 shows the new search result. The first column is the input size. An input with dimension $n$ includes $2^n$ point data. The second column is the optimal number of threads for this specific input size, which is determined in sec 4.1.2. The third column named “Plan I” is the best plan we get through the search algorithm without using the new added kernel functions. The last column named “Plan II” is the best plan we get through the search algorithm with the new added kernel functions.
Figure 4.7 shows the speedup of both “Plan I” and “Plan II” over the “base plan”. The result shows that, for smaller input size (less than $2^9$), there is no big difference between “Plan I” and “Plan II”. For larger size, “Plan II” has a better performance than that of “Plan I” by using new kernel functions $r_{16v1}$ and $r_{32v1}$. For input size equal or larger than $2^{11}$, “best plan” has a distinct speedup over the “base plan”, especially for the input size $2^{11}$ and $2^{15}$, which have around 12% and 8% speedup respectively. 21.5 GFLOPS is achieved by using “Plan II” for the input size $2^{15}$.

![Kernel Evaluation I](image)

**Figure 4.5:** Kernel Evaluation I

### 4.2 2D FFT

In this section, we discuss our experience of optimizing 2D FFT on C64 architecture. This section is organized as follows: previous implementation of 2D FFT is reviewed at sec 4.2.1, the scalability study of 2D FFT is reported at sec 4.2.2. Search-based scheme is also applied to 2D FFT and its result is reported at sec 4.2.3. The result of exploring larger work unit for 2D FFT is presented at sec 4.2.4.
Figure 4.6: Kernel Evaluation II

Figure 4.7: Search Plans with Kernel Function for 16-point and 32-point Work Unit
Table 4.3: Plan Comparison with Kernel of 16-point and 32-point Work Unit for 1D FFT

As described in section 2, a multidimensional FFT can be solved by alternatively performing 1D FFT on each dimension. For a $N \times N$ 2D FFT, we first perform a sequence of 1D FFT along each row by using any 1D FFT algorithm. After all transformations along the row dimension are done, we perform a sequence of 1D FFT along the column dimension. Our implementation of this 2D FFT on C64 follows this row-column algorithm described above.

4.2.1 Previous Implementation

The base parallel implementation mentioned in [3] assigned the whole row/column as a work unit to one thread. Work units were distributed to threads in the round-robin way.

This simple scheme has the load balance problem. For example, given a $32 \times 32$ 2D FFT, using more than 32 threads will not produce any performance improvement than using 32 threads. According to the simple scheme, each row/column as a work unit is assigned to one thread. For a $32 \times 32$ 2D FFT, 32 row/column is assigned to 32 threads. If more than 32 threads are used, the first 32 threads will work on their own work unit, other threads will remain idle because there are no work units assigned to them.
In another words, this simple scheme does not exploit enough concurrency and thus cannot fully utilize all the threads provided by C64. [3] resolved this issue by using fine-grain work unit. They divided each row/column of FFT into small tasks insteading of assigning the whole row/column to one thread. By using this fine-grain work unit, multiple threads may work on one single thread.

In addition, they also thought about possible data reuse when distributing the work unit to all threads. Common round-robin assignment scheme can distribute work units as evenly as possible to all threads. But this common scheme does not exploit the possible data reuse of 2D FFT computation.

In 2D FFT, the exact same set of operations are repeated performed on each row/column. For example, if a butterfly operation is performed on \(x(0, a)\) and \(x(0, b)\) with twiddle factor \(\omega\), then for \(0 \leq i \leq N\), a butterfly operation need to be performed on each \(x(i, a)\) and \(x(i, b)\) with the same twiddle factor \(\omega\). In [3], a major–reversal work distribution scheme is used to exploit these data reuse opportunities. When a thread completes a work unit consists of \(x(a, i_0), x(a, i_1), ..., x(a, i_m)\) in a row FFT \(x(a, :)\), this major–reversal scheme, instead of moving to the work unit in the same row, will reuse the computed index and twiddle factors by moving to the work unit to the next row FFT \((x(a+1, :)\) and perform the same set of butterfly operations on a work unit consists of \(x(a+1, i_0), x(a+1, i_1), ..., x(a+1, i_m)\). This procedure is repeated applied until this threads finishes all its workload or it reaches the last row FFT \(x(N-1, :)\). Similar procedures are applied to the column FFT and bit-reversal permutation.

Figure 4.8 illustrates this major–reversal work distribution scheme. Assume kernel function \(r8v1\) was used to compute \(16 \times 16\) 2D FFT and 4 threads are used in this computation. Each work unit includes 8 points because we use kernel function \(r8v1\) here. The total workload includes 256 points \((16 \times 16)\), and on average each thread is assigned to 64 points \((\frac{256}{4})\). In another words, each thread is assigned to 8
work units. For example, the following 8 work units are assigned to thread 1:

work unit 1: \{x(0,0),x(0,1),x(0,2),x(0,3),x(0,4),x(0,5),x(0,6),x(0,7)\}
work unit 2: \{x(1,0),x(1,1),x(1,2),x(1,3),x(1,4),x(1,5),x(1,6),x(1,7)\}
work unit 3: \{x(2,0),x(2,1),x(2,2),x(2,3),x(2,4),x(2,5),x(2,6),x(2,7)\}
work unit 4: \{x(3,0),x(3,1),x(3,2),x(3,3),x(3,4),x(3,5),x(3,6),x(3,7)\}
work unit 5: \{x(4,0),x(4,1),x(4,2),x(4,3),x(4,4),x(4,5),x(4,6),x(4,7)\}
work unit 6: \{x(5,0),x(5,1),x(5,2),x(5,3),x(5,4),x(5,5),x(5,6),x(5,7)\}
work unit 7: \{x(6,0),x(6,1),x(6,2),x(6,3),x(6,4),x(6,5),x(6,6),x(6,7)\}
work unit 8: \{x(7,0),x(7,1),x(7,2),x(7,3),x(7,4),x(7,5),x(7,6),x(7,7)\}

Kernel function \textit{r8v1} is applied to each work unit. The work of indexes computation and loading twiddle factors are only needed when thread 1 works on work unit 1. These indexes and twiddle factors can be reused when thread 1 works on other 7 work units.

For 2D FFT, a sequence of 1D FFT along the row dimension is firstly performed by using kernel functions for row FFT and then a sequence of 1D FFT along the column dimension is performed by using kernel functions for column FFT. Two
implementations of each kernel function are needed for the row FFT and column FFT respectively. For each kernel function, the implementation for row FFT and the one for column FFT are almost identical except for indexes computation.

For 2D FFT, [3] still considered 8-point work unit as the optimal work unit. A sequence of row/column 1D FFT are used to implement 2D FFT. The scheme for those row/column 1D FFTs is similar to the one described in sec 4.1.1. The same plan are used for both row and column 1D FFT. For each row/column 1D FFT, their scheme can be summarized as follows:

1. For the first 4 stages computation, a specialized kernel function for 16-point work unit is applied
2. For the remaining \((\lg_2 n - 4)\) stages computation, a kernel function for 8-point work unit is repeated applied until the last \((\lg_2 n - 4 - \frac{\lg n - 1}{3})\) stage(s) left.
3. For the last 1 or 2 stage(s) computation, kernel function for 2-point work unit is applied.

4.2.2 The Effect of Number of Threads

[3] also showed that linear speedup can be obtained for \(2^8 \times 2^8\) 2D FFT when the number of threads increases. As we have done for 1D case, we run 2D FFT code implemented in [3] with different input size and different number of threads.

Figure 4.9 shows the scalability for 2D FFT. An 2D FFT with dimension \(n\) includes \(2^n \times 2^n\) point data. For 2D FFT, there also exists the optimal number of threads \(p\) for each specific input size.

In addition, Figure 4.9 also shows that for an \(N \times N\) 2D FFT, the optimal number of running threads \(p\) is greater than \(N\). This result shows that all threads of C64 can be fully utilized by using fine-grain work unit.
4.2.3 Search-based Scheme

Since 2D FFT can be implemented as a combination of sequences of row/column 1D FFT, the search algorithm described in 4.1.3 can be used to search the best plan for 2D FFT. We use the same plan for both row and column 1D FFT. Kernel functions used in the search scheme is the same as those for 1D case, which is shown in Table 4.1.

Table 4.4 shows the experiment result. The first column is the input size. An 2D FFT with dimension $n$ includes $2^n \times 2^n$ point data. The second column is the optimal number of threads for this specific input size, which is determined in sec 4.2.2. The third column is the “base plan” which is described in [3]. The last column is the “best plan” generated by the search scheme.

As shown in Figure 4.10, “best plan” has around 6.3% and 7.9% speedup over the “base plan” for 2D FFT with $2^5 \times 2^5$ and $2^8 \times 2^8$ point respectively. For these two cases, the difference between “base plan” and “best plan” is that “best plan” uses kernel function $r2v4$ instead of $r2v8$ for the last 1 stage. For the $2^6 \times 2^6$
<table>
<thead>
<tr>
<th>Dim</th>
<th>#Threads</th>
<th>BasePlan</th>
<th>BestPlan</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16</td>
<td>r16v1-first</td>
<td>r16v1-first</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>r16v1-first+r2v8</td>
<td>r16v1-first+r2v4</td>
</tr>
<tr>
<td>6</td>
<td>128</td>
<td>r16v1-first+r2v8+r2v8</td>
<td>r16v1-first+r4v2</td>
</tr>
<tr>
<td>7</td>
<td>138</td>
<td>r16v1-first+r8v1</td>
<td>r16v1-first+r8v1</td>
</tr>
<tr>
<td>8</td>
<td>138</td>
<td>r16v1-first+r8v1+r2v8</td>
<td>r16v1-first+r8v1+r2v4</td>
</tr>
</tbody>
</table>

Table 4.4: Plan Comparison for 2D FFT

2D FFT, “best plan” has around 20.62% speedup over the “base plan”.

![Graph](image)

Figure 4.10: Search Plan for 2D FFT

4.2.4 Exploration of Larger Work Unit

In section 4.1.4, we find out that larger work unit can improve the performance of 1D FFT by reducing the number of barriers. Is it possible to have good performance by using larger work unit for 2D FFT?

For 1D FFT, we have implemented two optimized kernel functions for the 16-point work unit and 32-point work unit, which are named \( r16-v1 \) and \( r32-v1 \)
respectively by using the SPM. Similarly, for 2D FFT, kernel functions \textit{r16-v1} and \textit{r32-v1} are also implemented for both row FFT and column FFT.

We run the search algorithm by using new added kernel functions \textit{r16-v1} and \textit{r32-v1}. No plan with better performance is founded. Why we cannot get better performance by using larger work unit for 2D FFT?

Let us take $2^8 \times 2^8$ 2D FFT as an example to find out the possible reason. As we have described, for 2D FFT, we first do a sequence of 1D FFTs along the row dimension by using the kernel functions for row FFT and then do a sequence of 1D FFTs along the column dimension by using the kernel functions for column FFT. For $2^8 \times 2^8$ point 2D FFT, the “best plan” shown in Table 4.4 is “\textit{r16v1-first+r8v1+r2v4}”, which needs 3 barriers. 22.72GFLOPS is achieved by using the “best plan”. There are only 2 barriers needed if we use another plan “\textit{r16v1-first+r16v1}” and 21.91 GFLOPS is achieved by using this new plan.

The new plan reduces the synchronization overhead by removing one barrier. Why this new plan does not have the better performance? The reason is that the new plan has less reuse of indexes computation and twiddle factors than the “best plan”. Assume we run these $2^8 \times 2^8$ 2D FFT with 128 threads. On average, each thread is assigned to the work load with $2^9 \left(\frac{2^8 \times 2^8}{128}\right)$ points data. If we use kernel function \textit{r16v1}, each thread will do $2^5 \left(\frac{2^9}{2^7}\right)$ groups identical computations with the same indexes and twiddle factors. For each thread, these indexes only need to be computed once and twiddle factors are only need to be loaded once. While, if we use kernel function \textit{r8v1}, each thread will do $2^6 \left(\frac{2^9}{2^7}\right)$ groups identical computations with the same indexes and twiddle factors. For each thread, there are more reuse of indexes computation and twiddle factors by using kernel function \textit{r8v1} than using kernel function \textit{r16v1}. Besides that, kernel function \textit{r16v1} need 15 twiddle factors while kernel function \textit{r8v1} only need 7 twiddle factors. Therefore, for 2D FFT, using kernel function \textit{r16v1} may need more indexes computations and memory operations.
to load twiddle factors than using kernel function $r8v1$. 
Chapter 5

PERFORMANCE EVALUATION

In this chapter, we report the performance evaluation of both 1D and 2D FFT on C64 architecture. This chapter is organized as follows: A brief introduction of the simulation platform that we used to conduct the experiment is given in sec 5.1. A summary of performance evaluation of 1D and 2D FFT on C64 has been reported in sec 5.2. Detailed performance evaluation of both 1D and 2D FFT has been presented in sec 5.3 and 5.4 respectively. Finally, in sec 5.5, we compare the performance of FFT on C64 architecture with those on traditional cache-based single core architecture.

5.1 Simulation Platform

The experiment is conducted on the C64 FAST simulator[7]. FAST is an execution-driven, binary-compatible simulator for multithreaded C64 architecture. Although FAST is not cycle-accurate, it can accurately reproduce the functional behavior and count of hardware components including thread units, SRAM and DRAM banks, and the 3D-mesh network.

Figure 5.1 shows the Cyclops-64 software toolchain. The Cyclops-64 software toolchain includes the GNU C compiler, assembler, linker and the FAST simulator. User applications are compiled by GCC compiler, and then linked to the executable by the linker. Finally, the executable can be simulated by the FAST simulator.
5.2 A Summary of Primary Result

In this research, we evaluate the performance of both 1D and 2D FFT on C64 architecture with various input sizes. Our experiment results show that:

**Observation 1 (5.3)** We have observed that 1D FFT has a good scalability on C64 architecture. The speed (in GFLOPS) increases dramatically when the number of input size increases. The peak performance for larger input size like 65536-point FFT is very impressive on C64 architecture.

**Observation 2 (5.4)** We have observed that 2D FFT also has a good scalability on C64 architecture. The speed (in GFLOPS) increases dramatically when the number of input size increases. The peak performance for larger input size like $256 \times 256$ FFT is very impressive on C64 architecture.

**Observation 3 (5.5)** We have observed that the scalability of FFT running on a multi-core machine is different from that on a single-core machine. FFT running on a single-core machine can achieve high speed only for the smaller input size.
While, FFT running on a multi-core machine can obtain high speed for the larger input size.

5.3 1D FFT Performance

In this experiment, we run 1D FFT by using the “best plan” with the optimal number of threads. Figure 5.2 shows the experiment result. The horizontal axis is the input size, and the vertical axis is the speed in GFLOPS.

For smaller input size, the speed is not impressive. For example, for the smallest 16-point FFT, the speed shown on Figure 5.2 is only 152 MFLOPS. The reason is twofold. The first reason is that the in-order RISC processor of the thread unit on C64 operates at a moderate clock rate (500MHz). Another reason is that, for smaller input size, workload cannot distribute to all threads provided by C64. In another words, for smaller input size, most of the thread units are not utilized at all. For larger input size, the speed is very impressive. For example, 21.22 GFLOPS is achieved for the largest 65536-point FFT on C64 architecture. The highest GFLOPS shown in Figure 5.2 is 21.52 GFLOPS which is corresponding to computation of 32768-point FFT.

5.4 2D FFT Performance

We run 2D FFT by using the “best plan” with the optimal number of threads. Figure 5.3 shows the experiment result. For smaller input size, the speed is not impressive. For example, for the smallest $16 \times 16$ FFT, the speed shown in Figure 5.3 is only 1.51 GFLOPS. But for larger input size, the speed is very impressive. For the largest $256 \times 256$ FFT, 22.72 GFLOPS is achieved on C64 architecture.

5.5 Performance Comparison

Figure 5.4 is excerpted from [8]. It shows FFTW’s speed for 1D FFT on two machines: a 2GHz PowerPC970 (G5) and a 2.8GHz Pentium IV. For each machine,
Figure 5.2: Performance Evaluation for 1D FFT on C64

Figure 5.3: Performance Evaluation for 2D FFT on C64
they reported both the speed of FFTW tuned to that machine and the speed tuned to the other machine.

From Figure 5.4, we find out that for the G5 machine, the peak performance is around 3800 MFLOPS which is corresponding to the FFT with 64 points. After 64 points, the speed drops with the increase of input size. For the larger input size like 65536 points, the performance is only around 1100 MFLOPS.

Figure 5.4 also shows that for the Pentium IV machine, the peak performance is around 2100 MFLOPS which is corresponding to the FFT with 128 points. After 128 points, the speed drops with the increase of input size. For the larger input size like 65536 points, the performance is only around 700 MFLOPS.

One observation from Figure 5.4 is that the performance of FFT on these single core machines does not have good scalability. On those machines, although the performance for smaller input size is good, the performance for larger size is very poor.

The FFT performance behavior shown on those single core machines is different from that on C64. Figure 5.2 shows that, the peak performance of 1D FFT on C64 is around 21.52 GFLOPS which is corresponding to computation of 32768-point FFT.

The performance of 1D 65536-point FFT on C64 is around 20 and 30 times faster than that on G5 and Pentium IV machine respectively. In addition, the clock rate of C64 is only 500MHz, which is much slower than that of G5(2GHz) and Pentium IV(2.8GHz).

Therefore, we can make the conclusion that FFT can take the advantage of multi-core architecture like C64. FFT running on C64 has a good scalability than that running on the single-core machine.
Figure 5.4: FFTW3 Performance on G5 and Pentium IV
Chapter 6

RELATED WORK

In this chapter, we review related work. This chapter is organized as follows: in sec 6.1, FFT implementations on different architectures are reviewed. Research on automatic FFT code generators are summarized in sec 6.2. FFT implementations for multi-thread/multi-core architectures are presented in sec 6.3.

6.1 Various FFT Implementations

FFT has been extensively studied and implemented in various machines. Much research has been done to address the distributed FFT implementations on the hypercube architecture [9, 10]. Other parallel FFT implementations on arrays[11] and mesh architectures[12] have also been investigated. There has also been some research conducted in the shared memory FFT implementations[13]. The importance to consider the memory hierarchy to implement FFT effectively has been discussed in [14]. [15] also shows how to implement FFT efficiently by using local memory on CRAY-2. Two dataflow-based multithreaded FFT algorithms[16] are implemented in EARTH[17], a fine-grained data flow architecture. A FFT implementation on a GPU has been presented in [18].

6.2 Automatic FFT Code Generators

There have also severak efforts to automatically tune FFT on different architectures. FFTW planner[4] generates various plans on target machines and measures
the actual run time of many different plans to select the fastest one. FFTW uses dynamic programming to search the best plan from many different plans. SPIRAL[5] uses the special language SPL to represent the FFT problem as formulas. SPIRAL includes both algorithm level and implementation level optimizations. At algorithm level, SPIRAL applies rules on the formulas to generate the optimized formulas. At implementation level, SPIRAL translates the optimized formulas into C code which is further compiled using a standard C compiler and then measures the actual runtime. UHFFT[19] extends FFTW by offering two strategies to find the fastest plan. Besides that, during the decomposition and factorization, UHFFT also applies prime factor algorithm in addition to the mixed-radix Cooley-Tukey algorithm which is adopted by FFTW.

6.3 FFT Implementations for Multi-thread/Multi-core Architectures

There are also some FFT implementations on the multi-thread/multi-core architectures that have been presented. FFTW 3.1[8] implemented a multithreading DFT implementation to support the parallel FFT computation. Recently, [20] presents a FFT implementation for shared memory, especially for the SMP and multicore by extending their previous work SPIRAL. [21] extended the UHFFT to adapt to parallel architectures like SMP/CMP systems by searching the parallel plans. It develops some heuristics to reduce the time of searching plans on parallel architectures.
Chapter 7

CONCLUSION AND FUTURE WORK

In this chapter, we summarize our work and also present future works.

In this thesis, we have studied how to effectively implement and optimize FFT on C64 architecture. Our work includes:

• present an iterative offline search algorithm to search the plan for FFT computation with better performance on C64 multi-core architecture

• propose a technique to optimize large kernel functions by using SPM and significant performance gain has been achieved by using these large kernel functions

• implement the proposed method and study the scalability of both 1D and 2D FFT on C64 architecture

The study of the search-based scheme demonstrates that, for the FFT on C64 architecture, the plan proposed in [3] may not always achieve the best performance and the search-based scheme can get a better plan than the one based on the fixed scheme. In addition, although the offline search is not applicable for the traditional cache-based architecture, it is applicable for the C64 architecture.

The study of exploring kernel functions for larger work unit shows that performance of FFT on C64 is depend on both computation overhead and synchronization overhead. Tuning FFT on C64 need to take both factors into account. The study
of the optimization for large kernel function is a good example of how the SPM can be used effectively to optimize applications on C64.

The results of FFT scalability study shows that the optimal number of running threads is very important in tuning FFT on a multi-core architecture like C64.

In summary, our study shows that application development for multi-core architecture like C64 is not easy. We need to take both architecture features and application/algorithm properties into account when developing applications for multi-core architecture. This also poses more challenges for multi-core system software, especially for compilers and code generators.

There are several researches can be done in the future. As we have discussed, the memory hierarchy of C64 is different from the traditional cache-based architecture. C64 has a segmented memory space. SPM is regarded the fast local memory and can be used to exploit locality under the software control. In this thesis, we give an example to show how the SPM on C64 can be used to optimize FFT. This optimization is done by hand and highly dependent on the application property. If these kind of optimizations can be done by the compiler, it should be very attractive. One possible solution is to consider the SPM as a second level register files and then the register allocation algorithm can be modified to allocate SPM.

There are several automatic FFT code generators like FFTW and SPIRAL have been implemented for the traditional cache-based architecture. But the automatic FFT code generator for the multi-core architecture with explicit memory hierarchy like C64 is not available. Design and implementation of such FFT code generator is a very promising and challenge topic.

In this thesis, we assume all input data can fit into the on-chip GM. But in reality, especially for large input size, the input data cannot fit into the on-chip GM and need to be stored in off-chip DRAM with a long latency. If we do the FFT on
DRAM, the performance would be very poor because of the long latency of DRAM access. In order to improve the performance, we can divide this large FFT into several small blocks whose input data can fit into the on-chip GM. The purpose is to do the computation and the data movement simultaneously. The idea is that: we first need to move the data of the first block from off-chip to on-chip GM. Then we can do the computation of the first block on GM, and at the same time, we can move the input data of the second block from off-chip DRAM to on-chip GM. By doing this, the high latency of loading input data from DRAM to GM has been hided.
BIBLIOGRAPHY


## Appendix A

### GLOSSARY

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>Instruction-level parallelism</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread-level parallelism</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier transform</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier transform</td>
</tr>
<tr>
<td>TU</td>
<td>Thread unit</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating point unit</td>
</tr>
<tr>
<td>SPM</td>
<td>Scratch-pad memory</td>
</tr>
<tr>
<td>GM</td>
<td>Global interleaved memory</td>
</tr>
</tbody>
</table>
Appendix B

IMPORTANT DATA STRUCTURES

B.1 Kernel Functions for FFT Implementation on C64

• Kernel Function for 2-point Work Unit

– r2v1:
  // load w, y, x
  LDM_02_DBL(&w[w_ind], w_r, w_i);
  LDM_02_DBL(&x[b_ind], b_r, b_i);
  LDM_02_DBL(&x[a_ind], a_r, a_i);

  // do inplace forward radix-2 butterfly
  RADIX2_F(a_r, a_i, b_r, b_i, w_r, w_i);

  // store x, y
  STM_02_DBL(&x[a_ind], a_r, a_i);
  STM_02_DBL(&x[b_ind], b_r, b_i);

– r2v2:
  // load w, y, x
  LDM_02_DBL(&w[w0_ind], w0_r, w0_i);
  LDM_02_DBL(&w[w1_ind], w1_r, w1_i);
  LDM_04_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i);
  LDM_04_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i);

  // (a0, b0) do inplace forward radix-2 butterfly
  RADIX2_F(a0_r, a0_i, b0_r, b0_i, w0_r, w0_i);

  // (a1, b1) do inplace forward radix-2 butterfly
  RADIX2_F(a1_r, a1_i, b1_r, b1_i, w1_r, w1_i);

  // store x, y
STM_04_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i);
STM_04_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i);

- r2v4:
  // load w,y,x
  LDM_02_DBL(&w[w0_ind], w0_r, w0_i);
  LDM_02_DBL(&w[w1_ind], w1_r, w1_i);
  LDM_02_DBL(&w[w2_ind], w2_r, w2_i);
  LDM_02_DBL(&w[w3_ind], w3_r, w3_i);
  LDM_08_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i,
              a2_r, a2_i, a3_r, a3_i);
  LDM_08_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i,
              b2_r, b2_i, b3_r, b3_i);

  // (a0,b0) do inplace forward radix-2 butterfly
  RADIX2_F(a0_r, a0_i, b0_r, b0_i, w0_r, w0_i);

  // (a1,b1) do inplace forward radix-2 butterfly
  RADIX2_F(a1_r, a1_i, b1_r, b1_i, w1_r, w1_i);

  // (a2,b2) do inplace forward radix-2 butterfly
  RADIX2_F(a2_r, a2_i, b2_r, b2_i, w2_r, w2_i);

  // (a3,b3) do inplace forward radix-2 butterfly
  RADIX2_F(a3_r, a3_i, b3_r, b3_i, w3_r, w3_i);

  // store x,y
  STM_08_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i,
              a2_r, a2_i, a3_r, a3_i);
  STM_08_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i,
              b2_r, b2_i, b3_r, b3_i);

- r2v8:
  // load w,y,x
  LDM_02_DBL(&w[w0_ind], w0_r, w0_i);
  LDM_02_DBL(&w[w1_ind], w1_r, w1_i);
  LDM_02_DBL(&w[w2_ind], w2_r, w2_i);
  LDM_02_DBL(&w[w3_ind], w3_r, w3_i);
  LDM_02_DBL(&w[w4_ind], w4_r, w4_i);
  LDM_02_DBL(&w[w5_ind], w5_r, w5_i);
  LDM_02_DBL(&w[w6_ind], w6_r, w6_i);
  LDM_02_DBL(&w[w7_ind], w7_r, w7_i);

53
LDM_16_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i,
           a2_r, a2_i, a3_r, a3_i, a4_r, a4_i,
           a5_r, a5_i, a6_r, a6_i, a7_r, a7_i);
LDM_16_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i,
           b2_r, b2_i, b3_r, b3_i, b4_r, b4_i,
           b5_r, b5_i, b6_r, b6_i, b7_r, b7_i);

// (a0,b0) do inplace forward radix-2 butterfly
RADIX2_F(a0_r, a0_i, b0_r, b0_i, w0_r, w0_i);

// (a1,b1) do inplace forward radix-2 butterfly
RADIX2_F(a1_r, a1_i, b1_r, b1_i, w1_r, w1_i);

// (a2,b2) do inplace forward radix-2 butterfly
RADIX2_F(a2_r, a2_i, b2_r, b2_i, w2_r, w2_i);

// (a3,b3) do inplace forward radix-2 butterfly
RADIX2_F(a3_r, a3_i, b3_r, b3_i, w3_r, w3_i);

// (a4,b4) do inplace forward radix-2 butterfly
RADIX2_F(a4_r, a4_i, b4_r, b4_i, w4_r, w4_i);

// (a5,b5) do inplace forward radix-2 butterfly
RADIX2_F(a5_r, a5_i, b5_r, b5_i, w5_r, w5_i);

// (a6,b6) do inplace forward radix-2 butterfly
RADIX2_F(a6_r, a6_i, b6_r, b6_i, w6_r, w6_i);

// (a7,b7) do inplace forward radix-2 butterfly
RADIX2_F(a7_r, a7_i, b7_r, b7_i, w7_r, w7_i);

// store x,y
STM_16_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i,
           a2_r, a2_i, a3_r, a3_i,a4_r, a4_i,
           a5_r, a5_i, a6_r, a6_i, a7_r, a7_i);
STM_16_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i,
           b2_r, b2_i, b3_r, b3_i,b4_r, b4_i,
           b5_r, b5_i, b6_r, b6_i, b7_r, b7_i);

• Kernel Function for 4-point Work Unit
- r4v1:
  // load w
  LDM_02_DBL(&w[wa_p0_ind], wa_p0_r, wa_p0_i);
  LDM_02_DBL(&w[wc_p0_ind], wc_p0_r, wc_p0_i);
  LDM_02_DBL(&w[wa_p1_ind], wa_p1_r, wa_p1_i);
  LDM_02_DBL(&w[wb_p1_ind], wb_p1_r, wb_p1_i);

  // load a,b,c,d
  LDM_02_DBL(&x[a_ind], a_r, a_i);
  LDM_02_DBL(&x[b_ind], b_r, b_i);
  LDM_02_DBL(&x[c_ind], c_r, c_i);
  LDM_02_DBL(&x[d_ind], d_r, d_i);

  // (a,b) do inplace forward radix-2 butterfly
  RADIX2_F(a_r, a_i, b_r, b_i, wa_p0_r, wa_p0_i);

  // (c,d) do inplace forward radix-2 butterfly
  RADIX2_F(c_r, c_i, d_r, d_i, wc_p0_r, wc_p0_i);

  // (a,c) do inplace forward radix-2 butterfly
  RADIX2_F(a_r, a_i, c_r, c_i, wa_p1_r, wa_p1_i);

  // (b,d) do inplace forward radix-2 butterfly
  RADIX2_F(b_r, b_i, d_r, d_i, wb_p1_r, wb_p1_i);

  // store a,b,c,d
  STM_02_DBL(&x[a_ind], a_r, a_i);
  STM_02_DBL(&x[b_ind], b_r, b_i);
  STM_02_DBL(&x[c_ind], c_r, c_i);
  STM_02_DBL(&x[d_ind], d_r, d_i);

- r4v2:
  // load w
  LDM_02_DBL(&w[wa0_p0_ind], wa0_p0_r, wa0_p0_i);
  LDM_02_DBL(&w[wa1_p0_ind], wa1_p0_r, wa1_p0_i);
  LDM_02_DBL(&w[wc0_p0_ind], wc0_p0_r, wc0_p0_i);
  LDM_02_DBL(&w[wc1_p0_ind], wc1_p0_r, wc1_p0_i);

  LDM_02_DBL(&w[wa0_p1_ind], wa0_p1_r, wa0_p1_i);
  LDM_02_DBL(&w[wa1_p1_ind], wa1_p1_r, wa1_p1_i);
  LDM_02_DBL(&w[wb0_p1_ind], wb0_p1_r, wb0_p1_i);
  LDM_02_DBL(&w[wb1_p1_ind], wb1_p1_r, wb1_p1_i);
// load a, b, c, d
LDM_04_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i);
LDM_04_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i);
LDM_04_DBL(&x[c_ind], c0_r, c0_i, c1_r, c1_i);
LDM_04_DBL(&x[d_ind], d0_r, d0_i, d1_r, d1_i);

// (a, b) do inplace forward radix-2 butterfly
RADIX2_F(a0_r, a0_i, b0_r, b0_i, wa0_p0_r, wa0_p0_i);
RADIX2_F(a1_r, a1_i, b1_r, b1_i, wa1_p0_r, wa1_p0_i);

// (c, d) do inplace forward radix-2 butterfly
RADIX2_F(c0_r, c0_i, d0_r, d0_i, wc0_p0_r, wc0_p0_i);
RADIX2_F(c1_r, c1_i, d1_r, d1_i, wc1_p0_r, wc1_p0_i);

// (a, c) do inplace forward radix-2 butterfly
RADIX2_F(a0_r, a0_i, c0_r, c0_i, wa0_p1_r, wa0_p1_i);
RADIX2_F(a1_r, a1_i, c1_r, c1_i, wa1_p1_r, wa1_p1_i);

// (b, d) do inplace forward radix-2 butterfly
RADIX2_F(b0_r, b0_i, d0_r, d0_i, wb0_p1_r, wb0_p1_i);
RADIX2_F(b1_r, b1_i, d1_r, d1_i, wb1_p1_r, wb1_p1_i);

// store a, b, c, d
STM_04_DBL(&x[a_ind], a0_r, a0_i, a1_r, a1_i);
STM_04_DBL(&x[b_ind], b0_r, b0_i, b1_r, b1_i);
STM_04_DBL(&x[c_ind], c0_r, c0_i, c1_r, c1_i);
STM_04_DBL(&x[d_ind], d0_r, d0_i, d1_r, d1_i);

- r8v1:
  // load w
  LDM_02_DBL(&w[wa_p0_ind], wa_p0_r, wa_p0_i);
  LDM_02_DBL(&w[wc_p0_ind], wc_p0_r, wc_p0_i);
  LDM_02_DBL(&w[we_p0_ind], we_p0_r, we_p0_i);
  LDM_02_DBL(&w[wg_p0_ind], wg_p0_r, wg_p0_i);
  LDM_02_DBL(&w[wa_p1_ind], wa_p1_r, wa_p1_i);
  LDM_02_DBL(&w[wb_p1_ind], wb_p1_r, wb_p1_i);
  LDM_02_DBL(&w[we_p1_ind], we_p1_r, we_p1_i);
LDM_02_DBL(&w[wf_p1_ind], wf_p1_r, wf_p1_i);
LDM_02_DBL(&w[wa_p2_ind], wa_p2_r, wa_p2_i);
LDM_02_DBL(&w[wb_p2_ind], wb_p2_r, wb_p2_i);
LDM_02_DBL(&w[wc_p2_ind], wc_p2_r, wc_p2_i);
LDM_02_DBL(&w[wd_p2_ind], wd_p2_r, wd_p2_i);

// load a,b,c,d,e,f,g,h
LDM_02_DBL(&x[a_ind], a_r, a_i);
LDM_02_DBL(&x[b_ind], b_r, b_i);
LDM_02_DBL(&x[c_ind], c_r, c_i);
LDM_02_DBL(&x[d_ind], d_r, d_i);
LDM_02_DBL(&x[e_ind], e_r, e_i);
LDM_02_DBL(&x[f_ind], f_r, f_i);
LDM_02_DBL(&x[g_ind], g_r, g_i);
LDM_02_DBL(&x[h_ind], h_r, h_i);

// PASS 0
// (a,b) do inplace forward radix-2 butterfly
RADIX2_F(a_r, a_i, b_r, b_i, wa_p0_r, wa_p0_i);
// (c,d) do inplace forward radix-2 butterfly
RADIX2_F(c_r, c_i, d_r, d_i, wc_p0_r, wc_p0_i);
// (e,f) do inplace forward radix-2 butterfly
RADIX2_F(e_r, e_i, f_r, f_i, we_p0_r, we_p0_i);
// (g,h) do inplace forward radix-2 butterfly
RADIX2_F(g_r, g_i, h_r, h_i, wg_p0_r, wg_p0_i);

// PASS 1
// (a,c) do inplace forward radix-2 butterfly
RADIX2_F(a_r, a_i, c_r, c_i, wa_p1_r, wa_p1_i);
// (b,d) do inplace forward radix-2 butterfly
RADIX2_F(b_r, b_i, d_r, d_i, wb_p1_r, wb_p1_i);
// (e,g) do inplace forward radix-2 butterfly
RADIX2_F(e_r, e_i, g_r, g_i, we_p1_r, we_p1_i);
// (f,h) do inplace forward radix-2 butterfly
RADIX2_F(f_r, f_i, h_r, h_i, wf_p1_r, wf_p1_i);

// PASS 2
// (a,e) do inplace forward radix-2 butterfly
RADIX2_F(a_r, a_i, e_r, e_i, wa_p2_r, wa_p2_i);
// (b,f) do inplace forward radix-2 butterfly
RADIX2_F(b_r, b_i, f_r, f_i, wb_p2_r, wb_p2_i);
// (c,g) do inplace forward radix-2 butterfly
RADIX2_F(c_r, c_i, g_r, g_i, wc_p2_r, wc_p2_i);
// (d,h) do inplace forward radix-2 butterfly
RADIX2_F(d_r, d_i, h_r, h_i, wd_p2_r, wd_p2_i);

// store a,b,c,d,f,g,h
STM_02_DBL(&x[a_ind], a_r, a_i);
STM_02_DBL(&x[b_ind], b_r, b_i);
STM_02_DBL(&x[c_ind], c_r, c_i);
STM_02_DBL(&x[d_ind], d_r, d_i);
STM_02_DBL(&x[e_ind], e_r, e_i);
STM_02_DBL(&x[f_ind], f_r, f_i);
STM_02_DBL(&x[g_ind], g_r, g_i);
STM_02_DBL(&x[h_ind], h_r, h_i);

• Kernel Function for 16-point Work Unit

  - r16v1-first:
    // load a,b,c,d,e,f,g,h
    LDM_16_DBL(&x[ind<<1], a_r, a_i, b_r, b_i, c_r, c_i,
               d_r, d_i, e_r, e_i, f_r, f_i, g_r, g_i, h_r,
               h_i);
    // load i,j,k,l,m,n,o,p
    LDM_16_DBL(&x[(ind<<1)+16], i_r, i_i, j_r, j_i, k_r,
               k_i, l_r, l_i, m_r, m_i, n_r, n_i, o_r, o_i,
               p_r, p_i);

    // PASS 0 a b c d e f g h i j k l m n o p

    // (a,b) do inplace forward radix-2 butterfly
    RADIX2_W_0_F(a_r, a_i, b_r, b_i);
    // (c,d) do inplace forward radix-2 butterfly
    RADIX2_W_0_F(c_r, c_i, d_r, d_i);
    // (e,f) do inplace forward radix-2 butterfly
    RADIX2_W_0_F(e_r, e_i, f_r, f_i);
    // (g,h) do inplace forward radix-2 butterfly
    RADIX2_W_0_F(g_r, g_i, h_r, h_i);
    // (i,j) do inplace forward radix-2 butterfly
    RADIX2_W_0_F(i_r, i_i, j_r, j_i);
    // (k,l) do inplace forward radix-2 butterfly

RADIX2_W_0_F(k_r, k_i, l_r, l_i);
// (m,n) do inplace forward radix-2 butterfly
RADIX2_W_0_F(m_r, m_i, n_r, n_i);
// (o,p) do inplace forward radix-2 butterfly
RADIX2_W_0_F(o_r, o_i, p_r, p_i);

// PASS 1 ab cd ef gh ij kl mn op

// (a,c) do inplace forward radix-2 butterfly
RADIX2_W_0_F(a_r, a_i, c_r, c_i);
// (b,d) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(b_r, b_i, d_r, d_i);
// (e,g) do inplace forward radix-2 butterfly
RADIX2_W_0_F(e_r, e_i, g_r, g_i);
// (f,h) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(f_r, f_i, h_r, h_i);
// (i,k) do inplace forward radix-2 butterfly
RADIX2_W_0_F(i_r, i_i, k_r, k_i);
// (j,l) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(j_r, j_i, l_r, l_i);
// (m,o) do inplace forward radix-2 butterfly
RADIX2_W_0_F(m_r, m_i, o_r, o_i);
// (n,p) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(n_r, n_i, p_r, p_i);

// PASS 2 abcd efgh ijkl mnop

// (a,e) do inplace forward radix-2 butterfly
RADIX2_W_0_F(a_r, a_i, e_r, e_i);
// (b,f) do inplace forward radix-2 butterfly
RADIX2_F(b_r, b_i, f_r, f_i, W16_2_r, W16_2_i);
// (c,g) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(c_r, c_i, g_r, g_i);
// (d,h) do inplace forward radix-2 butterfly
RADIX2_F(d_r, d_i, h_r, h_i, W16_6_r, W16_6_i);
// (i,m) do inplace forward radix-2 butterfly
RADIX2_W_0_F(i_r, i_i, m_r, m_i);
// (j,n) do inplace forward radix-2 butterfly
RADIX2_F(j_r, j_i, n_r, n_i, W16_2_r, W16_2_i);
// (k,o) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(k_r, k_i, o_r, o_i);
// (l,p) do inplace forward radix-2 butterfly
RADIX2_F(l_r, l_i, p_r, p_i, W16_6_r, W16_6_i);

// PASS 3 abcdefgh ijklnmop

// (a,i) do inplace forward radix-2 butterfly
RADIX2_W_0_F(a_r, a_i, i_r, i_i);
// (b,j) do inplace forward radix-2 butterfly
RADIX2_F(b_r, b_i, j_r, j_i, W16_1_r, W16_1_i);
// (c,k) do inplace forward radix-2 butterfly
RADIX2_F(c_r, c_i, k_r, k_i, W16_2_r, W16_2_i);
// (d,l) do inplace forward radix-2 butterfly
RADIX2_F(d_r, d_i, l_r, l_i, W16_3_r, W16_3_i);
// (e,m) do inplace forward radix-2 butterfly
RADIX2_W8_2_F(e_r, e_i, m_r, m_i);
// (f,n) do inplace forward radix-2 butterfly
RADIX2_F(f_r, f_i, n_r, n_i, W16_5_r, W16_5_i);
// (g,o) do inplace forward radix-2 butterfly
RADIX2_F(g_r, g_i, o_r, o_i, W16_6_r, W16_6_i);
// (h,p) do inplace forward radix-2 butterfly
RADIX2_F(h_r, h_i, p_r, p_i, W16_7_r, W16_7_i);

// store a,b,c,d,e,f,g,h
STM_16_DBL(&x[ind<<1], a_r, a_i, b_r, b_i, c_r, c_i,
           d_r, d_i, e_r, e_i, f_r, f_i, g_r, g_i,
           h_r, h_i);

// store i,j,k,l,m,n,o,p
STM_16_DBL(&x[(ind<<1)+16], i_r, i_i, j_r, j_i, k_r,
           k_i, l_r, l_i, m_r, m_i, n_r, n_i, o_r, o_i,
           p_r, p_i);

– r16v1:
// load 1st 8-point data
LDM_02_DBL(&x[d00_ind], d00_r, d00_i);
LDM_02_DBL(&x[d01_ind], d01_r, d01_i);
LDM_02_DBL(&x[d02_ind], d02_r, d02_i);
LDM_02_DBL(&x[d03_ind], d03_r, d03_i);
LDM_02_DBL(&x[d04_ind], d04_r, d04_i);
LDM_02_DBL(&x[d05_ind], d05_r, d05_i);
LDM_02_DBL(&x[d06_ind], d06_r, d06_i);
LDM_02_DBL(&x[d07_ind], d07_r, d07_i);
// load w for pass 0,1,2
LDM_02_DBL(&w[w_p0_ind_0], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p1_ind_0], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p1_ind_1], w_p1_2, w_p1_3);
LDM_02_DBL(&w[w_p2_ind_0], w_p2_0, w_p2_1);
LDM_02_DBL(&w[w_p2_ind_1], w_p2_2, w_p2_3);
LDM_02_DBL(&w[w_p2_ind_2], w_p2_4, w_p2_5);
LDM_02_DBL(&w[w_p2_ind_3], w_p2_6, w_p2_7);

// pass 0,1,2 butterfly operations on 1st 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);

RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);

RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

// store 1st 8-point intermediate result to SPM buffer
STM_02_DBL(&spm_buf[0], d00_r, d00_i);
STM_02_DBL(&spm_buf[2], d01_r, d01_i);
STM_02_DBL(&spm_buf[4], d02_r, d02_i);
STM_02_DBL(&spm_buf[6], d03_r, d03_i);
STM_02_DBL(&spm_buf[8], d04_r, d04_i);
STM_02_DBL(&spm_buf[10], d05_r, d05_i);
STM_02_DBL(&spm_buf[12], d06_r, d06_i);
STM_02_DBL(&spm_buf[14], d07_r, d07_i);

// load 2nd 8-point data
LDM_02_DBL(&x[d08_ind], d00_r, d00_i);
LDM_02_DBL(&x[d09_ind], d01_r, d01_i);
LDM_02_DBL(&x[d10_ind], d02_r, d02_i);
LDM_02_DBL(&x[d11_ind], d03_r, d03_i);
LDM_02_DBL(&x[d12_ind], d04_r, d04_i);
LDM_02_DBL(&x[d13_ind], d05_r, d05_i);
LDM_02_DBL(&x[d14_ind], d06_r, d06_i);
LDM_02_DBL(&x[d15_ind], d07_r, d07_i);

// pass 0,1,2 butterfly operations on 2nd 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);

RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);

RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

//store 2nd 8-point intermediate result to SPM buffer
STM_02_DBL(&spm_buf[16], d00_r, d00_i);
STM_02_DBL(&spm_buf[18], d01_r, d01_i);
STM_02_DBL(&spm_buf[20], d02_r, d02_i);
STM_02_DBL(&spm_buf[22], d03_r, d03_i);
STM_02_DBL(&spm_buf[24], d04_r, d04_i);
STM_02_DBL(&spm_buf[26], d05_r, d05_i);
STM_02_DBL(&spm_buf[28], d06_r, d06_i);
STM_02_DBL(&spm_buf[30], d07_r, d07_i);

// load intermediate result from SPM buffer

// load 1st 8-point from SPM buffer
LDM_02_DBL(&spm_buf[0], d00_r, d00_i);
LDM_02_DBL(&spm_buf[16], d01_r, d01_i);
LDM_02_DBL(&spm_buf[2], d02_r, d02_i);
LDM_02_DBL(&spm_buf[18], d03_r, d03_i);
LDM_02_DBL(&spm_buf[4], d04_r, d04_i);
LDM_02_DBL(&spm_buf[20], d05_r, d05_i);
LDM_02_DBL(&spm_buf[6], d06_r, d06_i);
LDM_02_DBL(&spm_buf[22], d07_r, d07_i);

//load w for pass 3
LDM_02_DBL(&w[w_p3_ind_00], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p3_ind_01], w_p0_2, w_p0_3);
LDM_02_DBL(&w[w_p3_ind_02], w_p0_4, w_p0_5);
LDM_02_DBL(&w[w_p3_ind_03], w_p0_6, w_p0_7);
LDM_02_DBL(&w[w_p3_ind_04], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p3_ind_05], w_p1_2, w_p1_3);
LDM_02_DBL(&w[w_p3_ind_06], w_p1_4, w_p1_5);
LDM_02_DBL(&w[w_p3_ind_07], w_p1_6, w_p1_7);

//pass 3 butterfly operations on 1st 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_2, w_p0_3);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_4, w_p0_5);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_6, w_p0_7);

//store the 1st 8-point to memory
STM_02_DBL(&x[d00_ind], d00_r, d00_i);
STM_02_DBL(&x[d08_ind], d01_r, d01_i);
STM_02_DBL(&x[d01_ind], d02_r, d02_i);
STM_02_DBL(&x[d09_ind], d03_r, d03_i);
STM_02_DBL(&x[d02_ind], d04_r, d04_i);
STM_02_DBL(&x[d10_ind], d05_r, d05_i);
STM_02_DBL(&x[d03_ind], d06_r, d06_i);
STM_02_DBL(&x[d11_ind], d07_r, d07_i);

// load 2nd 8-point from SPM
LDM_02_DBL(&spm_buf[8], d00_r, d00_i);
LDM_02_DBL(&spm_buf[24], d01_r, d01_i);
LDM_02_DBL(&spm_buf[10], d02_r, d02_i);
LDM_02_DBL(&spm_buf[26], d03_r, d03_i);
LDM_02_DBL(&spm_buf[12], d04_r, d04_i);
LDM_02_DBL(&spm_buf[28], d05_r, d05_i);
LDM_02_DBL(&spm_buf[14], d06_r, d06_i);
LDM_02_DBL(&spm_buf[30], d07_r, d07_i);

//pass 3 butterfly operations on 2nd 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p1_0, w_p1_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p1_4, w_p1_5);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p1_6, w_p1_7);

//store the 2nd 8-point to memory
STM_02_DBL(&x[d04_ind], d00_r, d00_i);
STM_02_DBL(&x[d12_ind], d01_r, d01_i);
STM_02_DBL(&x[d05_ind], d02_r, d02_i);
STM_02_DBL(&x[d13_ind], d03_r, d03_i);
STM_02_DBL(&x[d06_ind], d04_r, d04_i);
STM_02_DBL(&x[d14_ind], d05_r, d05_i);
STM_02_DBL(&x[d07_ind], d06_r, d06_i);
STM_02_DBL(&x[d15_ind], d07_r, d07_i);

• Kernel Function for 32-point Work Unit
  – r32v1:
    // load 1st 8-point data
    LDM_02_DBL(&x[d00_ind], d00_r, d00_i);
    LDM_02_DBL(&x[d01_ind], d01_r, d01_i);
    LDM_02_DBL(&x[d02_ind], d02_r, d02_i);
    LDM_02_DBL(&x[d03_ind], d03_r, d03_i);
    LDM_02_DBL(&x[d04_ind], d04_r, d04_i);
    LDM_02_DBL(&x[d05_ind], d05_r, d05_i);
    LDM_02_DBL(&x[d06_ind], d06_r, d06_i);
    LDM_02_DBL(&x[d07_ind], d07_r, d07_i);

    // load w for pass 0,1,2
    LDM_02_DBL(&w[w_p0_ind_0], w_p0_0, w_p0_1);
    LDM_02_DBL(&w[w_p1_ind_0], w_p1_0, w_p1_1);
    LDM_02_DBL(&w[w_p1_ind_1], w_p1_2, w_p1_3);
    LDM_02_DBL(&w[w_p2_ind_0], w_p2_0, w_p2_1);
    LDM_02_DBL(&w[w_p2_ind_1], w_p2_2, w_p2_3);
    LDM_02_DBL(&w[w_p2_ind_2], w_p2_4, w_p2_5);
LDM_02_DBL(&w[w_p2_ind_3], w_p2_6, w_p2_7);

// pass 0,1,2 butterfly operations on 1st 8-point data

RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);

RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);

RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

// store the 1st 8-point intermediate result to SPM buffer

STM_02_DBL(&spm_buf[0], d00_r, d00_i);
STM_02_DBL(&spm_buf[2], d01_r, d01_i);
STM_02_DBL(&spm_buf[4], d02_r, d02_i);
STM_02_DBL(&spm_buf[6], d03_r, d03_i);
STM_02_DBL(&spm_buf[8], d04_r, d04_i);
STM_02_DBL(&spm_buf[10], d05_r, d05_i);
STM_02_DBL(&spm_buf[12], d06_r, d06_i);
STM_02_DBL(&spm_buf[14], d07_r, d07_i);

// load 2nd 8-point data

LDM_02_DBL(&x[d08_ind], d00_r, d00_i);
LDM_02_DBL(&x[d09_ind], d01_r, d01_i);
LDM_02_DBL(&x[d10_ind], d02_r, d02_i);
LDM_02_DBL(&x[d11_ind], d03_r, d03_i);
LDM_02_DBL(&x[d12_ind], d04_r, d04_i);
LDM_02_DBL(&x[d13_ind], d05_r, d05_i);
LDM_02_DBL(&x[d14_ind], d06_r, d06_i);
LDM_02_DBL(&x[d15_ind], d07_r, d07_i);
// pass 0,1,2 butterfly operations on 2nd 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);
RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

// store the 2nd 8-point intermediate result to SPM buffer
STM_02_DBL(&spm_buf[16], d00_r, d00_i);
STM_02_DBL(&spm_buf[18], d01_r, d01_i);
STM_02_DBL(&spm_buf[20], d02_r, d02_i);
STM_02_DBL(&spm_buf[22], d03_r, d03_i);
STM_02_DBL(&spm_buf[24], d04_r, d04_i);
STM_02_DBL(&spm_buf[26], d05_r, d05_i);
STM_02_DBL(&spm_buf[28], d06_r, d06_i);
STM_02_DBL(&spm_buf[30], d07_r, d07_i);

// load 3rd 8-point data
LDM_02_DBL(&x[d16_ind], d00_r, d00_i);
LDM_02_DBL(&x[d17_ind], d01_r, d01_i);
LDM_02_DBL(&x[d18_ind], d02_r, d02_i);
LDM_02_DBL(&x[d19_ind], d03_r, d03_i);
LDM_02_DBL(&x[d20_ind], d04_r, d04_i);
LDM_02_DBL(&x[d21_ind], d05_r, d05_i);
LDM_02_DBL(&x[d22_ind], d06_r, d06_i);
LDM_02_DBL(&x[d23_ind], d07_r, d07_i);

// pass 0,1,2 butterfly operations on 3rd 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);

RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);

RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

// store the 3rd 8-point intermediate result to SPM buffer
STM_02_DBL(&spm_buf[32], d00_r, d00_i);
STM_02_DBL(&spm_buf[34], d01_r, d01_i);
STM_02_DBL(&spm_buf[36], d02_r, d02_i);
STM_02_DBL(&spm_buf[38], d03_r, d03_i);
STM_02_DBL(&spm_buf[40], d04_r, d04_i);
STM_02_DBL(&spm_buf[42], d05_r, d05_i);
STM_02_DBL(&spm_buf[44], d06_r, d06_i);
STM_02_DBL(&spm_buf[46], d07_r, d07_i);

// load 4th 8-point data
LDM_02_DBL(&x[d24_ind], d00_r, d00_i);
LDM_02_DBL(&x[d25_ind], d01_r, d01_i);
LDM_02_DBL(&x[d26_ind], d02_r, d02_i);
LDM_02_DBL(&x[d27_ind], d03_r, d03_i);
LDM_02_DBL(&x[d28_ind], d04_r, d04_i);
LDM_02_DBL(&x[d29_ind], d05_r, d05_i);
LDM_02_DBL(&x[d30_ind], d06_r, d06_i);
LDM_02_DBL(&x[d31_ind], d07_r, d07_i);

// pass 0,1,2 butterfly operations on 4th 8-point data
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d04_r, d04_i, d05_r, d05_i, w_p0_0, w_p0_1);
RADIX2_F(d06_r, d06_i, d07_r, d07_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);
RADIX2_F(d04_r, d04_i, d06_r, d06_i, w_p1_0, w_p1_1);
RADIX2_F(d05_r, d05_i, d07_r, d07_i, w_p1_2, w_p1_3);

RADIX2_F(d00_r, d00_i, d04_r, d04_i, w_p2_0, w_p2_1);
RADIX2_F(d01_r, d01_i, d05_r, d05_i, w_p2_2, w_p2_3);
RADIX2_F(d02_r, d02_i, d06_r, d06_i, w_p2_4, w_p2_5);
RADIX2_F(d03_r, d03_i, d07_r, d07_i, w_p2_6, w_p2_7);

// store the 4th 8-point intermediate result to SPM buffer
STM_02_DBL(&spm_buf[48], d00_r, d00_i);
STM_02_DBL(&spm_buf[50], d01_r, d01_i);
STM_02_DBL(&spm_buf[52], d02_r, d02_i);
STM_02_DBL(&spm_buf[54], d03_r, d03_i);
STM_02_DBL(&spm_buf[56], d04_r, d04_i);
STM_02_DBL(&spm_buf[58], d05_r, d05_i);
STM_02_DBL(&spm_buf[60], d06_r, d06_i);
STM_02_DBL(&spm_buf[62], d07_r, d07_i);

// SECOND ROUND

// load 1st 4-point data from SPM
LDM_02_DBL(&spm_buf[0], d00_r, d00_i);
LDM_02_DBL(&spm_buf[16], d01_r, d01_i);
LDM_02_DBL(&spm_buf[32], d02_r, d02_i);
LDM_02_DBL(&spm_buf[48], d03_r, d03_i);

// 1st group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_00], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_00], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_08], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 1st group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

// store the 1st 4-point data to memory
STM_02_DBL(&x[d00_ind], d00_r, d00_i);
STM_02_DBL(&x[d08_ind], d01_r, d01_i);
STM_02_DBL(&x[d16_ind], d02_r, d02_i);
STM_02_DBL(&x[d24_ind], d03_r, d03_i);

// load 2nd 4-point data from SPM
LDM_02_DBL(&spm_buf[2], d00_r, d00_i);
LDM_02_DBL(&spm_buf[18], d01_r, d01_i);
LDM_02_DBL(&spm_buf[34], d02_r, d02_i);
LDM_02_DBL(&spm_buf[50], d03_r, d03_i);

// 2nd group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_01], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_01], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_09], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 2nd group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

// store the 2nd 4-point data to memory
STM_02_DBL(&x[d01_ind], d00_r, d00_i);
STM_02_DBL(&x[d09_ind], d01_r, d01_i);
STM_02_DBL(&x[d17_ind], d02_r, d02_i);
STM_02_DBL(&x[d25_ind], d03_r, d03_i);

// load 3rd 4-point data from SPM
LDM_02_DBL(&spm_buf[4], d00_r, d00_i);
LDM_02_DBL(&spm_buf[20], d01_r, d01_i);
LDM_02_DBL(&spm_buf[36], d02_r, d02_i);
LDM_02_DBL(&spm_buf[52], d03_r, d03_i);

// 3rd group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_02], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_02], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_10], w_p1_2, w_p1_3);
// pass 3,4 butterfly operations on 3rd group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_1, w_p1_3);

// store the 3rd 4-point data to memory
STM_02_DBL(&x[d02_ind], d00_r, d00_i);
STM_02_DBL(&x[d10_ind], d01_r, d01_i);
STM_02_DBL(&x[d18_ind], d02_r, d02_i);
STM_02_DBL(&x[d26_ind], d03_r, d03_i);

// load 4th 4-point data from SPM
LDM_02_DBL(&spm_buf[6], d00_r, d00_i);
LDM_02_DBL(&spm_buf[22], d01_r, d01_i);
LDM_02_DBL(&spm_buf[38], d02_r, d02_i);
LDM_02_DBL(&spm_buf[54], d03_r, d03_i);

// 4th group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_03], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_03], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_11], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 4th group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_1, w_p1_3);

// store the 4th 4-point data to memory
STM_02_DBL(&x[d03_ind], d00_r, d00_i);
STM_02_DBL(&x[d11_ind], d01_r, d01_i);
STM_02_DBL(&x[d19_ind], d02_r, d02_i);
STM_02_DBL(&x[d27_ind], d03_r, d03_i);

// load 5th 4-point data from SPM
LDM_02_DBL(&spm_buf[8], d00_r, d00_i);
LDM_02_DBL(&spm_buf[24], d01_r, d01_i);
LDM_02_DBL(&spm_buf[40], d02_r, d02_i);
LDM_02_DBL(&spm_buf[56], d03_r, d03_i);
// 5th group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_04], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_04], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_12], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 5th group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

// store the 5th 4-point data to memory
STM_02_DBL(&x[d04_ind], d00_r, d00_i);
STM_02_DBL(&x[d12_ind], d01_r, d01_i);
STM_02_DBL(&x[d20_ind], d02_r, d02_i);
STM_02_DBL(&x[d28_ind], d03_r, d03_i);

// load 6th 4-point data from SPM
LDM_02_DBL(&spm_buf[10], d00_r, d00_i);
LDM_02_DBL(&spm_buf[26], d01_r, d01_i);
LDM_02_DBL(&spm_buf[42], d02_r, d02_i);
LDM_02_DBL(&spm_buf[58], d03_r, d03_i);

// 6th group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_05], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_05], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_13], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 6th group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

// store the 6th 4-point data to memory
STM_02_DBL(&x[d05_ind], d00_r, d00_i);
STM_02_DBL(&x[d13_ind], d01_r, d01_i);
STM_02_DBL(&x[d21_ind], d02_r, d02_i);
STM_02_DBL(&x[d29_ind], d03_r, d03_i);

// load 7th 4-point data from SPM
LDM_02_DBL(&spm_buf[12], d00_r, d00_i);
LDM_02_DBL(&spm_buf[28], d01_r, d01_i);
LDM_02_DBL(&spm_buf[44], d02_r, d02_i);
LDM_02_DBL(&spm_buf[60], d03_r, d03_i);

// 7th group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_06], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_06], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_14], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 7th group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

// store the 7th 4-point data to memory
STM_02_DBL(&x[d06_ind], d00_r, d00_i);
STM_02_DBL(&x[d14_ind], d01_r, d01_i);
STM_02_DBL(&x[d22_ind], d02_r, d02_i);
STM_02_DBL(&x[d30_ind], d03_r, d03_i);

// load 8th 4-point data from SPM
LDM_02_DBL(&spm_buf[14], d00_r, d00_i);
LDM_02_DBL(&spm_buf[30], d01_r, d01_i);
LDM_02_DBL(&spm_buf[46], d02_r, d02_i);
LDM_02_DBL(&spm_buf[62], d03_r, d03_i);

// 7th group: load w for pass 3,4
LDM_02_DBL(&w[w_p3_ind_07], w_p0_0, w_p0_1);
LDM_02_DBL(&w[w_p4_ind_07], w_p1_0, w_p1_1);
LDM_02_DBL(&w[w_p4_ind_15], w_p1_2, w_p1_3);

// pass 3,4 butterfly operations on 8th group
RADIX2_F(d00_r, d00_i, d01_r, d01_i, w_p0_0, w_p0_1);
RADIX2_F(d02_r, d02_i, d03_r, d03_i, w_p0_0, w_p0_1);
RADIX2_F(d00_r, d00_i, d02_r, d02_i, w_p1_0, w_p1_1);
RADIX2_F(d01_r, d01_i, d03_r, d03_i, w_p1_2, w_p1_3);

//store the 8th 4-point data to memory
STM_02_DBL(&x[d07_ind], d00_r, d00_i);
STM_02_DBL(&x[d15_ind], d01_r, d01_i);
STM_02_DBL(&x[d23_ind], d02_r, d02_i);
STM_02_DBL(&x[d31_ind], d03_r, d03_i);
Appendix C

THE EXPERIMENTAL INFRASTRUCTURE

C.1 Directory Structure of the Code

- base 1D FFT implementation: ~/fft_code/fft_base/fft_1d
- base 2D FFT implementation: ~/fft_code/fft_base/fft_2d
- my 1D FFT implementation: ~/fft_code/fft_1d
- my 2D FFT implementation: ~/fft_code/fft_2d

C.2 README

- /*
  README for 1D FFT code
  Author: Liping Xue
  */

- This code is based on the base implementation from Long Chen, I use this code to evaluate different plans for 1D FFT

- How to use this code to evaluate a given plan?
  I will use an example of 2^11-point 1D FFT to show how to evaluate this given plan.

  1. represent plan in "c64_fft_z1d"
  Assume the plan for 2^11-point 1D FFT is "r16v1-first+r16v1+r8v1", the following code sequence shows how we represent this plan in "c64_fft_z1d".

```c
// bit reverse permute
c64_fft_bitrev_perm(x, dim, my_thread, threads);
```
tnt_barrier_wait(NULL);

// pass = 0,1,2,3
c64_fft_first_bfr16(x, dim, my_thread, threads, forward);
tnt_barrier_wait(NULL);
pass = 4;
// pass = 4,5,6,7
c64_fft_spm16(x, w, dim, pass, my_thread, threads, forward, spm);
tnt_barrier_wait(NULL);
pass = pass+4;
// pass = 8,9,10
c64_fft_bfr8v1(x, w, dim, pass, my_thread, threads, forward);
tnt_barrier_wait(NULL);
---------------------------------------------------------------

2. Recompile the code
   . How to Compile the code
      - use command "Make clean" to clean
      - use command "Make", the executable "c64_fft_main.bin"
        will be generated

3. Run the code
   . How to run the code
      - use the following command to run 2^11 1D FFT with 2 threads
        to evaluate this plan
      "cyclops-linux-elf-sim -spmd --bw -p2 --helper-threads=1
      c64_fft_main.bin 11"

• /*
   README for 2D FFT code
   Author: Liping Xue
   */

- This code is based on the base implementation from
  Long Chen, I use this code to evaluate different plans for 2D FFT

- How to use this code to evaluate a given plan?
I will use an example of \((2^8 \times 2^8)\) 2D FFT to show how to evaluate this given plan.

1. represent plan in "c64_fft_z2d"
Assume the plan for \((2^8 \times 2^8)\) 2D FFT is "r16v1-first+r8v1+r2v4", the following code sequence shows how we represent this plan in "c64_fft_z2d".

```c
//row FFT

//bit reversal on row FFT
C64_FFT_BITREV_PERM_REUSE_BALANCE(working_x, dim_row, dim_col, 0, 1, BY_ROW, my_thread, n_row, threads);
tnt_barrier_wait(NULL);

// pass = 0,1,2,3
C64_FFT_FIRST_BFR16_ROW(x, dim_row, dim_col, my_thread, threads, forward, BY_ROW);
tnt_barrier_wait(NULL);

pass = 4;

// pass = 4,5,6
C64_FFT_BFR8V1_ROW(x, w, dim_row, dim_col, pass, my_thread, threads);
tnt_barrier_wait(NULL);
C64_FFT_BFR2V4_ROW(x, w, dim_row, dim_col, pass, my_thread, threads);
tnt_barrier_wait(NULL);

//column FFT

//bit reversal on col FFT
C64_FFT_BITREV_PERM_REUSE_BALANCE(working_x, dim_row, dim_col, 0, 1, BY_COL, my_thread, n_col, threads);
tnt_barrier_wait(NULL);

// pass = 0,1,2,3
C64_FFT_FIRST_BFR16_COL(x, dim_row, dim_col, my_thread, threads, forward, BY_COL);
tnt_barrier_wait(NULL);
```

---

76
pass = 4;
//pass = 4,5,6
c64_fft_bfr8v1_col(x, w, dim_row, dim_col, pass, my_thread, threads);
tnt_barrier_wait(NULL);
c64_fft_bfr2v4_col(x, w, dim_row, dim_col, pass, my_thread, threads);
tnt_barrier_wait(NULL);

2. Recompile the code
   . How to Compile the code
      - use command "Make clean" to clean
      - use command "Make", the executable "c64_fft_main" will be generated

3. Run the code
   . How to run the code
      - use the following command to run $(2^8 \times 2^8)$ 2D FFT with 2 threads to evaluate this plan
        "cyclops-linux-elf-sim -spmd --bw -p2 --helper-threads=1 c64_fft_main 8"