FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine

David A. Bader and Virat Agarwal

College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
{bader, virat}@cc.gatech.edu

Abstract. The Fast Fourier Transform (FFT) is of primary importance and a fundamental kernel in many computationally intensive scientific applications. In this paper we investigate its performance on the Sony-Toshiba-IBM Cell Broadband Engine, a heterogeneous multicore chip architected for intensive gaming applications and high performance computing. The Cell processor consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs). We exploit the architectural features of the Cell processor to design an efficient parallel implementation of Fast Fourier Transform (FFT). While there have been several attempts to develop a fast implementation of FFT on the Cell, none have been able to achieve high performance for input series with several thousand complex points. We use an iterative out-of-place approach to design our parallel implementation of FFT with 1K to 16K complex input samples and attain a single precision performance of 18.6 GFLOP/s on the Cell. Our implementation beats FFTW on Cell by several GFLOP/s for these input sizes and outperforms Intel Duo Core (Woodcrest) for inputs of greater than 2K samples. To our knowledge we have the fastest FFT for this range of complex inputs.

1 Introduction

The Cell Broadband Engine (or the Cell/B.E.) [15,8,9,18] is a novel high-performance architecture designed by Sony, Toshiba, and IBM (STI), primarily targeting multimedia and gaming applications. The Cell BE consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. The Cell is used in Sony's PlayStation 3 gaming console, Mercury Computer System's dual Cell-based blade servers, and IBM's QS20 Cell Blades.

In this paper we present our design of an efficient parallel implementation of Fast Fourier Transform on the Cell Broadband Engine. FFT is of primary importance and a fundamental kernel in many computationally intensive scientific applications such as computer tomography, data filtering and fluid dynamics. Another important application area of FFTs is in spectral analysis of speech, sonar, radar, seismic and vibration detection. FFTs are also used in digital filtering, signal decomposition, and in solution of partial differential equations. The performance of these applications rely heavily on the availability of a fast routine for Fourier transforms.

The literature contains several publications related to FFTs on the Cell/B.E. processor. Williams *et al.* [19] analyze the Cell's peak performance for FFT of various types (1D, 2D), accuracy (single, double precision) and input sizes. Cico, Cooper and Greene [7] estimate the performance of 22.1 GFLOP/s for a single FFT that is reside in the local store of one SPE, or 176.8 GFLOP/s for computing 8 independent FFTs with 8K complex input samples. (Note that all other computation rates given in this paper – except for Cico *et al.* – consider the performance of a single FFT and include the overheads when considering that the source and output of the FFT are both stored in main memory.) In another work, Chow, Fossum and Brokenshire [6] achieve 46.8 GFLOP/s for a large FFT (16 million complex samples) on the Cell that is highly-specialized for this particular input size. FFTW on the Cell [11] is a highly-portable FFT library of various types, precision and input size.

In our design of FFTC we use an iterative out-of-place approach to solve 1D FFTs with 1K to 16K complex input samples. We describe our methodology to partition the work among the SPEs to efficiently parallelize a *single FFT computation* where the source and output of the FFT are both stored in main memory. This differentiates our work from the prior literature and better represents the performance that one realistically sees in practice. The algorithm requires a synchronization among the SPEs after each stage of FFT computation. Our synchronization barrier is designed to use inter SPE communication without any intervention from the PPE. The synchronization barrier requires only $2 \log p$ stages (p: number of SPEs) of inter SPE communication by using a tree-based approach. This significantly improves the performance, as PPE intervention not only results in a high communication latency but also in sequentialization of the synchronization step. We achieve a performance improvement of over 4 as we vary the number of SPEs from 1 to 8. We attain a performance of 18.6 GFLOP/s for a single-precision FFT with 8K complex input samples and also show significant speedup in comparison with other architectures. Our implementation is generic for this range of complex inputs. The source code is freely available from our CellBuzz project in SourceForge (`http://sourceforge.net/projects/cellbuzz/`).

This paper is organized as follows. We first describe the Fast Fourier Transform and the algorithm we choose to parallelize in Section 2. The novel architectural features of the Cell processor are reviewed in Section 3. We then present our design to parallelize FFT on the Cell and optimize for the SPEs in Section 4.

2 Fast Fourier Transform

Fast Fourier Transform (FFT) is an efficient algorithm that is used for computing the Discrete Fourier Transform. Some of the important application areas of FFTs have been mentioned in the previous section. There are several algorithmic variants of the FFTs that have been well studied for parallel processors and vector architectures [1,2,3,4].

In our design we utilize the naive Cooley-Tukey radix-2 Decimate in Frequency (DIF) algorithm. The pseudo-code for an out-of-place approach of this algorithm is given in Alg. 1. The algorithm runs in $\log N$ stages and each stage requires O(N) computation, where N is the input size.

Algorithm 1: Sequential FFT algorithm

```
Input: array $A[0]$ of size $N$
$NP \longleftarrow 1$ ;
$problemSize \longleftarrow N$;
$dist \longleftarrow 1$;
$i1 \longleftarrow 0$;
$i2 \longleftarrow 1$;
while $problemSize > 1$ do
    Begin Stage;
    $a \longleftarrow A[i1]$;
    $b \longleftarrow A[i2]$;
    $k = 0, jtwiddle = 0$;
    for $j \leftarrow 0$ to $N - 1$ step $2 * NP$ do
        $W \longleftarrow w[jtwiddle]$;
        for $jfirst \leftarrow 0$ to $NP$ do
            $b[j + jfirst] \leftarrow a[k + jfirst] + a[k + jfirst + N/2]$;
            $b[j + jfirst + Dist] \leftarrow (a[k + jfirst] - a[k + jfirst + N/2]) * W$;
        $k \leftarrow k + NP$;
        $jtwiddle \leftarrow jtwiddle + NP$;
    swap($i1, i2$);
    $NP \leftarrow NP * 2$;
    $problemSize \leftarrow problemSize/2$;
    $dist \leftarrow dist * 2$;
    End Stage;
Output: array $A[i1]$ of size $N$
```

The array w contains the *twiddle factors* required for FFT computation. At each stage the computed complex samples are stored at their respective locations thus saving a bit-reversal stage for output data. This is an iterative algorithm which runs until the parameter *problemSize* reduces to 1. Fig. 1 shows the butterfly stages of this algorithm for an input of 16 sample points (4 stages).

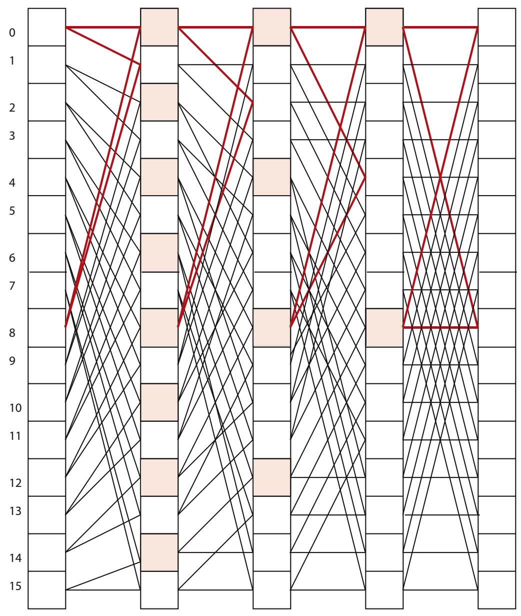


Fig. 1. Butterflies of the ordered DIF FFT algorithm

Apart from the theoretical complexity, another common performance metric used for the FFT algorithm is the floating point operation (FLOP) count. On analyzing the sequential algorithm, we see that during each iteration of the innermost *for* loop there is one complex addition for the computation of first output sample, which accounts for 2 FLOPs. The second output sample requires one complex subtraction and multiplication which accounts for 8 FLOPs. Thus, for the computation of two output samples during each innermost iteration we require 10 FLOPs, which suggests that we require 5 FLOPs for the computation of a complex sample at each stage. The total computations in all stages are $N \log N$ which makes the total FLOP count for the algorithm as $5N \log N$.

3 Cell Broadband Engine Architecture

The Cell Broadband Engine (Cell/B.E.) processor is a heterogeneous multi-core chip that is significantly different from conventional multiprocessor or multi-core architectures. It consists of a traditional microprocessor (the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Fig. 2 gives an architectural overview of the Cell/B.E. processor. We refer the reader to [17,10,16,12,5] for additional details.

The PPE runs the operating system and coordinates the SPEs. It is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KByte L1

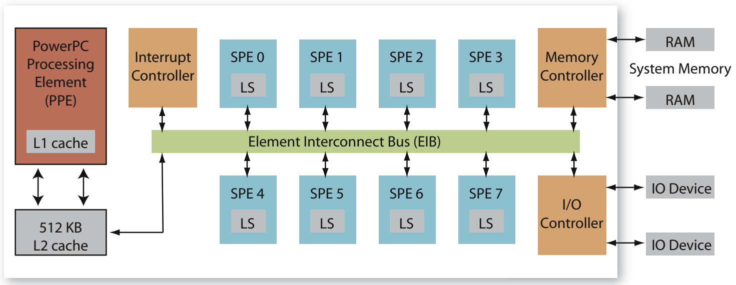


Fig. 2. Cell Broadband Engine Architecture

instruction and data caches, and a 512 KByte L2 cache. The PPE is a dual issue, in-order execution design, with two way simultaneous multithreading. Ideally, all the computation should be partitioned among the SPEs, and the PPE only handles the control flow.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a micro-architecture designed for high performance data streaming and data intensive computation. It includes a 256 KByte *local store* (LS) memory to hold SPU program's instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into the Local Store or write computation results back to the main memory. DMA is non-blocking so that the SPU can continue program execution while DMA transactions are performed.

The SPU is an in-order dual-issue statically scheduled architecture. Two SIMD [14] instructions can be issued per cycle: one compute instruction and one memory operation. The SPU branch architecture does not include dynamic branch prediction, but instead relies on compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as far as possible.

The MFC supports naturally aligned transfers of 1,2,4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 KBytes. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. Peak performance is achievable when both the effective address and the local storage address are 128 bytes aligned and the transfer is an even multiple of 128 bytes. In the Cell/B.E., each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in on-chip communication. Kistler *et al.* [16] analyze the communication network of the Cell/B.E. and state that applications that rely heavily on random scatter and or gather accesses to main memory can take advantage of the high communication bandwidth and low latency.

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 GFLOP/s (single precision). The EIB supports a peak bandwidth of 204.8 GB/s for intrachip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

4 FFTC: Our FFT Algorithm for the Cell/B.E. Processor

There are several architectural features that make it difficult to optimize and parallelize the Cooley-Tukey FFT algorithm on the Cell Broadband Engine. The algorithm is branchy due to presence of a doubly nested *for* loop within the outer *while* loop. This results in a compromise on the performance due to the absence of a branch predictor on the Cell. The algorithm requires an array that consists of the $N/2$ complex twiddle factors. Since each SPE has a limited local store of 256 KB, this array cannot be stored entirely on the SPEs for a large input size. The limit in the size of the local store memory also restricts the maximum input data that can be transferred to the SPEs. Parallelization of a single FFT computation involves synchronization between the SPEs after every stage of the algorithm, as the input data of a stage is the output data of the previous stage. To achieve high performance it is necessary to divide the work equally among the SPEs so that no SPE waits at the synchronization barrier. Also, the algorithm requires $\log N$ synchronization stages which impacts the performance. It is difficult to vectorize every stage of the FFT computation. For vectorization of the first two stages of the FFT computation it is necessary to shuffle the output data vector, which is not an efficient operation in the SPE instruction set architecture. Also, the computationally intensive loops in the algorithm need to be unrolled for best pipeline utilization. This becomes a challenge given a limited local store on the SPEs.

4.1 Parallelizing FFTC for the Cell

As mentioned in the previous section for best performance it is important to partition work among the SPEs to achieve load balancing. We parallelize by dividing the input array held in main memory into $2p$ chunks, each of size $\frac{N}{2p}$, where p is the number of SPEs.

During every stage, SPE i is allocated chunk i and $i + p$ from the input array. The basis for choosing these chunks for an SPE lies in the fact that these chunks are placed at an offset of $N/2$ input elements. For the computation of an output complex sample we need to perform complex arithmetic operation between input elements that are separated

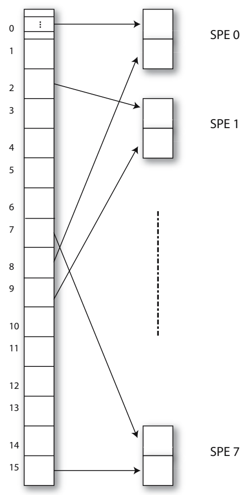


Fig. 3. Partition of the input array among the SPEs (e.g. 8 SPEs in this illustration)

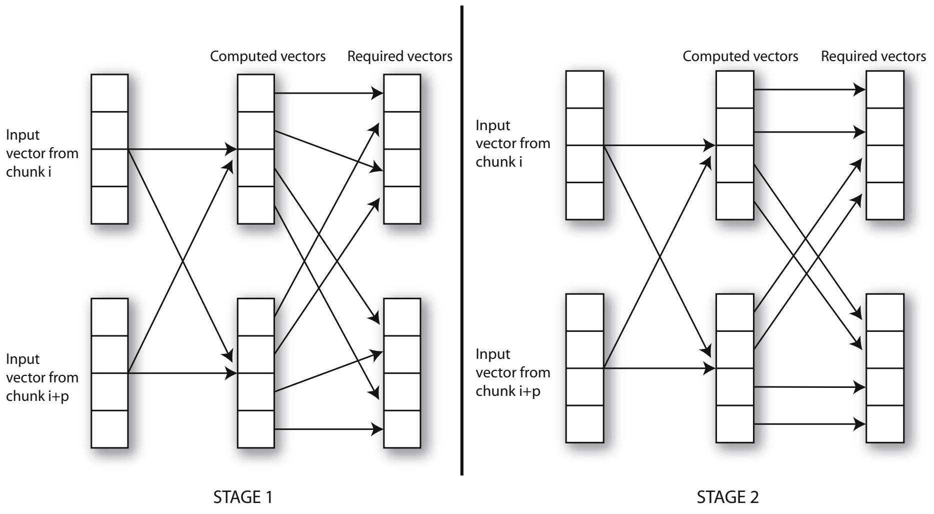

Computed vectors
Required vectors
Input vector from chunk i
Input vector from chunk i+p
STAGE 1
Computed vectors
Required vectors
Input vector from chunk i
Input vector from chunk i+p
STAGE 2

Fig. 4. Vectorization of the first two stages of the FFT algorithm. These stages require a shuffle operation over the output vector to generate the desired output.

by this offset. Fig. 3 gives an illustration of this approach for work partitioning among 8 SPEs.

The PPE does not intervene in the FFT computation after this initial work allocation. After spawning the SPE threads it waits for the SPEs to finish execution.

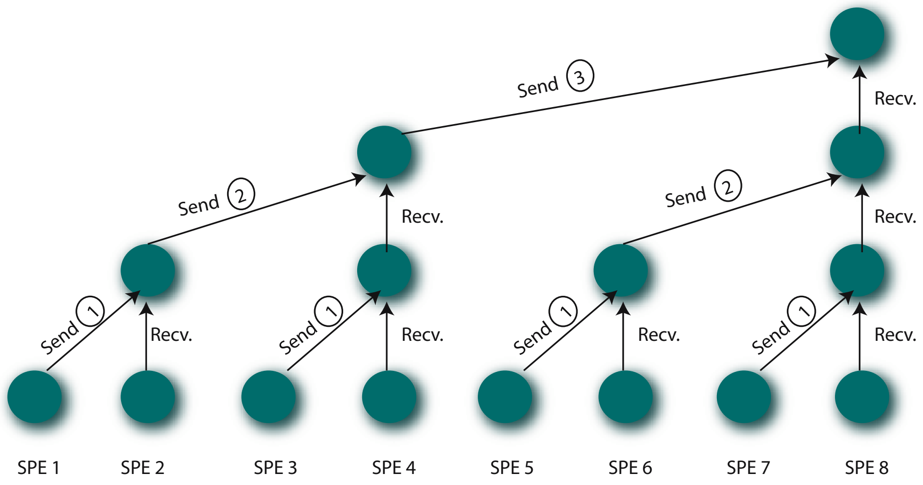


Fig. 5. Stages of the synchronization barrier using inter SPE communication. The synchronization involves sending inter SPE mailbox messages up to the root of the tree and then sending back acknowledgment messages down to the leaves in the same topology.

Algorithm 2: Parallel FFTC algorithm: View within SPE

```
Input: array in PPE of size $N$
Output: array in PPE of size $N$
$NP \longleftarrow 1$ ;
$problemSize \longleftarrow N$;
$dist \longleftarrow 1$;
$fetchAddr \longleftarrow$ PPE input array;
$putAddr \longleftarrow$ PPE output array;
$chunkSize \longleftarrow \frac{N}{2*p}$;
Stage 0 (SIMDization achieved with shuffling of output vector);
Stage 1 ;
while $NP < buffersize$ && $problemSize > 1$ do
    Begin Stage;
    Initiate all DMA transfers to get data;
    Initialize variables;
    for $j \leftarrow 0$ to $2 * chunkSize$ do
        Stall for DMA buffer;
        for $i \leftarrow 0$ to $buffersize/NP$ do
            for $jfirst \leftarrow 0$ to $NP$ do
                SIMDize computation as $NP > 4$;
            Update $j, k, jtwiddle$;
        Initiate DMA put for the computed results
    swap($fetchAddr, putAddr$);
    $NP \leftarrow NP * 2$;
    $problemSize \leftarrow problemSize/2$;
    $dist \leftarrow dist * 2$;
    End Stage;
    Synchronize using Inter-SPE communication;
while $problemSize > 1$ do
    Begin Stage;
    Initiate all DMA transfers to get data;
    Initialize variables;
    for $k \leftarrow 0$ to $chunkSize$ do
        for $jfirst \leftarrow 0$ to min($NP, chunkSize - k$) step $buffersize$ do
            Stall for DMA buffer;
            for $i \leftarrow 0$ to $buffersize$ do
                SIMDize computation as $buffersize > 4$;
            Initiate DMA put for the computed results;
        Update $j, k, jtwiddle$;
    swap($fetchAddr, putAddr$);
    $NP \leftarrow NP * 2$;
    $problemSize \leftarrow problemSize/2$;
    $dist \leftarrow dist * 2$;
    End Stage;
    Synchronize using Inter SPE communication;
```

4.2 Optimizing FFTC for the SPEs

After dividing the input array among the SPEs, each SPE is allocated 2 chunks each of size $\frac{N}{2p}$. Each SPE, fetches this chunk from main memory using DMA transfers and uses double-buffering to overlap memory transfers with computation. Within each SPE, after computation of each buffer, the computed buffer is written back into main memory at the correct offset using DMA transfers.

The detailed pseudo-code is given in Alg. 2. The first two stages of the FFT algorithm are duplicated, that correspond to the first two iterations of the outer *while* loop in sequential algorithm. This is necessary as the vectorization of these stages requires a shuffle operation (*spu_shuffle()*) over the output to re-arrange the output elements to their correct locations. Please refer to Fig. 4 for an illustration of this technique for stages 1 and 2 of the FFT computation.

The innermost *for* loop (in the sequential algorithm) can be easily vectorized for $NP > 4$, that correspond to the stages 3 through $\log N$. However, it is important to duplicate the outer *while* loop to handle stages where $NP <$ *buffersize*, and otherwise. The global parameter *buffersize* is the size of a single DMA get buffer. This duplication is required as we need to stall for a DMA transfer to complete, at different places within the loop for these two cases. We also unroll the loops to achieve better pipeline utilization. This significantly increases the size of the code thus limiting the unrolling factor.

SPEs are synchronized after each stage, using *inter-SPE communication*. This is achieved by constructing a binary synchronization tree, so that synchronization is achieved in $2 \log p$ stages. The synchronization involves the use of inter-SPE mailbox communication without any intervention from the PPE. Please refer to Fig. 5 for an illustration of the technique.

This technique performs significantly better than other synchronization techniques that either use chain-like inter-SPE communication or require the PPE to synchronize between the SPEs. The chain-like technique requires $2p$ stages of inter-SPE communication whereas with the intervention of the PPE latency of communication reduces the performance of this barrier.

5 Performance Analysis of FFTC

We use the Cell SDK 2.1 for instruction level profiling and performance analysis of the code. The code was compiled using the `xlc` compiler, that is included in the SDK, with level 3 optimization.

For parallelizing a single 1D FFT on the Cell, it is important to divide the work among the SPEs. Fig. 6 shows the performance of our algorithm with varying the number of SPEs for 1K and 4K complex input samples. The performance scales well with the number of SPEs which suggests that load is balanced among the SPEs.

Our design requires a barrier synchronization among the SPEs after each stage of the FFT computation. We focus on FFTs that have from 1K to 16K complex input samples. For relatively small inputs and as the number of SPEs increases, the synchronization cost becomes a significant issue since the time

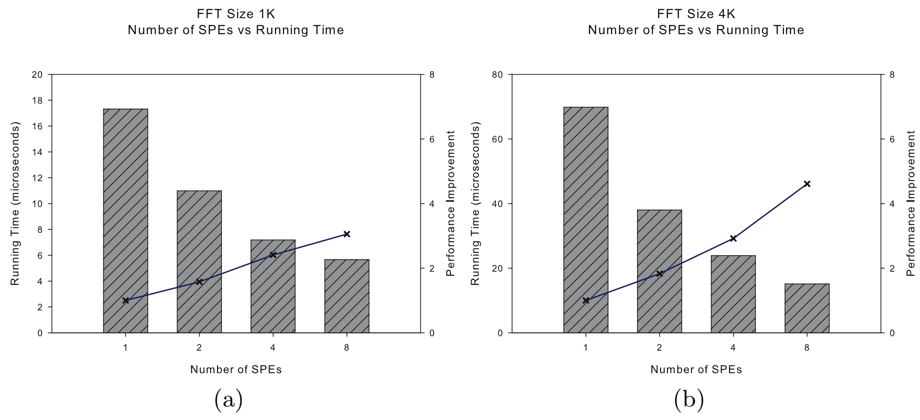


Fig. 6. Running Time of our FFTC code on 1K and 4K inputs as we increase the number of SPEs

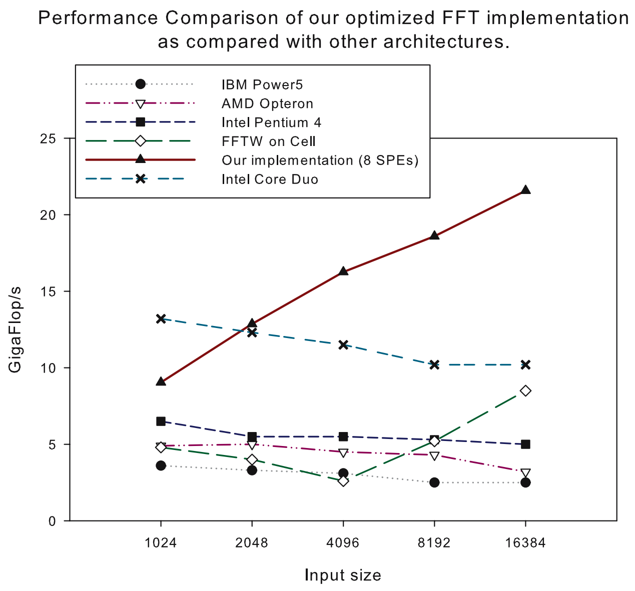
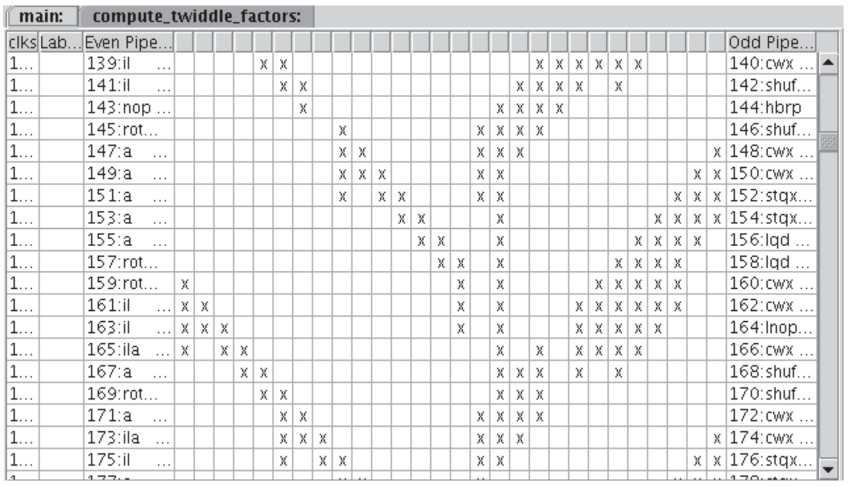
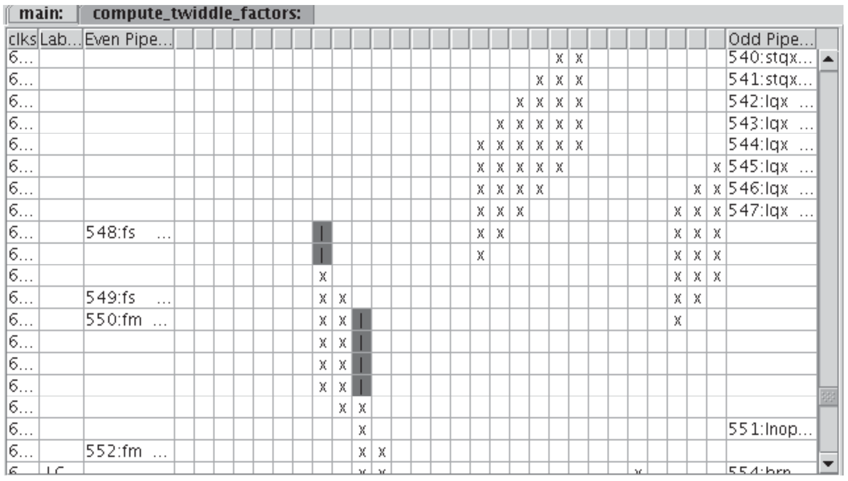


Fig. 7. Performance comparison of FFTC with other architectures for various input sizes of FFT. The performance numbers are from benchFFT from the FFTW website.

per stage decreases but the cost per synchronization increases. With instruction level profiling we determine that the time required per synchronization stage using our tree-synchronization barrier is about 1 microsecond (3200 clock cycles). We achieve a high performance barrier using inter-SPE mailbox communication which significantly reduces the time to send a message, and by using the tree-based technique we reduced the number of communication stages required for the barrier ($2 \log p$ steps).



clks	Lab...	Even Pipe...																													Odd Pipe...
1...		139:il ...					x	x													x	x	x	x	x	x					140:cwx ...
1...		141:il ...						x	x											x	x	x	x		x						142:shuf...
1...		143:nop ...							x										x	x	x	x									144:hbrp
1...		145:rot...									x							x	x	x	x										146:shuf...
1...		147:a ...									x	x						x	x	x										x	148:cwx ...
1...		149:a ...									x	x	x					x	x										x	x	150:cwx ...
1...		151:a ...									x		x	x				x	x									x	x	x	152:stqx...
1...		153:a ...												x	x				x								x	x	x	x	154:stqx...
1...		155:a ...													x	x			x							x	x	x	x		156:lqd ...
1...		157:rot...														x	x		x						x	x	x	x			158:lqd ...
1...		159:rot...	x														x		x					x	x	x	x	x			160:cwx ...
1...		161:il ...	x	x													x		x				x	x	x	x	x	x			162:cwx ...
1...		163:il ...	x	x	x												x		x				x	x	x	x	x				164:lnop...
1...		165:ila ...	x		x	x													x		x		x	x	x	x					166:cwx ...
1...		167:a ...				x	x												x	x	x		x		x						168:shuf...
1...		169:rot...					x	x											x	x	x										170:shuf...
1...		171:a ...						x	x									x	x	x	x										172:cwx ...
1...		173:ila ...						x	x	x								x	x	x										x	174:cwx ...
1...		175:il ...						x		x	x							x	x										x	x	176:stqx...



clks	Lab...	Even Pipe...																													Odd Pipe...
6...																						x	x								540:stqx...
6...																					x	x	x								541:stqx...
6...																				x	x	x	x								542:lqx ...
6...																			x	x	x	x	x								543:lqx ...
6...																		x	x	x	x	x	x								544:lqx ...
6...																		x	x	x	x	x								x	545:lqx ...
6...																		x	x	x	x								x	x	546:lqx ...
6...																		x	x	x								x	x	x	547:lqx ...
6...		548:fs ...								▮								x	x									x	x	x	
6...										▮								x										x	x	x	
6...										x																		x	x	x	
6...		549:fs ...								x	x																	x	x		
6...		550:fm ...								x	x	▮																x			
6...										x	x	▮																			
6...										x	x	▮																			
6...										x	x	▮																			
6...											x	x																			
6...												x																			551:lnop...
6...		552:fm ...										x	x																		

Fig. 8. Analysis of the pipeline utilization using the *IBM Assembly Visualizer for Cell Broadband Engine*. The top figure shows full pipeline utilization for certain parts of the code and the bottom figure shows areas where the pipeline stalls due to data dependency.

Fig. 7 shows the single precision performance for complex inputs of FFTC, our optimized FFT, as compared with the following architectures:

- **IBM Power 5:** IBM OpenPower 720, Two dual-core 1.65 GHz POWER5 processors.
- **AMD Opteron:** 2.2 GHz Dual Core AMD Opteron Processor 275.
- **Intel Duo Core:** 3.0 GHz Intel Xeon Core Duo (Woodcrest), 4MB L2 cache.
- **Intel Pentium 4:** Four-processor 3.06 GHz Intel Pentium 4, 512 KB L2.

We use the performance numbers from benchFFT [11] for the comparison with the above architectures. We consider the FFT implementation that gives best performance on these architectures for comparison.

The Cell/B.E. has a two instruction pipelines, and for achieving high performance it is important to optimize the code so that the processor can issue two instructions per clock cycle. This level of optimization requires inspecting the assembly dump of the SPE code. For achieving pipeline utilization it is required that the gap between dependent instructions needs to be increased. We use the *IBM Assembly Visualizer for Cell/B.E.* tool to analyze this optimization. The tool highlights the stalls in the instruction pipelines and helps the user to reorganize the code execution while maintaining correctness. Fig. 8 shows the analysis of pipeline utilization. Some portions utilize these pipelines effectively (top figure) whereas there are a few stalls in other parts of the code which still need to be optimized (bottom figure).

6 Conclusions

In summary, we present FFTC, our high-performance design to parallelize the 1D FFT on the Cell Broadband Engine processor. FFTC uses an iterative out-of-place approach and we focus on FFTs with 1K to 16K complex input samples. We describe our methodology to partition the work among the SPEs to efficiently parallelize a single FFT computation. The computation on the SPEs is fully vectorized with other optimization techniques such as loop unrolling and double buffering. The algorithm requires a synchronization among the SPEs after each stage of FFT computation. Our synchronization barrier is designed to use inter SPE communication only without any intervention from the PPE. The synchronization barrier requires only $2 \log p$ stages (p: number of SPEs) of inter SPE communication by using a tree-based approach. This significantly improves the performance, as PPE intervention not only results in a high communication latency but also results in sequentializing the synchronization step. We achieve a performance improvement of over 4 as we vary the number of SPEs from 1 to 8. We expect that the performance of FFTC will scale on the next generation of the IBM Cell Broadband Engine processor that may offer 32 SPEs [13]. We also demonstrate FFTC's performance of 18.6 GFLOP/s for an FFT with 8K complex input samples and show significant speedup in comparison with other architectures. Our implementation outperforms Intel Duo Core (Woodcrest) for input sizes greater than 2K and to our knowledge we have the fastest FFT for these range of complex input samples.

Acknowledgments

This work was supported in part by an IBM Shared University Research (SUR) award and NSF Grants CNS-0614915, CAREER CCF-0611589, and DBI-0420513. We would also like to thank Sidney Manning (IBM Corporation) for providing valuable inputs during the course of our research. We acknowledge our

Sony-Toshiba-IBM Center of Competence for the use of Cell Broadband Engine resources that have contributed to this research.

References

1. Agarwal, R.C., Cooley, J.W.: Vectorized mixed radix discrete Fourier transform algorithms. Proc. of the IEEE 75(9), 1283–1292 (1987)
2. Ashworth, M., Lyne, A.G.: A segmented FFT algorithm for vector computers. Parallel Computing 6(2), 217–224 (1988)
3. Averbuch, A., Gabber, E., Gordissky, B., Medan, Y.: A parallel FFT on an MIMD machine. Parallel Computing 15, 61–74 (1990)
4. Bailey, D.H.: A high-performance FFT algorithm for vector supercomputers. Intl. Journal of Supercomputer Applications 2(1), 82–87 (1988)
5. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell Broadband Engine Architecture and its first implementation. Technical Report (November 2005)
6. Chow, A.C., Fossum, G.C., Brokenshire, D.A.: A Programming Example: Large FFT on the Cell Broadband Engine. In: GSPx. Tech. Conf. Proc. of the Global Signal Processing Expo. (2005)
7. Cico, L., Cooper, R., Greene, J.: Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor. White paper (2006)
8. IBM Corporation. Cell Broadband Engine technology. `http://www.alphaworks.ibm.com/topics/cell`
9. IBM Corporation. The Cell project at IBM Research. `http://www.research.ibm.com/cell/home.html`
10. Flachs, B., et al.: A streaming processor unit for a Cell processor. In: International Solid State Circuits Conference, San Fransisco, CA, USA, vol. 1, pp. 134–135 (February 2005)
11. Frigo, M., Johnson, S.G.: FFTW on the Cell Processor (2007), `http://www.fftw.org/cell/index.html`
12. Hofstee, H.P.: Cell Broadband Engine Architecture from 20,000 feet. Technical Report (August 2005)
13. Hofstee, H.P.: Real-time supercomputing and technology for games and entertainment. In: Proc. SC, Tampa, FL (November 2006)(keynote talk)
14. Jacobi, C., Oh, H.-J., Tran, K.D., Cottier, S.R., Michael, B.W., Nishikawa, H., Totsuka, Y., Namatame, T., Yano, N.: The vector floating-point unit in a synergistic processor element of a Cell processor. In: ARITH 2005. Proc. 17th IEEE Symposium on Computer Arithmetic, Washington, DC, USA, pp. 59–67. IEEE Computer Society Press, Los Alamitos (2005)
15. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM J. Res. Dev. 49(4/5), 589–604 (2005)
16. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3), 10–23 (2006)
17. Pham, D., et al.: The design and implementation of a first-generation Cell processor. In: International Solid State Circuits Conference, San Fransisco, CA, USA, vol. 1, pp. 184–185 (February 2005)
18. Sony Corporation. Sony release: Cell architecture. `http://www.scei.co.jp/`
19. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientific computing. In: CF 2006. Proc.3rd Conference on Computing Frontiers, pp. 9–20. ACM Press, New York (2006)