Optimizing the Fast Fourier Transform on a Multi-core Architecture

Long Chen[1], Ziang Hu[1], Junmin Lin[2], Guang R. Gao[1]

[1]University of Delaware
Dept. of Electrical & Computer Engineering
Newark, DE 19716 USA
{lochen,hu,ggao}@capsl.udel.edu

[2]Tsinghua University
Dept. of Computer & Technology
Beijing, P.R.China 100086
linjunmin@tsinghua.org.cn

Abstract

The rapid revolution in microprocessor chip architecture due to multicore technology is presenting unprecedented challenges to the application developers as well as system software designers: how to best exploit the parallelism potential due to such multi-core architectures ? In this paper, we report an in-depth study on such challenges based on our experience of optimizing the Fast Fourier Transform (FFT) on the IBM Cyclops-64 chip architecture - a large-scale multi-core chip architecture consisting 160 thread units, associated memory banks and an interconnection network that connect them together in a shared memory organization.

We demonstrate how multi-core architectures like the C64 could be used to achieve a high performance implementation of FFT both in 1D and 2D cases. We analyze the optimization challenges and opportunities including problem decomposition, load balancing, work distribution, and data-reuse, together with the exploiting of the C64 architecture features such as the multi-level of memory hierarchy and large register files.

Furthermore, the experience learned during the hand-tuned optimization process have provided valuable guidance in our compiler optimization design and implementation.

The main contributions of this paper include: 1) our study demonstrates that successful optimization for C64-like large-scale multi-core architectures requires a careful analysis that can identify certain domain-specific features of a target application (e.g. FFT) and match them well with some key multi-core architecture features; 2) Our optimization, assisted with hand-tunned process, provided quantitative evidence on the importance of each optimization identified in 1) ; 3) Automatic optimization by our compiler, the design and implementation of which is guided by the feedbacks from 1) and 2), shows excellent results that are often comparable to the results derived from our time-consuming hand-tunned code.

1 Introduction

Due to the increasing power consumption, heat dissipation, and other issues, microprocessor chip architecture has been turning to multi-core rapidly in recent years. This is presenting unprecedented challenges to application developers as well as system software designers on how to exploit the potential parallelism provided by the multiple cores in a single chip.

While people believe that multi-core processors are going to become the mainstream in the future, there are few experiences of application development reported on those novel architectures.

In this paper, we report our detailed study in the implementation and optimization of the Fast Fourier Transform (FFT) on IBM Cyclops-64 (C64) multi-core architecture. FFT is one of the most important DSP algorithms and has been studied extensively on various architectures [14]. Digital signal processing (DSP) algorithms such as Fast Fourier Transform (FFT) are both computation-intensive and memory-intensive due to the large amount of data involved in the underlying applications. Moreover, real DSP applications usually require performing these algorithms on large data sets at real time. Thus a high performance computing engine is highly desired for this domain. On the other hand, most DSP algorithms have inherent parallelism [10, 15], more or less. Therefore, it seems that these algorithms can take the advantage of the emerging multi-core architectures to achieve better performance.

The C64 chip, the computing engine of an IBM petaflop

1-4244-0910-1/07/$20.00

supercomputer project, is used in our study. The C64 chip employs the multi-core design by integrating 160 thread units, 80 floating point units, 160 32KB SRAM banks, and a 96-port crossbar on a single chip. Other components include an A-switch that is used to form 3D mesh for a C64 system with multiple chips, 4 off-chip DRAM controllers, GigaBit Ethernet controller and other I/O devices. A C64 chip provides massively on-chip parallelism, massive on-chip memory bandwidth, large register file for each thread unit and multiple level of memory hierarchy. An important feature is that there is no data cache in the chip. C64 architecture represents one kind of multi-core architectures that is intended for high performance computing. Our experience of FFT study on C64 may benefit application/system software developers on other multicore platforms.

During our study of 1D and 2D FFT on C64, we found out that domain specific knowledge is very important. To achieve better performance, we carefully analyzed the FFT algorithm features, identified a set of important issues, set up the experiments based on the analysis, found out the optimal parameters that matches the C64 architecture features. Those issues include problem decomposition, load balancing, work distribution, data-reuse, register tiling, and memory hierarchy aware instruction scheduling. Some of the above optimizations had been done manually before the C64 compiler can automatically generate more efficient code. Our study provided valuable guidance to the compiler designers on what should be done in C64 compiler optimizations. For instance, the C64 compiler was enhanced by adding the memory segment aware instruction scheduling, such that the automatically generated code for FFT by the compiler is very close to the tediously hand-tunned code eventually.

The main contributions of this paper include: 1) our study demonstrates that successful optimization for C64-like large-scale multi-core architectures requires a careful analysis that can identify certain domain-specific features of a target application (e.g. FFT) and match them well with some key multi-core architecture features; 2) Our optimization, assisted with hand-tunned process, provided quantitative evidence on the importance of each optimization identified in 1) ; 3) Automatic optimization by our compiler, the design and implementation of which is guided by the feedbacks from 1) and 2), shows excellent results that are often comparable to the results derived from our time-consuming hand-tunned code.

The rest of this paper is organized as follows. In Section 2, a short introduction of the FFT algorithm is given. The problem characteristics are analyzed and the basic approach to the problem solution is also discussed here. In Section 3, we present the C64 chip architecture and its major features. In Section 4, the optimizations and analysis of the FFT on the C64 architecture are conducted. Section 5 is about the related works. And Section 6 concludes the paper with future work discussion.

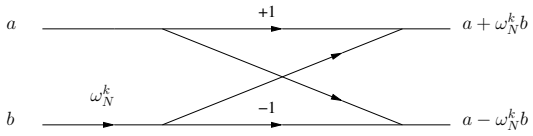

Figure 1. Cooley-Tukey butterfly operation

2 Fast Fourier Transform

The FFT is a fast algorithm for computing the Discrete Fourier Transform (DFT). In the literature, the FFT has been extensively studied and implemented as an important frequency analysis tool in many areas such as image processing, signal processing, and other domains.

There are many variants of the FFT algorithms. In this paper, we focus on the most common FFT algorithm, the radix-2 Cooley-Tukey algorithm [6], which employs the *divide and conquer* approach. Consider the computation of the $N = 2^t$ points DFT, $x(n)$. The algorithm divides this N-point data sequence into two $N/2$-point data sequences $f_1(n)$ and $f_2(n)$, corresponding to the even-indexed and odd-indexed points of $x(n)$, respectively. Then the N-point DFT can be computed as,

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

where ω_N^k are *twiddle factors*, $F_1(k)$ and $F_2(k)$ are the $N/2$-point DFTs of $f_1(n)$ and $f_2(n)$, respectively. The subproblems $F_1(k)$ and $F_2(k)$ are recursively solved to obtain the final solution of the original problem. Consequently, the FFT gives an $\Theta(N \log_2 N)$ algorithm for computing DFT. The computation above is usually referred to as the Cooley-Tukey butterfly operation, which is shown in Figure 1. In general, FFT on multidimensional data set can be realized by performing 1D FFT alternately on each dimension of the data, interleaved with data transpose steps.

Generally speaking, the recursive FFT algorithm introduces non-negligible recursion overhead thus is not favored. Another approach is to employ the iterative implementation. The iterative algorithm subdivides the resulting subproblems iteratively until the problem size becomes one. In order to achieve such an implementation, the input data have to be reordered before the butterfly computations are performed. This is called *bit-reversal permutation*. In the Cooley-Tukey algorithm, this permutation is performed before the butterfly computations.

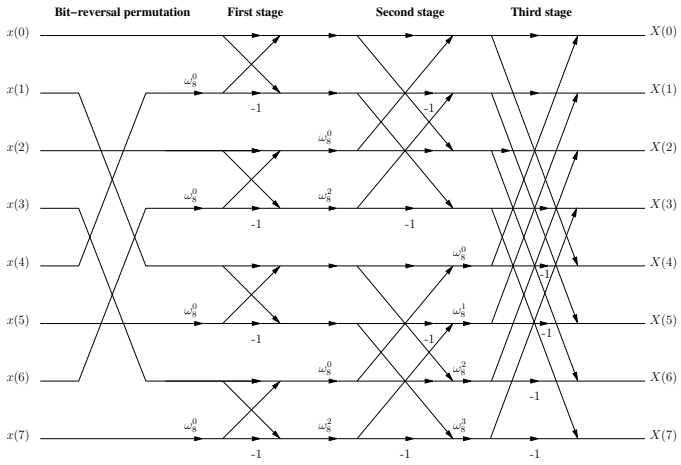

Figure 2. 8-point Cooley-Tukey example

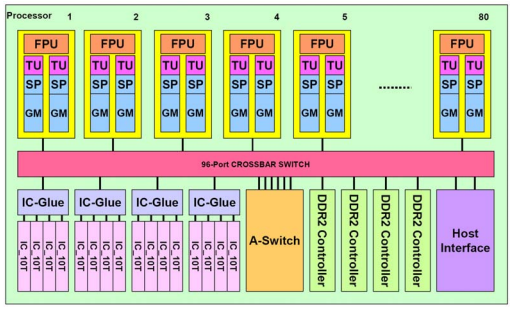

Figure 3. Cyclops-64 chip architecture

Figure 2 shows an example the iterative FFT decomposition of 8 points using the Cooley-Tukey algorithm. Before the butterfly computation, the bit-reversal permutation is performed on the input data. Then the computation is decomposed through 3 stages. We use iterative approach in this paper.

3 Cyclops-64 Architecture

The C64 chip, shown in figure 3, is the core computation engine of the C64 supercomputer system, which consists of thousands of such chips connected through a 3D-mesh network. One C64 chip features massive parallelism with 80 64-bit processors, each consisting 1 floating point unit (FPU) and 2 thread units (TUs). Each thread unit is a single-issue, in-order RISC processor operating at a moderate clock rate (500MHz). It has 64 64-bit registers and 32 KB SRAM. Other on-chip components include 16 shared instruction caches (ICs), 4 off-chip DRAM controllers, A-Switch, and etc. All on-chip resources are connected to an on-chip pipelined crossbar network with a large number of ports (96×96), which sustains a 4GB/s bandwidth per port, thus 384GB/s in total.

The C64 architecture has a segmented memory space, including the scratch-pad (SP) memory, on-chip global interleaved memory (GM), and off-chip DRAM. It is interesting to note that C64 does not have data cache. Instead, the on-chip SRAM banks are partitioned into the SP memory and GM. While the GM are shared among all threads on the chip, the SP memory is regarded as the fast local memory of the corresponding thread unit (2 cycles for load, 1 cycle for store).

All the thread units within a chip are connected by a 16-bit signal bus, which provides a means to efficiently implement barriers. Furthermore, the C64 instruction set architecture (ISA) features a large number of atomic in-memory instructions. All these greatly facilitate the thread-level parallelism with fast inter-thread synchronizations.

4 Optimizations and Discussions

In this section, we discuss our experiences on the implementation, analysis and optimizations of the FFT on C64 architecture. In the experiments, we consider the data sizes of 2^{16} and 256×256 for 1D FFT and 2D FFT, respectively. In both cases, the input data are double-precision complex numbers, and can fit into on-chip GM. The twiddle factors are pre-computed and stored in on-chip GM as well. All the experiments were conducted on the FAST simulator[7], which is a functionally-accurate simulator that, among other features, models the memory hierarchy of C64 architecture, including the latencies and bandwidth of each memory segment.

4.1 1D FFT

We start with a base parallel implementation of the 1D FFT. Then, by analyzing the problem thoroughly, we apply a sequence of optimization techniques to improve the performance of the parallel implementation.

4.1.1 Base Parallel Implementation

Before we go into details about the implementation, let us first introduce an important definition: work unit. A *work unit* is an arbitrarily defined piece of the work that is the smallest unit of concurrency that the parallel program can exploit. The size of a work unit may vary in different implementations. In this base parallel implementation, we consider a butterfly operation to be a work unit, which includes 1) read 2-point data and the twiddle factor from GM and, 2) perform a bufferfly operations upon them, then, 3) write the

2-point results back to GM. We call this a 2-point work unit because it contains 2 points that can be computed independently from other data. This fine-grained approach matches the natural granularity of the FFT in the sequential program structure, which is the smallest unit of concurrency that the FFT exposes. To achieve a balanced workload among all threads, the work units are assigned to threads in a round-robin way, during each stage of the FFT computation. Barriers are used to synchronize threads before the next stage starts. This parallel implementation has a performance of 6.54Gflops.

4.1.2 Optimal Work Unit

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

Let us look at an alternative approach. By using 2-point work units, a 4-point FFT computation can be completed in two stages and requires 2 such work units at each stage, 4 in total. In other words, this computation requires 24 load operations, 40 floating point operations, and 16 store operations. Instead of containing of 2 points, if one work unit has 4-point data that can be computed independently from other data, the thread can read all data into registers, perform required computation, and write back the results. In this case, this work unit consists of 16 load operations, 40 floating point operations, and 8 store operations. Following the convention, we call this 4-point work unit. Similar to the base implementation, threads need to be synchronized after all of them finish their 4-point work units, which are 2-stage FFT computations. Compared with the previous implementation, this method eliminates half number of the barriers and reduces the number of memory operations by 40%, and increases the percentage of floating point operations to the total number of instructions from 50% to 62.5%[1]. This is definitely an encouraging sign to achieve better performance. Let us extend this idea more ambitiously. Assuming we have a machine with an unlimited number of registers, if one work unit has 2^{16}-point data, then only one barrier is needed and the percentage of floating point operations to the total number of instructions would increase to 80%! In general, given a machine with a unlimited number of registers, using a work unit of N-point data can get rid of $(\lg_2 N - 1)$ barriers. Moreover, the percentage of floating point operations to the total number of instructions is $\frac{5N \lg_2 N}{6N \lg_2 N + 4N}$. It is clear that the more data one work unit has, the more benefits we will gain.

However, in practical, no machine has unlimited registers, neither does C64. Further, a huge work unit may limit the concurrency exposed by the program. So we have to decide an appropriate size of the work unit, which should expose enough parallelism and still can fully utilize the register file without serious register spilling. Let us examine the above example again. For 4-point work unit, it needs 8 registers for input data, 4 registers for the corresponding indexes, another 8 registers for the twiddle factors. Thus the total number of registers needed would be 20. While completing this work unit will not cause register spilling, it does underutilized the register file: about half registers are not used during the entire computation. Considering the case of 8-point work unit, it requires 16 registers for input data, 8 registers for the corresponding indexes, another 24 registers for the twiddle factors. The total number of registers to be used would be 50. In theory, the execution of this work unit will use most of the registers and it will not generate register spilling. If we go a little bit further with the 16-point work unit, the total number of registers needed during the computation increases to 112, which imposes much greater pressure on the register file and will certainly introduce serious register spilling and thus typically will slow down the computation. Therefore, based on our analysis, the 8-point work unit could be the best choice for C64. Note that 8-point work unit implies a 3-stage FFT computation. Given a FFT computation with n-point data, when $\lg_2 n$ cannot be divided exactly by 3, the last $(\lg_2 n - \frac{\lg_2 n}{3})$ stage(s) can be computed with 4-point work units or 2-point work units.

Our analysis and conclusion have been confirmed by the experimental results with different sizes of work unit, which are shown in figure 4. Obviously 8-point work unit outperforms other sizes of work unit. After applying this 8-point work unit, we reach a performance of 13.17Gflops, which is 101.5% improvement over the base parallel implementation.

4.1.3 Special Handling of the First Stages

As shown in figure 2, every bufferfly operation performs on consecutive data during the first stage. For example, the top left butterfly operation acts upon $x(0)$ and $x(4)$, which

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

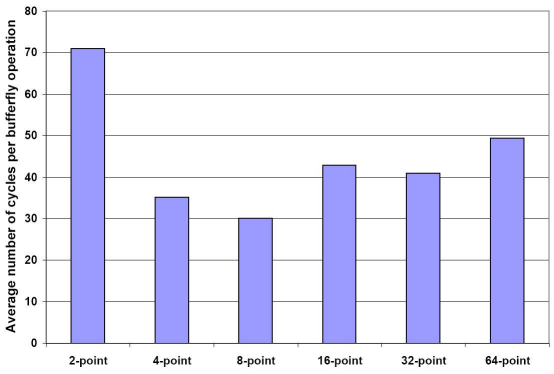

Figure 4. Number of cycles per bufferfly operation versus the the size of work unit

are contiguous in the memory after the bit-reversal permutation. It holds true that all points within the same work unit are consecutive in the memory before the first stage, for any valid size of work unit. This implies that less registers are required for the first $\lg_2 M$ stages, as when M-point work unit is used, only the starting pointer and the size of the work unit are needed to access this work unit, instead of computing the indexes for all the points and keeping them in registers.

Inspired by this observation, we try to search the appropriate M, the size of work unit for the first $\lg_2 M$ stages. Since it is clear that $M \geq 8$, let us consider 16-point work unit again. We need 32 registers for the input data and 1 register for the starting address of this work unit, another 64 registers for the 32 twiddle factors. Thus the total number of registers would be 97, which still exceeds the maximum available registers in C64 architecture. It seems that 8-point is the maximum size of work unit that we can use during the entire FFT. However, let us look at figure 2 more carefully. In this figure, all bufferfly operations performed during the first stages are using the same twiddle factor ω_8^0. In the second stage, only 2 distinct twiddle factors are used, i.e., ω_8^0 and ω_8^2. In general, in the i-th stage of a complete FFT computation, 2^{i-1} distinct twiddle factors are used, and they include all twiddle factors used in the preceding stages. In other words, during the execution of the first $\lg M$ stages, there are fewer twiddle factors being used. By knowing this fact, we re-consider the possibility of using 16-point work unit. Instead of 64, we only need 16 registers to keep 8 distinct twiddle factors used in the fist 4 stages. Thus the total number of registers required is 49, which can fit into the C64 register file. Further, we define these 8 twiddle factors as *macros* in the program. This approach reduces the number of memory operations, while the inaccuracy introduced is well under control[2]. After applying these approaches for the first 4 stages, we achieve an improvement of 28.4% over the earlier implantation, while the absolute performance reaches 16.92Gflops.

4.1.4 Eliminating Unnecessary Memory Operations

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

4.1.5 Loop Unrolling

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

4.1.6 Register Renaming and Instruction Scheduling (Manually)

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

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

Table 1. 2^{16} 1D FFT Incremental Optimizations

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

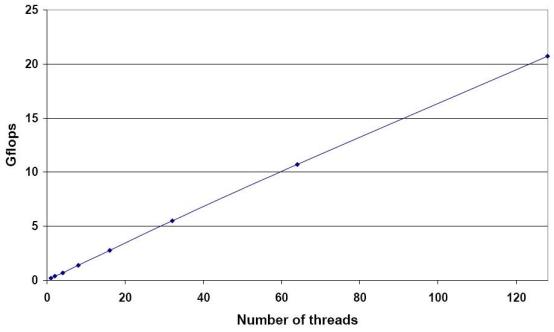

Figure 5. Speedup of the optimized 1D FFT implementation

4.1.7 Memory Hierarchy Aware Compilation

While the above manual optimizations can achieve a relatively high performance, the entire process is tedious and error-prone. The different delays of memory instructions when accessing different memory segments have to be carefully investigated and manipulated. On the other hand, this work would be an ideal job for a smart compiler that could identify the segments where variables reside, and apply the corresponding latencies when scheduling the instructions. Inspired by this idea, we tailored the compiler such that it accounts for the different latencies when accessing variables specified with segment pragmas when applying instruction scheduling. By employing this memory hierarchy aware compiler with the code from 4.1.5, we achieve a 8.8% improvement, which corresponds to a performance of 19.84Gflops. While the absolute performance is a little bit lower than the manually optimized code in 4.1.6 (it is still comparable to the latter), this optimization dramatically reduces the effort to achieve a high performance implementation on architectures with deep memory hierarchy like C64.

So far, we finish the optimizing 1D FFT implementation. We list all the techniques applied and the corresponding results in Table 1. Figure 5 shows the speedup of this optimized implementation. The effect of the memory hierarchy aware compiler is not shown in either Table 1 or Figure 5. From these plots, we observe that the performance of this implementation scales nearly linearly up to 128 threads.

4.2 2D FFT

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

4.2.1 Base Parallel Implementation

In the base implementation, we simply employ one row/column FFT as a work unit. All row FFTs are independent of each other, so they can be computed in parallel. So do all column FFTs. After completing all row FFTs, a barrier is used to synchronize all threads before they perform the column FFTs. Work units are distributed to threads in the round-robin way. By utilizing the optimized 1D FFT implementation presented in the previous section, this parallel implementation achieves a performance of 15.11Gflops.

4.2.2 Load Balancing

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

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

4.2.3 Work Distribution and Data Reuse

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

4.2.4 Memory Hierarchy Aware Compilation

Again, we apply the memory hierarchy aware compiler to the 2D FFT implementation, which introduces another 3.25% improvement over the previous compilation, thus the overall performance raises to 20.00Gflops. Finally, similar to the 1D FFT, the optimized 2D FFT implementation also scales nearly linearly up to 128 threads, as shown Figure 6.

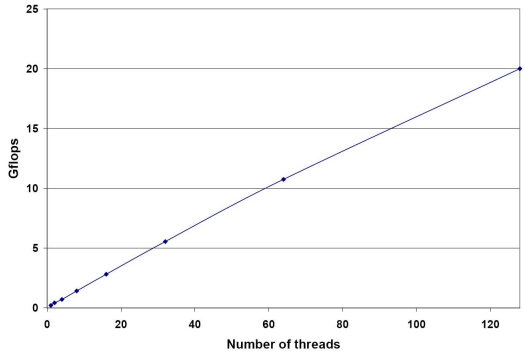

Figure 6. Speedup of the optimized 2D FFT implementation

5 Related Work

The FFT problem has been extensively studied on various machines. A large number of literature addresses the distributed memory FFT implementations on the hypercube architecture [9, 13, 17] by taking the advantage of the small communication delay between processors that are physically close in the network. Other parallel FFT implementations have investigated on arrays[12] and mesh architectures[18]. An instructive shared-memory FFT for Alliant FX/8 is presented in[19]. Additional performance studies on shared-memory FFT are discussed in [3, 16]. Moreover, by using the Kronecker notation, the work in[11] shows how to design parallel DFT algorithms with various architecture constraints. The significance of considering memory hierarchy to an effective FFT implementation has been pursued in[4]. The work in [5] shows how to use local memory to compute the FFT efficiently on CRAY-2. The issue of data re-use is also discussed in [1, 2]. Further, an excellent review of various sequential and parallel DFT algorithms proposed in the literature until 1991 appeared in[14]. Two dataflow-based multithreaded FFTs[21] are presented to exploit the features of EARTH[20], a fine-grained dataflow architecture. FFTW [8] features a dynamic programming algorithm to determine the best execution *plan*. It supports multithreaded programming interfaces, and is portable and adaptable on various architectures.

6 Conclusion and Future Work

In this paper, we presented the implementation and optimizations of the FFT on the C64 multi-core architecture, together with extensive analysis. The results demonstrates that multi-core architectures like C64 can be used to achieve excellent performance results with respect to both speedup

and absolute performance for DSP problems like FFT. For instance, the best result of the FFT obtained on a 3.60GHz Intel Xeon Pentium 4 processor is 5.5Gflops[8], which is only around one quarter of using one C64 chip.

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

On the other hand, multi-core system software, especially the compiler, faces more challenges. In the study we shows that memory hierarchy aware instruction scheduling may dramatically improve the performance while reduces the burden from the programmers.

One of the architecture features of C64 has not been fully explored is the fast scratchpad memory (SP) for each thread unit. SP may be used as larger register file. More FFT points may be stored in SP such that each FFT work unit may contain more points. Another issue is to study larger FFT problem sizes when data cannot be fully stored in on-chip memories.

Acknowledgments

We would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of the IBM Cyclops-64 architecture; ET International, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. Special thanks to Michael Merrill for his initial FFT implementation. We would also like to acknowledge other members of the CAPSL group at University of Delaware, in particular Weirong Zhu, Yuan Zhang and Shuxin Yang.

References

[1] R. Agarwal and J. Cooley. Vectorized mixed radix discrete fourier transform algorithms. In *Proceedings of the IEEE*, volume 75, pages 1283– 1292, 1987.

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

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

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

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

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

[7] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS2005), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005)*, Madison, Wisconsin, June 2005.

[8] M. Frigo and S. Johnson. FFTW.

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

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

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

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

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

[14] C. V. Loan. *Computational framework for the fast Fourier transform*. SIAM, Philadelphia, 1992.

[15] H. Nguyen and L. K. John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *International Conference on Supercomputing*, pages 11–20, 1999.

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

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

[18] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *International Parallel Processing Symposium*, pages 92–101, 1991.

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

[20] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, May 1999.

[21] P. Thulasiraman, K. B. Theobald, A. A. Khokhar, and G. R. Gao. Multithreaded algorithms for the fast fourier transform. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 176–185, 2000.