Performance Tuning of the Fast Fourier Transform on a Multi-core Architecture

Liping Xue Long Chen Ziang Hu Guang R. Gao,

CAPSL Technical Memo 81
Feburary 8th, 2008

Email: {xue,lochen,hu,ggao}@capsl.udel.edu

Abstract

We are now entering the multi-core era, many multi-core chips are designed and manufactured by various vendors, such as Intel, AMD and Sun etc. IBM Cyclops-64(C64) is a multi-core architecture that provides massive on-chip parallelism, massive on-chip bandwidth, and multiple level memory hierarchy. This type of multi-core architecture presents big challenges to application developers and system software designers on how to exploit the thread level parallelism(TLP) provided by the multi-core chips.

While a lot of researchers believe that multi-core architecture will become the mainstream in the future, there are only a few studies about the application development on those advanced architectures have been reported .

The emerging multi-core architectures not only 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 the IBM Cyclops-64 (C64) architecture, a novel multi-core architecture consisting of 160 threads, an explicit memory hierarchy, and an on-chip interconnection network.

C64 does not have data cache and thus a simple porting of a cache-oblivious algorithm may not be able to take advantage of its architecture features. In addition, current implementations of the cache-oblivious method are entirely based on cache-memory hierarchy that does not lend itself to the construction of an accurate performance model for C64 like platforms, which involve explicit data movement in the explicit memory hierarchy. Therefore, to make a cache-oblivious FFT working efficiently on such architecture is probably non-trivial.

The work presented in this paper takes a different path. We first present an iterative search approach to find the optimal sequence of kernel functions to compute the FFT. This approach also constructs an accurate/deterministic performance model analytically. Then, the model is used to calculate the performance of different FFT computation sequences iteratively. Such performance numbers will be productively used by our search based optimization procedure.

We then propose a new technique for optimizing the scratchpad memory (SPM) space utilization. This technique fully exploits the opportunity provided by the explicitly-addressable on-chip memory hierarchy. It can judiciously explore life-range splitting methods and achieve a significant performance gain, which is evidenced by our experiments. The experimental results have demonstrated up to 25.5% performance improvement over a previous efficient FFT implementation on C64.

1 Introduction

Microprocessor chip architecture has been turning to the multi-core era, many multi-core architectures have been proposed by various vendors, such as Intel, AMD, Sun, etc. This type of architecture presents great challenges to application developers and system software designers on how to exploit the thread level parallelism (TLP), and other architectural features provided by these novel chips. However, in the literature, there are only a few studies about the application development on those advanced architectures.

In this paper, we report our study on tuning of the Fast Fourier Transform (FFT) on the IBM Cyclops-64 (C64) multi-core architecture. The C64 chip, the experiment platform used in our study, features massive on-chip parallelism, and massive on-chip bandwidth. One interesting feature of this architecture is that it does not have data cache. It brings a great challenge to the software development on C64, together with the existence of the explicit memory hierarchy.

Long et al. [1] reported their study on implementing and optimizing the FFT on C64. They first defined *work unit* as “an arbitrarily defined piece of the work that is the smallest unit of concurrency that the parallel program can exploit”. Then, based on experimental results, they found the optimal work units for computing the FFT on the C64 architecture. Finally, kernel functions were developed and optimized for these optimal work units, and a sequence of kernel functions was statically specified for all problem sizes. However, as we show in this paper, for some problem sizes, these “optimal” work units and the sequence used in [1] are not optimal, in terms of the performance.

Since C64 does not have data cache, a simple porting of the cache-oblivious algorithm, i.e., FFTW [2], may not be able to take advantage of its architecture features. In addition, current implementations of the cache-oblivious method are 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 involving data movement through a C64-like explicit memory hierarchy. Therefore, to make a cache-oblivious FFT implementation working efficiently on this architecture is probably non-trivial.

In this paper, we first present an iterative search approach to find the optimal sequence of FFT kernel functions for different problem sizes. Furthermore, our approach analytically constructs an accurate and deterministic performance model, which is used to compute the performances of a sequence of kernel functions iteratively. Such performance numbers will be productively used by our search based optimization procedure. We then propose a new technique to take advantage of the explicit memory hierarchy. This technique exploits the opportunity provided by the C64 explicitly-addressable on-chip memory hierarchy. By using this technique, we could further refine the search approach and achieve a significant performance gain. We verify all our proposed methods via simulation experiments. The experimental results have demonstrated up to 25.5% performance improvement over [1].

The rest of this paper is organized as follows. In Section 2, we give a brief introduction of FFT. In Section 3, we present the C64 architecture and its major features. In Section 4, we present our search scheme to find the optimal sequence of kernel functions. Then we present a technique to utilize the fast memory segment to further optimize the FFT. In Section 5, we present some brief comments on recent literature that is closely related to the problem addressed in this paper, and in Section 6, we conclude the paper with some open-ended issues to be addressed.

2 Fast Fourier Transform

The FFT algorithm is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFT has been widely used in many areas, including digital signal processing, image processing and other domains.

While there are many variants of FFT algorithms, the most common one is the Cooley-Tukey algorithm [3]. This algorithm recursively breaks down a DFT of size N into two smaller DFTs of size $N1$ and $N2$, respectively, where $N = N1 \times N2$. The most well-known use of Cooley-Tukey algorithm is the radix-2 Cooley-Tukey algorithm, i.e., $N1 = N2 = N/2$. Let us consider the $N = 2^t$ point DFT, $x(n)$, the radix-2 algorithm divides the N-point data sequences into two $N/2$-point data sequences, $f1(n)$ and $f2(n)$, corresponding to the even-indexed and odd-indexed points of $x(n)$, respectively. Then the N-point DFT $X(k)$ 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*, $F1(k)$ and $F2(k)$ are the $N/2$-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 $\Theta(Nlog_2N)$. The above computation is usually referred to as the *Cooley-Tukey butterfly operation*, which is shown in Figure 1. Although the basic idea of the Cooley-

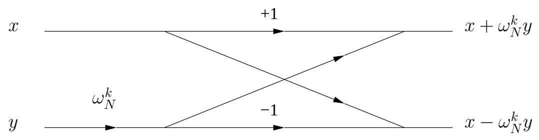

Figure 1: Radix-2 Cooley-Tukey Butterfly Operation

Tukey algorithm is recursive, many practical implementations use an iterative algorithm to avoid the recursion overhead. For the iterative algorithm, the input data need to be reordered before the butterfly computation, which is *bit-reversal permutation*. Our implementation in this paper is an iterative one as well.

3 Cyclops-64 Architecture

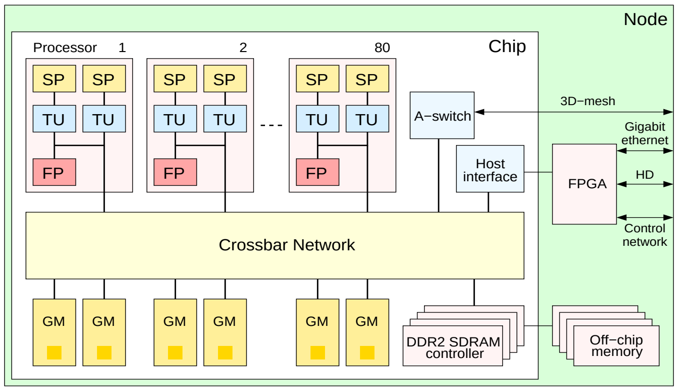

Figure 2: C64 Chip Architecture

The C64 chip, shown in Figure 2, is designed to be the computation engine of a Petaflop supercomputer system. Such system consists of thousands of C64 chips that are connected through a 3-D mesh network. The C64 chip favors massive parallelism by integrating 80 64-bit processors, 160 embedded SRAM banks and an interconnection network 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 RISC processor running at 500MHz clock rate. The interconnection network is an on-chip pipelined crossbar network with huge number of ports (96×96), which can provide 4GB/s bandwidth per port.

The C64 memory hierarchy consists of three level memories, the scratchpad (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: scratchpad memory (SPM) and GM. Each thread unit has its own SPM, which is the fast local memory of the corresponding thread unit. The GM is shared by all thread units on the chip with uniform access latency.

4 Optimizations and Discussions

In this section, we first review the previous FFT implementation on C64 [1]. Then we present our experiences of tuning FFT on C64 architecture. All experiments are conducted on the FAST simulator [4], which is a functionally-accurate simulator that is employed for software development and testing before the real chip becomes available. It models the memory hierarchy of C64 architecture, including the latencies and bandwidth for each memory segment. The input data are double-precision complex numbers and can fit into the on-chip GM. The twiddle factors are precomputed and stored in GM as well.

4.1 Previous Implementation

In a previous work on optimizing the FFT on C64 [1], the authors defined the notion of work unit, and they found the 8-point work unit was the optimal work unit for computing the FFT on C64, i.e., the atomic execution unit of each thread has 8 points, which implies a 3-stage butterfly computation. For each work unit, they implemented and optimized a *kernel function* that carries out the above butterfly computation. Table 1 summarizes the kernel functions implemented in [1] and their descriptions. One

Kernel Index	Kernel Name	Description
1	r2v1	2-point work unit, 1 stage computation, working on 1 group data
2	r2v2	2-point work unit, 1 stage computation, working on 2 groups data
3	r2v4	2-point work unit, 1 stage computation, working on 4 groups data
4	r2v8	2-point work unit, 1 stage computation, working on 8 groups data
5	r4v1	4-point work unit, 2 stages computation, working on 1 group data
6	r4v2	4-point work unit, 2 stages computation, working on 2 group data
7	r8v1	8-point work unit, 3 stages computation, working on 1 group data
9	r16v1-first	16-point work unit, 4 stages computation, working on 1 group data

Table 1: Kernel Functions and Their Descriptions

may notice that, in Table 1, for a given k-point work unit, there may be more than one kernel function.

For example, for the 2-point work unit, it was implemented with four different kernel functions, namely, *r2v1*, *r2v2*, *r2v4*, and *r2v8*, respectively. The basic one, *r2v1*, is the implementation of the Cooley-Tukey butterfly operation. *r2v2*, *r2v4* and *r2v8* are considered as vector versions of *r2v1*, which work on 2, 4 and 8 groups of 2-point data, respectively. Similarly, for the 4-point work unit, there are two different kernel functions, namely, *r4v1* and *r4v2*, respectively. *r16v1-first* is a specially optimized kernel function to compute the first 4 butterfly stages of the FFT, working on 16 points at one time.

Having the above kernel functions, a scheme for computing the FFT was proposed. Given n-point data ($lg_2 n$ stages), for the first 4 stages, *r16v1-first* is applied. For the remaining $(lg_2 n - 4)$ stages, $r8v1$ is repeated applied. If there is one or two last stage(s) left, *r2v8* or *r4v2* will be used, respectively, to reduce the number of branch instructions, All work units are assigned among all threads in a round-robin way to achieve load balance. Overall, the sequence of kernel functions used was statically fixed.

4.2 The Effect of Number of Threads

[1] showed that linear speedup can be obtained for 2^{16} 1D FFT when the number of running threads is increasing. Can we always get more speedup if we use more threads? We run the 1D FFT code mentioned in [1] with different input size and different number of threads. The result in Figure 3 shows the scalability for n-point FFT on C64. 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 be achieved. The performance starts to degrade when running with more than p threads due to the increase of synchronization overhead. We also find out that, this optimal number of threads p is proportional to the size of input data. The larger size of the input data, the more threads are needed. This result also

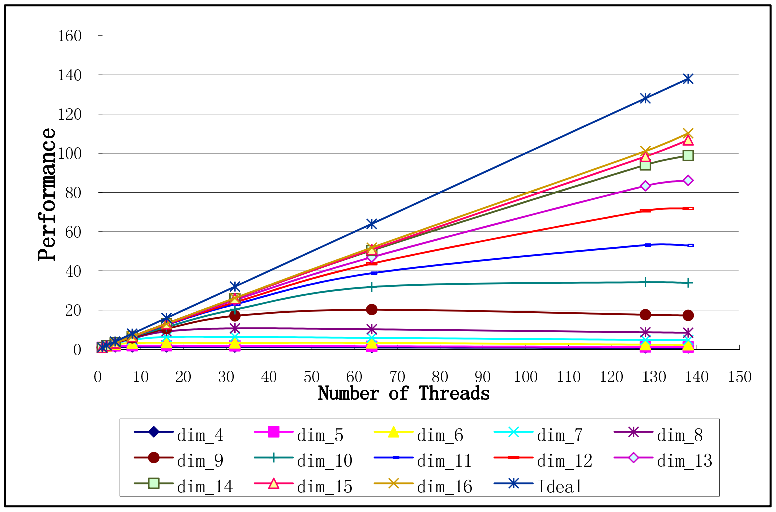

Figure 3: Scalability of 1D FFT on C64

brings a new challenge for tuning applications on multi-core architectures. Compared to a single core architecture, the tuning procedure needs to determine the optimal number of threads.

4.3 Search-based Scheme

The computing scheme proposed in [1] uses a fixed sequence for any problem size. It is very simple. But, is it really optimal for the C64 architecture? In order to answer this question, we design and implement an iterative search-based scheme.

In the rest of this paper, we use the same definition of *work unit*. Additionally, we define *plan* as a sequence of kernel functions used to perform a given n-point 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 concurrent threads.

This idea of this search scheme can be described as follows. Let $F(i)$ denote the minimum number of 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 with the minimum number of cycles $F(m)$, which can be calculated according to the following equation,

$$F(m) = \min\{F(m - S(k)) + T(k)\}, k \in Z \tag{1}$$

where Z is the set of all kernel functions in Table 1, $T(k)$ denotes the total number of cycles of running the kernel function k, plus a barrier on the given n-point data, $S(k)$ denotes the number of computation stages of the kernel function k. Dynamic programming is used to calculate $F(m)$. Suppose p is the optimal plan for a m stages computation, and the last kernel function used in p is k. Since kernel function k can finish $S(k)$ stages computation, the minimum number of cycles to finish the previous $(m - S(k))$ stages computation is $F(m - S(k))$. If p is the optimal plan for m stages computation, then the subplan p' of p should be the optimal plan to compute previous $(m - S(k))$ stages.

The search-based approach is presented as a two step algorithm,

Step 1: given n-point data, run each kernel function k in Table 1 plus a barrier with an empirically optimal number of threads to get $T(k)$. Compute $S(k)$ for each kernel function.

Step 2: run the search algorithm to search the plan. A data structure *BestPlan* is used to store the optimal plan. BestPlan$[i]$ denotes the optimal plan for the first i stages computation. $curr$ stores the current local minimum number of cycles to accomplish 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 2 shows the results generated by using the above algorithm. The first column is the input size. An input with dimension i has 2^i point data. The second column shows the optimal number of threads used for computing the FFT of this specific input size, which is determined empirically. In our search algorithm, we first fix the number of threads for the specific input size, and then search different plans running with optimal p threads. The third column shows the plan described in [1], which is called the *base plan*. The last column is *Plan I,* which is obtained by using this search algorithm, where the differences between the base plan and this plan are in bold. Comparing Plan I and the base plan shown in Table 2, one observation is that Plan I favors *r2v4* over *r2v8*. Intuitively, we consider that *r2v4* is better than *r2v8*. But why *r2v4* is better than *r2v8*? In order to answer this question, we introduce a value, avc_pb which means average cycles per butterfly, to evaluate the efficiency of kernel functions.

Algorithm 1 $Search(m, Z)$

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

Dim	# of Threads	Base Plan	Plan I
4	8	r16v1-first	r16v1-first
5	8	r16v1-first+r2v8	**r4v1+r8v1**
6	16	r16v1-first+r4v2	**r8v1+r8v1**
7	16	r16v1-first+r8v1	r16v1-first+r8v1
8	32	r16v1-first+r8v1+r2v8	r16v1-first+**r4v2+r4v2**
9	64	r16v1-first+r8v1+r4v2	r16v1-first+r8v1+r4v2
10	128	r16v1-first+r8v1+r8v1	r16v1-first+r8v1+r8v1
11	128	r16v1-first+r8v1+r8v1+r2v8	r16v1-first+r8v1+r8v1+**r2v4**
12	138	r16v1-first+r8v1+r8v1+r4v2	r16v1-first+r8v1+r8v1+r4v2
13	138	r16v1-first+r8v1+r8v1+r8v1	r16v1-first+r8v1+r8v1+r8v1
14	138	r16v1-first+r8v1+r8v1+r8v1+r2v8	r16v1-first+r8v1+r8v1+r8v1+**r2v4**
15	138	r16v1-first+r8v1+r8v1+r8v1+r4v2	r16v1-first+r8v1+r8v1+r8v1+r4v2
16	138	r16v1-first+r8v1+r8v1+r8v1+r8v1	r16v1-first+r8v1+r8v1+r8v1+r8v1

Table 2: Plan Comparison

The smaller the avc_pb, the higher the efficiency of the kernel function. Figure 4 shows avc_pb of different kernel functions. The kernel function "r16-first" has the lowest avc_pb value, which implies highest efficiency. Besides that, "r8v1" has the second lowest avc_pb value. Therefore, for the input n-point data FFT, if $lg_2 n - 4$ can be divided exactly by 3, the scheme proposed in [1] can achieve the best performance. However, if $lg_2 n - 4$ cannot be divided exactly by 3, the scheme proposed in [1] used a kernel function for 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, that scheme choosed to use kernel function *r2v8* and *r4v2*. Although the number of branch instructions can be reduced by using these two kernel functions mentioned above, that scheme cannot achieve the best performance for some input sizes.

Figure 5 shows the performance comparison between the base plan and Plan I, both running with the number of threads specified in Table 2. From the figure, we observe that the performance of Plan I is always not worse than that of the base plan. For small input sizes, Plan I has distinct speedup up to 25.5% over the base plan. The significant performance difference between these two plans are due to the different kernel functions chosen by these two plans. For example, although *r16v1-first* is very

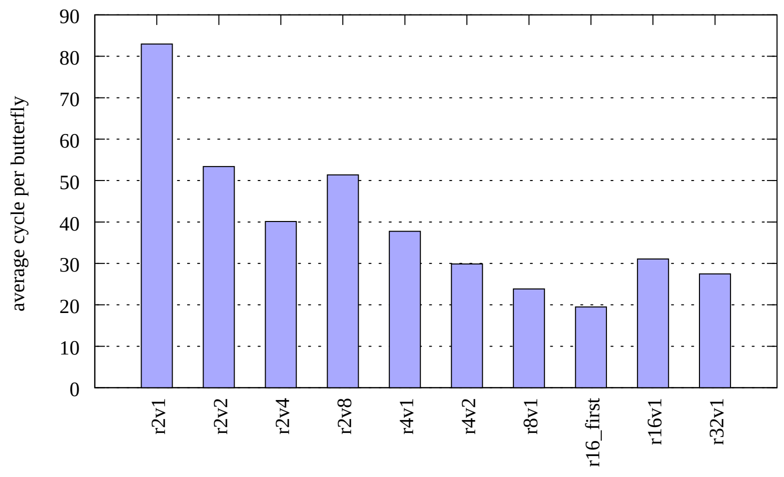

Figure 4: Kernel Evaluation

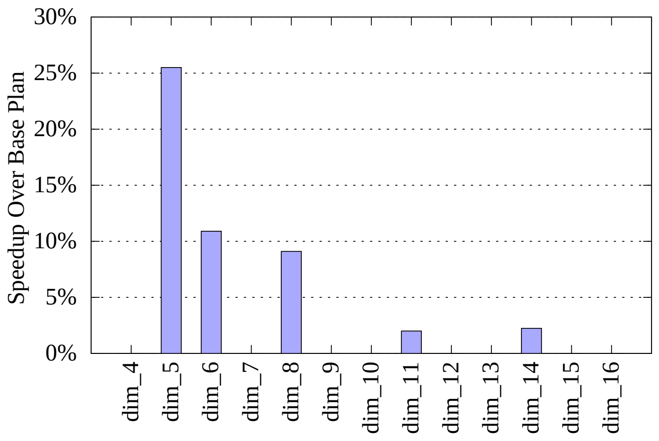

Figure 5: Search the Best Plans

efficient and is chosen as the optimal plan (in Plan 1) for the input size 2^4, it does not necessarily mean that it should be a natural choice for other input sizes. In fact, we evaluate the efficiency of different kernel functions used in the experiment and find that the order of efficiency of those kernel functions is *r16v1-first* $>$ *r8v1* $>$ *r4v1* $>$ *r2v8*. Therefore, it is possible that we can obtain a better performance by searching different combinations of kernel functions, for example, *r4v1+r8v1* versus *r16v1+r2v8* for the input size 2^5. This performance difference also shows that the fixed plan proposed in [1] is not optimal (for some problem sizes) and illustrates the importance of the search approach. On the other hand, for the large input sizes, there is no much difference between these two plans. For example, for input size (2^{11} and 2^{14}), the Plan I has only around 2% speedup over the base plan.

4.4 Exploration of Larger Work Units

Generally speaking, the performance of a parallel program is determined by two factors: the computation time and the synchronization overhead. In our case, the computation time 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. For example, for a 2^{11} point FFT, as shown in Table 2, it needs 11 stages computation, and 4 barriers. In [1], a specialized kernel function *r16v1-first* was used for the first

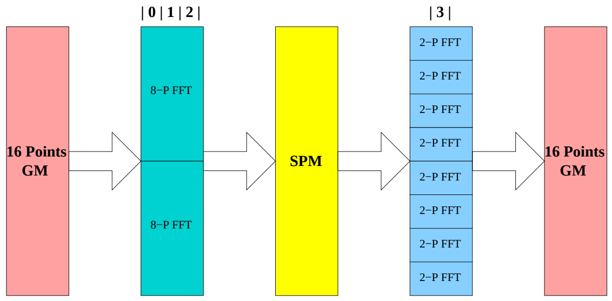

Figure 6: Optimize Kernel for 16-point Working Unit

4 stages computation. The authors also claimed that a general kernel function for 16-point work unit is not good for C64 because the number of registers needed exceeds the maximum available registers, which incurs performance degradation.

After carefully considering the architectural features, however, the performance of the general kernel function for 16-point work unit can be improved by using SPM, which is the fast memory in C64 with very low latency, i.e., 2 cycles for load and 1 cycle for store. Figure 6 illustrates this idea: (1) group the 16-point into two groups of 8-point data; for each group, read the 8-point data from SRAM, perform the 3-stage computation on the 8-point data and store the intermediate results back to the SPM; (2) regroup the 16-point into eight groups of 2-point data; for each group, load the 2-point data from SPM, perform 1-stage computation and store the results back to SRAM. We call this optimized general kernel function *r16v1*. By using this new kernel function, we get a new plan, *r16-first* + *r16v1* + *r8v1*, for 2^{11} point FFT, and one barrier can be eliminated. Experimental results show around 7% performance improvement compared with the Plan I in Table 2, when running with 128 threads.

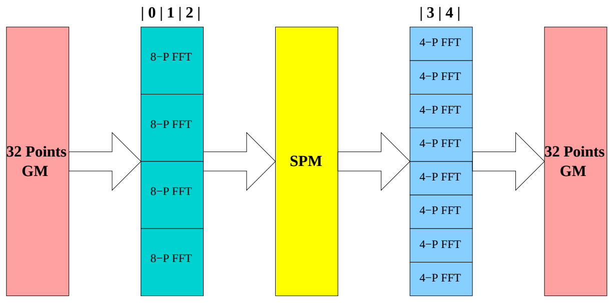

Figure 7: Optimize Kernel for 32-point Working Unit

Inspired by the above result, we decide to explore an even larger 32-point work unit. Similar optimizations used for *r16v1* are applied again for this 32-point work unit. Figure 7 illustrates this idea: (1) group the 32-point data into four groups of 8-point data; for each group, read the 8-point data from SRAM, perform the 3-stage computation on the 8-point data and store the intermediate results back to the SPM; (2) regroup the 32-point into eight groups of 4-point data; for each group, load the 4-point data from SPM, perform 2-stage computation and store the results back to SRAM. We call this kernel function *r32v1*.

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

Since now we have two more kernel functions, i.e., *r16v1 and r32v1*, we run the search algorithm described in Section 4.3 again. Table 3 shows the searching results. The first column is the input size. The second column shows the optimal number of threads for this specific input size. The third column and the fourth column show the plans without and with the new kernel functions, respectively. The differences between two plans are in bold. Furthermore, Figure 8 shows the speedup of both Plan I and

Dim	# of Threads	Plan I	Plan II
4	8	r16v1-first	r16v1-first
5	8	r4v1+r8v1	r4v1+r8v1
6	16	r8v1+r8v1	r8v1+r8v1
7	16	r16v1-first+r8v1	r16v1-first+r8v1
8	32	r16v1-first+r4v2+r4v2	r16v1-first+r4v2+r4v2
9	64	r16v1-first+r8v1+r4v2	r16v1-first+r8v1+r4v2
10	128	r16v1-first+r8v1+r8v1	r16v1-first+r8v1+r8v1
11	128	r16v1-first+r8v1+r8v1+r2v4	r16v1-first+**r16v1**+r8v1
12	138	r16v1-first+r8v1+r8v1+r4v2	r16v1-first+**r32v1**+r8v1
13	138	r16v1-first+r8v1+r8v1+r8v1	r16v1-first+**r32v1**+**r16v1**
14	138	r16v1-first+r8v1+r8v1+r8v1+r2v4	r16v1-first+**r16v1**+r8v1+r8v1
15	138	r16v1-first+r8v1+r8v1+r8v1+r4v2	r16v1-first+**r32v1**+r8v1+r8v1
16	138	r16v1-first+r8v1+r8v1+r8v1+r8v1	r16v1-first+**r32v1**+**r32v1**+r2v4

Table 3: Plan Comparison with Kernel Function of 16-point and 32-point Work Unit

Plan II over the base plan. The results show that, for the small input sizes (less than 2^9), there is no much difference between Plan I and Plan II. For the large sizes, Plan II has a better performance than that of Plan I. Since the overhead of barrier is proportional to the number of threads and the empirical optimal number of running threads for the small input sizes is much smaller than that for large input sizes. Therefore, for the small input sizes, barrier overhead is relatively small and thus reducing a barrier cannot achieve much performance gain. While for the larger input size, barrier overhead is higher because more threads are used and thus reducing a barrier can achieve much performance gain. As shown in Figure 8, for the input size equal or larger than 2^{11}, Plan II has a distinct speedup over the base plan, especially for the input size 2^{11} and 2^{15}, which has around 12% and 8% speedup respectively. We achieved 21.5 GFLOPS for the input size 2^{15}.

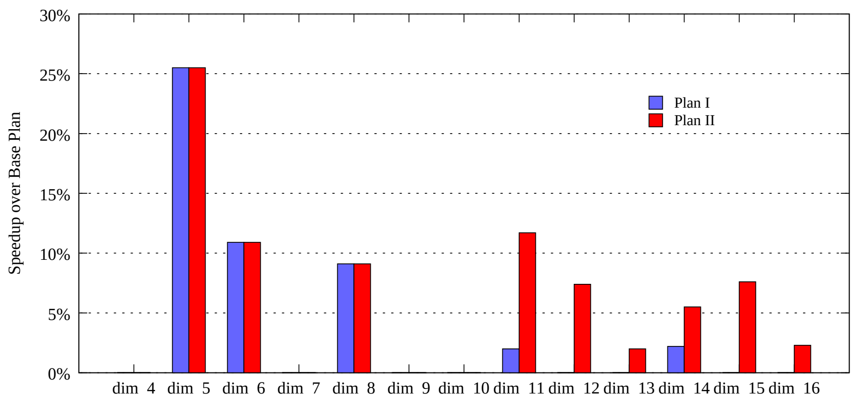

Figure 8: Search The Best Plans with Kernel Function for 16-point and 32-point work unit

5 Related Works

FFT has been extensively studied and implemented in various machines. Many researches have been done to address the distributed memory FFT implementations on the hypercube architecture [5,6]. Other parallel FFT implementation on arrays [7] and mesh architecture [8] have also been investigated. There are also some researches have been conducted in the shared memory FFT implementation [9]. The importance to consider the memory hierarchy to implement FFT effectively has been discussed in [10]. [11] also shows how to implement FFT efficiently by using local memory on CRAY-2. Two dataflow-based multithreaded FFT [12] are implemented in EARTH [13], a fine-grained data flow architecture. Besides that, there are several researches have been done to automatically tuning the FFT on different architectures. FFTW [2] planner generates various plans on the specific architecture 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 [14] uses a 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. FFTW 3.1 [15] implements a multithreading DFT implementation to support the parallel FFT computation. Recently, [16] presents a FFT implementation for shared memory, especially for the SMP and multi-core.

6 Conclusion

In this paper, we have presented an iterative search scheme to search the best plan for the FFT computation on the C64 multi-core architecture. The experiment results demonstrate that the plan derived from the search-based approach outperforms the fixed approach in [1]. In addition, a technique to optimize the large kernel functions by using SPM to achieve better performance has been discussed. This study is a good example of how to use SPM to optimize applications on C64.

Our study shows that application development for multi-core architectures like C64 is not easy: both the architecture features and application/algorithm properties have to been taken into account. This also poses more challenges for multi-core system software, especially for the compiler and code generator.

There are several researches can be done in the future. For example, how to utilize the SPM to optimize other applications. Automated FFT code generator for multi-core architectures with explicit memory hierarchy is also a very interesting topic. Besides that, we can also study the larger FFT problem which can not fit 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, ETI, the Department of Defense, the Department of Energy, the National Science Foundation, 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, Guangmin Tan and Shuxin Yang.

References

[1] Chen, L., Hu, Z., Lin, J., Gao, G.R.: Optimizing the fast fourier transform on a multi-core architecture. In: Workshop on Performance Optimization for High-Level Languages and Libraries in the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, USA (2007)

[2] Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (1998) 1381–1384

[3] J.W.Colley, J.W.Tukey: An algorithm for the machine calculation of complex fourier series. In: Math.Comput. Volume 4. (1965) 297–301

[4] del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: 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 (2005)

[5] S.L.Johnsson, Krawitz, R.: Cooley-tukey FFT on the connection machine. In: Parallel Computing. Volume 18. (1992) 1201–1221

[6] D.M.S.L.Johnsson, Krawitz, R., R.Frye: A radix 2 FFT on the connection machine. In: Proceeding of Supercomputing 89. (1989) 809–819

[7] Johnsson, S., D.Cohen: Computational arrays for the discrete fourier transform. (1981)

[8] Singh, V., V.Kumar, G.Agha, C.Tomlinson: Scalability of parallel sorting on mesh multicomputers. In: International Parallel Processing Symposium. (1991) 92–101

[9] Swarztrauber, P.: Multiprocessor FFTs. In: Parallel Computing. Volume 5. (1987) 197–210

[10] Bailey, D.: FFTs in external or hierarchical memory. In: Proceedings of the Supercomputing 89. (1989) 234–242

[11] D.A.Carlson: Using local memory to boost the performance of fft algorithms on the cray-2 supercomputer. In: J.Supercomput. Volume 4. (1990) 345–356

[12] P.Thulasiraman, Theobald, K., A.A.Khokhar, Gao, G.: Multithreaded algorithms for the fast fourier transform. In: ACM Symposium on Parallel Algorithms and Architectures. (2000) 176–185

[13] K.B.Thepbald: EARTH: An efficient architecture for running threads. In: PhD thesis. (1999)

[14] Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232–275

[15] Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. In: Proceedings of the IEEE 93. Volume 2. (2005) 216–231

[16] Franz Franchetti, Yevgen Voronenko, M.P.: FFT program generation for shared memory: SMP and Multicore. In: Proc. Supercomputing (SC). (2006)