The Pennsylvania State University

The Graduate School

College of Engineering

PFFTC: AN IMPROVED FAST FOURIER TRANSFORM FOR THE IBM CELL BROADBAND ENGINE

A Thesis in

Computer Science and Engineering

by

Andrew P. Shaffer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Andrew P. Shaffer was reviewed and approved* by the following:

Padma Raghavan
Professor of Computer Science and Engineering
Thesis Advisor

Sanjukta Bhowmick
Assistant Professor of Computer Science and Engineering

Bruce T. Einfalt
Research Engineer, Penn State Applied Research Laboratory
Special Signatory

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

The Fast Fourier Transform (FFT) is a widely used algorithm that is frequently employed in environments where high performance is critical. In the context of embedded systems, FFTs often have hard runtime constraints and must be evaluated using limited hardware.

In this thesis, we present a partitioned FFT algorithm (PFFTC) for the Cell Broadband Engine (Cell BE) that improves upon previous FFT implementations for this platform. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an "asynchronous" signal-based barrier method.

We evaluate the performance of PFFTC and other FFT algorithms for the Cell BE. Our results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS, and achieves speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm's reported performance for complex single-precision FFTs with 1,024-16,384 data points.

TABLE OF CONTENTS

List of Tables

List of Figures

1 INTRODUCTION

The Fast Fourier Transform (FFT) is a widely used algorithm that is frequently employed in environments where high performance is critical. In the context of embedded systems, FFTs often have hard runtime constraints and must be evaluated using limited hardware. FFTs that are on the critical path of a larger algorithm must be evaluated quickly to avoid delaying the completion of the entire algorithm. These factors indicate the importance developing FFT algorithms that can solve individual FFT problems with very low latency.

In this thesis, we present Partitioned Fastest Fourier Transform for the IBM Cell Broadband Engine (PFFTC) – an FFT algorithm for the Cell Broadband Engine (Cell BE) that improves upon previous FFT implementations for this platform. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an “asynchronous” signal-based barrier method.

We evaluate the performance of PFFTC and other FFT algorithms for the Cell BE. Our results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS, and achieves speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm’s reported performance for complex single-precision FFTs with 1,024-16,384 data points [1]. We compute that our algorithm achieves performance ranging from 15%

to 30% of the peak FFT performance that is possible on the Cell BE for these problem sizes.

The remainder of this thesis is organized as follows: we first describe the Cell BE architecture and prior FFT implementations for the Cell BE in Section 2. We describe the design of our new PFFTC algorithm and discuss its optimization features in Section 3. We then describe our testing environment and methodology in Section 4, and present the results of our performance and accuracy evaluations in Section 5. We conclude with a review of our optimization techniques that can be applied to other problems and discuss further improvements that might be made to our PFFTC algorithm in Section 6.

2 BACKGROUND

As shown in Figure 1, the Cell BE is a multicore architecture containing heterogeneous processing elements. It is capable of very high single precision floating point computation rates, and has been extensively detailed in previous literature [2],[3],[4],[5]. We also present a survey of basic performance results for benchmarks that we have run on the Cell BE in Appendix A. In this section, we provide a brief overview of the Cell BE architecture and several prior FFT implementations that have been developed for this platform.

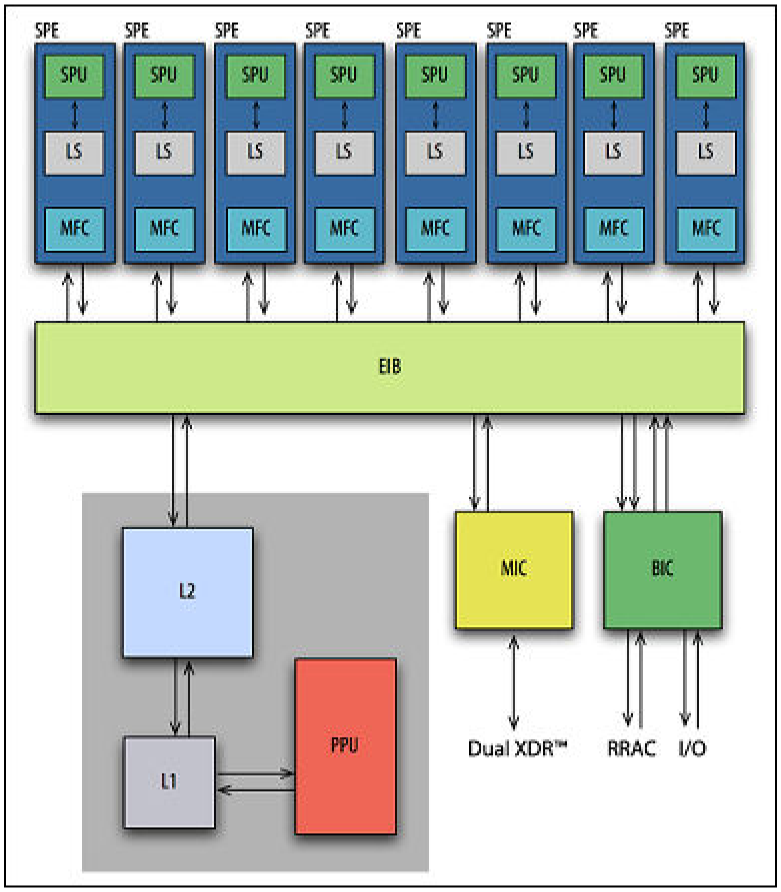

Figure 1: The Cell BE Architecture.

Source: A Rough Guide to Scientific Computing on the PlayStation 3 [2]

The heart of the Cell BE's design is an array of up to eight Synergistic Processing Elements (SPEs), which are specialized SIMD processors optimized for high single-precision floating point performance. Each SPE has access to a personalized 256 KB local store and a 128-entry quadword register file. Each SPE is also capable of dual issue of memory and arithmetic instructions and can sustain 25.6 GFLOPS of single-precision floating point math, although this math is not fully IEEE Standard 754-compliant [4].

Because the SPEs have only a small local store and because they sacrifice some general-purpose processor functions to achieve high single precision floating point computation rates on a small chip, the SPEs are much better suited for running highly optimized math kernels than they are for evaluating general purpose programs. To support general programs like the Linux operating system and to coordinate the activities of the SPEs, the Cell BE also contains a PowerPC Processing Element (PPE). The PPE houses a traditional PowerPC processor and its cache hierarchy. SPE-driven programs on the Cell BE platform are typically initiated on the PPE and then allocate one or more SPEs to assist with the execution of their high performance components.

To support data movement between the SPEs, the PPE, and the system's main memory, these components are connected by a high-bandwidth Element Interconnect Bus (EIB). Data is communicated between elements on the bus via application-issued DMA commands, allowing the programmer to explicitly manage memory access patterns and to finely tune double buffering schemes. Each SPE, as well as the main memory, is capable of supporting a bandwidth of 25.6 GB/sec.

Previous FFT implementations for the Cell BE have taken a variety of approaches. FFTW operates by dynamically generating an execution plan built from a library of statically optimized FFT algorithm components [6],[7]. The selection of components for the execution plan is based upon the size of the problem being solved and the performance of each of the components on the underlying hardware platform. This makes FFTW highly portable, but does not allow it to take advantage of optimizations that are specific to the Cell BE.

In contrast, Mercury Computer Systems' FFT_ZIPX algorithm relies on a highly tuned FFT kernel that has been extensively modified to run efficiently on a single SPE [8]. The FFT_ZIPX kernel attains nearly the theoretical limit of a single SPE's FFT performance [9], but the Cell BE implementation of this kernel cannot be run on any other system because it is so specialized. Also, because FFT_ZIPX is designed to solve an FFT problem using only a single SPE, it leaves most of the Cell BE's hardware unused. Multiple instances of FFT_ZIPX can be run simultaneously to utilize the Cell BE's SPEs to the maximum extent possible when solving large sets of FFT problems, but the implementation's use of only one SPE per FFT problem means that it cannot achieve the theoretical minimum latency for solving a single FFT problem on the Cell BE.

A third approach is used by the FFTC algorithm, in which a single FFT problem is distributed across all eight SPEs in the Cell BE [1]. The FFTC algorithm implements an iterative out-of-place FFT algorithm that distributes each butterfly stage across all eight SPEs, with the data for each stage being taken from main memory, processed, and then returned to main memory at the end of the stage. This approach necessitates moving the problem data between main memory and the SPEs' local stores multiple times throughout the problem execution. It also requires synchronizing the SPEs at the end of each butterfly stage. However, FFTC's approach allows all of the SPEs to be used in the solution of a single FFT problem. The FFTC package reports the best performance for solving small single-precision complex FFT problems on the Cell BE as of the time of its publication, and it still has the best previously published performance for these problems of any Cell BE FFT algorithm of which we are aware.

3 PFFTC: A PARTITIONED FFT ALGORITHM FOR THE IBM CELL BE

PFFTC is a low-latency FFT implementation that solves single-precision complex FFT problems up to 16,384 data points using 2-8 SPEs. It requires no working buffer in main memory, allows prefetching of upcoming problem data to improve throughput, and supports efficient on-the-fly switching between forward and inverse FFTs.

As shown in Figure 2, PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. In this section, we describe the operations performed by PFFTC during each phase, and also the optimizations that we used to make the algorithm operate efficiently.

3.1 Partitioning Stage

The partitioning stage is designed to divide a single FFT problem into four, eight or sixteen smaller independent sub-problems that can be solved independently in parallel. The sub-problems that are created by the partitioning stage correspond exactly to the sub-problems that would be considered by the lowest level of recursion in a recursive implementation of the FFT algorithm if the recursive algorithm were cut off at a fixed recursion depth of two, three, or four levels of in-place recursive partitioning. However, to increase efficiency, the partitioning is performed in a single pass over the data by means of hard-coded permutation functions that scan the input from start to finish. These functions apply the composition of either two, three or four levels of recursive partitioning in a single operation.

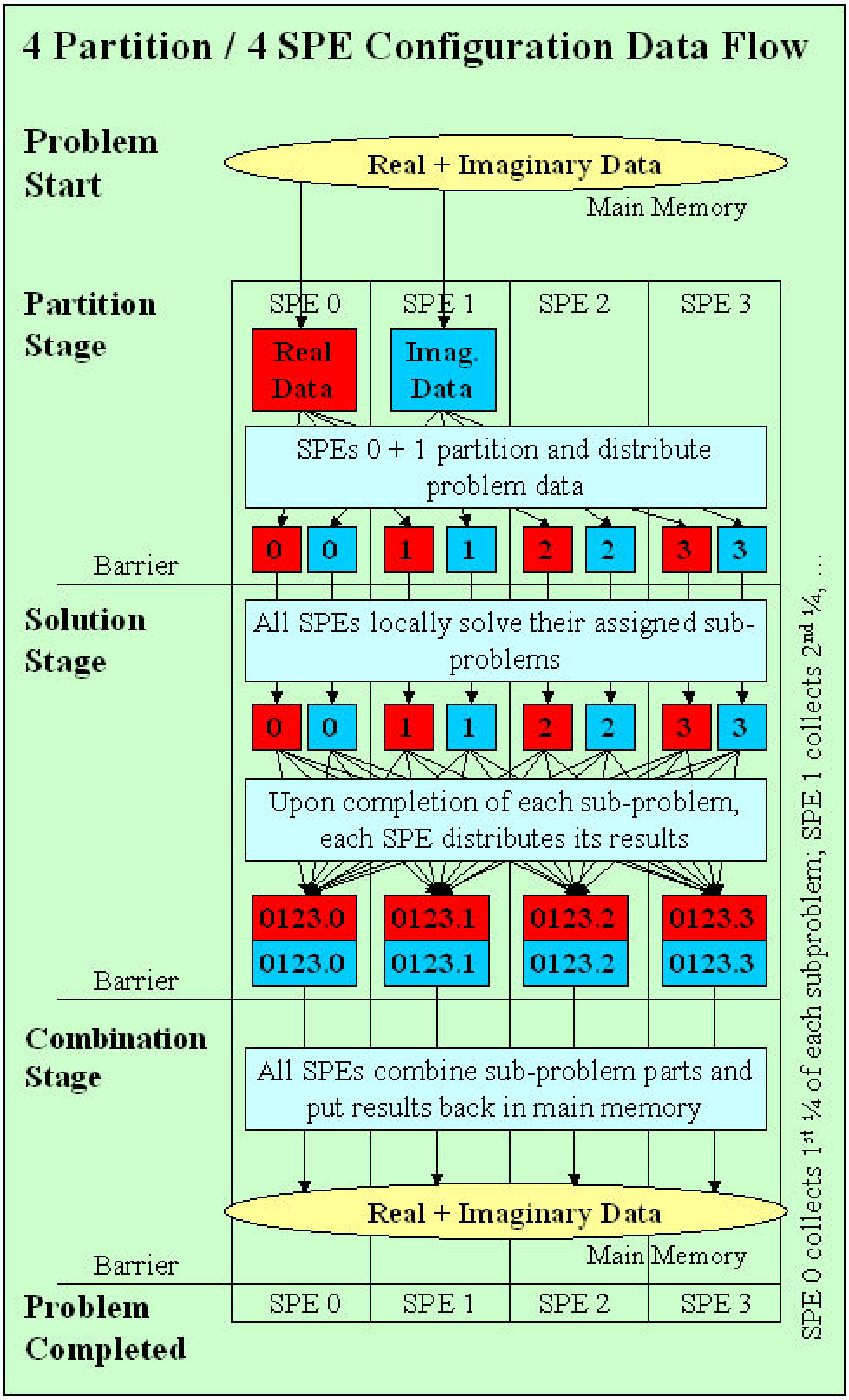
4 Partition / 4 SPE Configuration Data Flow
Problem Start
Real + Imaginary Data
Main Memory
Partition Stage
SPE 0
SPE 1
SPE 2
SPE 3
Real Data
Imag. Data
SPEs 0 + 1 partition and distribute problem data
0 0 1 1 2 2 3 3
Barrier
Solution Stage
All SPEs locally solve their assigned sub-problems
0 0 1 1 2 2 3 3
Upon completion of each sub-problem, each SPE distributes its results
0123.0
0123.1
0123.2
0123.3
Barrier
Combination Stage
All SPEs combine sub-problem parts and put results back in main memory
Real + Imaginary Data
Main Memory
Barrier
Problem Completed
SPE 0
SPE 1
SPE 2
SPE 3
SPE 0 collects 1st ¼ of each subproblem; SPE 1 collects 2nd ¼, …

Figure 2: PFFTC Data Flow Diagram.

The partitioning stage is only ever executed on SPE 0 and SPE 1, regardless of how many SPEs are allocated to PFFTC. SPE 0 partitions the problem's real data and SPE 1 partitions the problem's imaginary data. The use of two SPEs to perform the partitioning stage reduces the stage's processing time by 50%, and also ensures that the full 25.6 GB/sec bandwidth of the main memory is used to transfer the problem data to the SPEs. The use of only two SPEs to perform the partitioning stage also ensures that the partitioned data for each sub-problem is located in exactly two contiguous blocks in the local stores of the SPEs, simplifying data routing and distribution in the subsequent solution stage.

Double buffering is used to bring the problem data into the local stores of SPE 0 and SPE 1 in multiple pieces, allowing partitioning to begin before the transfer of problem data has been completed. Two data blocks are used to minimize DMA control overhead for problems with 2,048 or fewer data points. For larger problem sizes, four data blocks are retrieved with the double buffering.

Once the partitioning functions have finished their processing, all of the sub-problems' real data resides in contiguous blocks on SPE 0 and all of the sub-problems' imaginary data resides in contiguous blocks on SPE 1. SPE 0 and SPE 1 finish the stage by forwarding the first sub-problem that will be solved by each SPE to the SPEs that will solve the sub-problems.

3.2 Solution Stage

During the solution stage, the independent sub-problems created during the partitioning stage are solved in parallel on the allocated SPEs. The sub-problems are distributed among the SPEs in round-robin fashion to balance the workload, and are solved using an iterative FFT kernel derived from the FFTC algorithm.

To achieve high performance in the iterative algorithm used to solve each sub-problem, we use one set of variables to hold the initial data for each butterfly computation, a second set of variables to hold the intermediate values, and a third set of variables to hold the final output of the butterfly computation until it can be written back to the local store. The first iteration of the butterfly stage loop is manually unrolled so that the twiddle factors needed by the loop and the input data needed for the second loop iteration can be loaded concurrently with the execution of the first loop iteration. Then, for each successive iteration of the butterfly stage loop, the input values for the subsequent iteration are prefetched and the output values of the previous iteration are output while the intermediate and final values for the current iteration are computed. The output of the final loop iteration is written out to the local store after the loop has ended. In this way, we are able to avoid memory access stalls when data must be read from or written to the local store throughout the main part of the loop body, in a manner similar to the way in which double buffering prefetches the data for a later problem while a current problem is being computed.

Additionally, we further improve performance by manually unrolling the pipelined butterfly stage loop body four times. This degree of unrolling is possible because of the large size of the SPE register file. The execution of four butterfly computations per loop iteration improves performance by providing four sets of independent instructions in the loop body that can be ordered to avoid all data dependency stalls between the operations producing the intermediate results and the operations needing these values as input to compute the final results for each butterfly computation. In combination with the instruction-level double buffering optimization discussed previously, this loop unrolling allows our algorithm to avoid data dependency and local store access stalls altogether during its main processing loop.

To keep the solution stage running as quickly as possible, all of the sub-problems solved on each SPE except for the forwarded first sub-problem are retrieved from the local stores of SPE 0 and SPE 1 using double buffering. The combined bandwidth of SPE 0 and SPE 1 is 51.2 GB/sec, making the solution stage decidedly compute-bound. We take advantage of the excess communication capacity available while the solution stage is running by having the solution stage forward the output of each sub-problem to the SPEs where it will be needed in the subsequent combination stage as soon as each sub-problem has been solved.

3.3 Combination Stage

The combination stage corresponds directly to an iterative version of the recursive combinations performed by a recursive implementation of the FFT algorithm. Our algorithm to combine the sub-problems operates in-place, and performs as many passes over the data as the recursion depth cut-off that is used when creating the sub-problems in the partitioning stage.

As shown in Figure 3, each butterfly stage in an in-place recursive FFT algorithm combines pairs of adjacent sub-problems into a single larger sub-problem with size equal to the combined size of the original sub-problems, and each butterfly stage computation places its output values into the same memory locations as its input values. Thus, if one SPE is provided with the first data point of each sub-problem, it will be able to compute

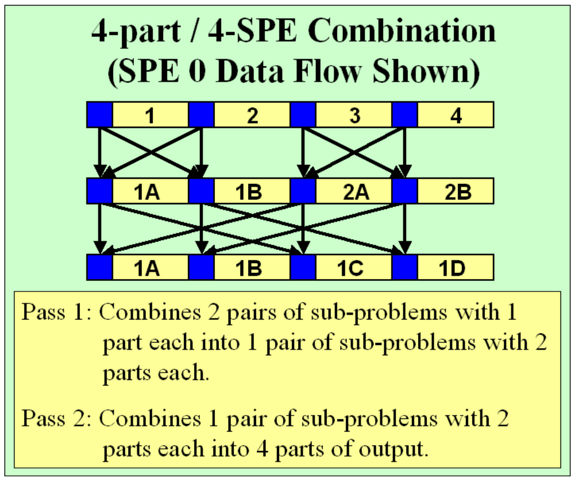
4-part / 4-SPE Combination
(SPE 0 Data Flow Shown)
1
2
3
4
1A
1B
2A
2B
1A
1B
1C
1D
Pass 1: Combines 2 pairs of sub-problems with 1 part each into 1 pair of sub-problems with 2 parts each.
Pass 2: Combines 1 pair of sub-problems with 2 parts each into 4 parts of output.

Figure 3: Communication Free Combination Stage.

the butterfly computations involving the first data point of each sub-problem for an initial butterfly stage. Then, because the memory locations of the output for each butterfly stage are the same as the butterfly computation's input data, it will be able to compute the butterfly computations involving the first data point of the sub-problems from the previous butterfly stage's output, and also the butterfly computations involving the midpoint of the sub-problems from the previous butterfly stage's output. This pattern continues for all points in the original set of sub-problems and for as many levels of the recursion as were created during the partitioning stage, so if an SPE is provided with the mth through nth elements of each sub-problem, it can compute all of the remaining butterfly stages corresponding to these data points without requiring any additional data from any other SPE.

We maximize DMA efficiency and balance the load of the combination stage across all allocated SPEs by assigning the first portion of each sub-problem to SPE 0, the second portion to SPE 1, and so on for all allocated SPEs, such that the portions are balanced evenly. The portion of each sub-problem that is needed by each SPE for the combination stage is forwarded to where it is needed as each sub-problem is computed during the solution stage. Then, once the solution stage has finished, all of the data is already in place for the combination stage to run to completion without any further inter-SPE communication.

Lastly, the combination stage is designed to ensure that the FFT result is returned to main memory as quickly as possible. This is accomplished by beginning the transfer of each portion of the problem result back to main memory as soon as it has been computed.

3.4 Asynchronous Synchronization

At the end of each of the three major stages in the PFFTC algorithm it is necessary to synchronize the SPEs. After the partitioning stage, a synchronization is necessary to ensure that the forwarded first sub-problem that will be solved by each SPE has been completely transmitted before the SPE begins to solve it. This synchronization also ensures that the data for any later sub-problems is ready to be prefetched before it is retrieved for use in the solution stage. Then, once the solution stage has been completed, a second synchronization is needed to ensure that all data forwarding performed during the solution stage has been completed before allowing the SPEs to work with this data in the combination stage. Finally, after the combination stage, a third synchronization is necessary to ensure that every SPE has completely transmitted its portion of the original problem result back to main memory before notifying the PPE that the result is ready.

We chose to implement the synchronization operations using a signal-based method to reduce the impact of these synchronization operations on PFFTC's throughput. Each SPE has two signal registers that can be placed in logical-OR mode to accumulate the set bits of any messages that are received, and we assign each SPE used by our algorithm a unique signal message equal to $2^{\text{spe id}}$. As shown in Figure 4, whenever a synchronization is performed, each SPE sends its unique signal message to every other SPE upon reaching the beginning of the synchronization operation. These signals can be enqueued almost instantaneously and they are sent asynchronously, so the SPE can resume processing that is unrelated to the synchronization operation as soon as the signal enqueuing is complete. Once the SPE has completed its non-synchronized work, it can check which of the other SPEs have reached the synchronization start point by polling its signal register to see which bits have been set. After the bit corresponding to each allocated SPE has been set in the signal register, the polling SPE has established that all of the other SPEs have also reached the synchronization start point, and it can safely continue beyond the synchronization operation.

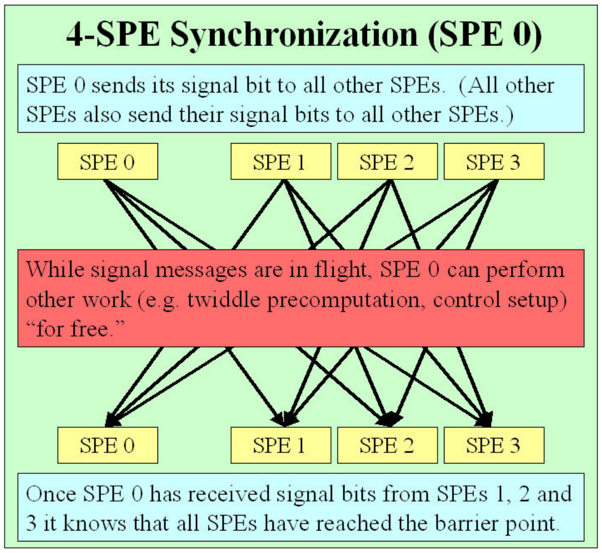
4-SPE Synchronization (SPE 0)
SPE 0 sends its signal bit to all other SPEs. (All other SPEs also send their signal bits to all other SPEs.)
SPE 0
SPE 1
SPE 2
SPE 3
While signal messages are in flight, SPE 0 can perform other work (e.g. twiddle precomputation, control setup) "for free."
SPE 0
SPE 1
SPE 2
SPE 3
Once SPE 0 has received signal bits from SPEs 1, 2 and 3 it knows that all SPEs have reached the barrier point.

Figure 4: Asynchronous Synchronization.

Although the signal-based barrier requires a significantly more messages to be sent between SPEs than is necessary in a tree-based synchronization, these messages can all be sent independently from one another, allowing the communication on all SPEs to start as early as possible and to be amortized over a longer runtime without affecting performance. For instance, if eight SPEs are allocated to PFFTC, SPEs 2-7 can send all of their synchronization messages for the synchronization at the end of the partitioning stage while SPE 0 and SPE 1 are performing the partitioning stage. Then, when SPE 0 and SPE 1 have finished their work for the partitioning stage, each of these two SPEs needs to send only seven independent signal messages to complete the barrier operation. Since the EIB has four bus channels and all of the signals being sent are independent, sending seven messages from two SPEs will require a maximum of four signal delay periods. Likewise, at the end of the combination stage, every SPE can send its synchronization signal to every other SPE except SPE 0 even before the problem data has been fully transmitted back to main memory. This is possible because only SPE 0 needs to know when each SPE has completed its data transmission so that it can notify the PPE when the problem has been completed. If eight SPEs are allocated, this allows all but seven of the synchronization signals to be sent early, so the number of signals on the algorithm's critical path is minimal. Since the EIB has four bus channels and all of the signals being sent are independent, sending one message from each of seven SPEs will require a maximum of two signal delay periods. Both of these cases experience significantly less than the six signal delay periods on the algorithm's critical path needed to perform the gather and scatter operations that would be used by a tree-based synchronization method.

4 ENVIRONMENT & METHODOLOGY

In this section, we discuss our testing environment and the methodology that we used to evaluate the latency and accuracy of our new PFFTC algorithm and the latency of prior Cell BE FFT packages.

We made use of two PlayStation 3 systems and one IBM QS20 blade server in our evaluations. These systems are pictured in Figure 5. The PlayStation 3 systems each support a single Cell BE with six SPEs, and the blade server supports two Cell BEs which each contain eight SPEs. All of these platforms had the IBM Cell BE Software Development Kit (SDK) version 2.1 installed.

Figure 5: IBM QS20 Blade Server and Sony PlayStation 3.

The FFTW package requires an outdated version of the IBM SDK, so we chose not to test it empirically. Instead, we present its reported results for both the PlayStation 3 and the blade server platforms from the FFTW website [7].

Next, the FFT_ZIPX package from Mercury Computer Systems only runs under Yellow Dog Linux 6.0. The blade server implementation for this package is quite expensive, so we evaluated only the less expensive PlayStation 3 implementation for this package. Our PlayStation 3 for the FFT_ZIPX tests runs under Yellow Dog Linux kernel 2.6.23-9.ydl6.1.

After evaluating FFT_ZIPX, we found that FFTC requires eight SPEs to operate, so it could only be evaluated on the blade server system [1]. Our blade server system runs under Linux kernel 2.6.23.

Lastly, we designed our PFFTC package to operate using between 2-8 SPEs, so it is capable of operating on both the PlayStation 3 and the blade server platforms. We evaluate its performance on both of these systems. The PlayStation 3 system used to test our PFFTC package runs under Linux kernel 2.6.16.

When measuring the runtime of each algorithm, we track the time that elapses between the first SPE being notified that a problem is ready in main memory until the PPE is notified that the problem result is fully stored in the main memory. We measure the runtime needed to solve 1,000 FFT problems in sequence and then report the average runtime needed to solve each problem in the 1,000 problem test to ensure that our tests run long enough to be timed accurately. We also report the best average timing result from a sample of 100 test runs to filter out the effect of transient system processes and network traffic, and we put each FFT algorithm through a “warm-up” run before beginning its main timing loop to avoid the effects of TLB faults and page faults.

Our test loop for each package is structured to completely solve each FFT problem from start to finish before beginning any work for any subsequent problems so that our timing effectively measures the latency of solving a single FFT problem in isolation. The only exception to this is that we allow the PFFTC package to prefetch the memory address of each FFT problem’s input data during the solution of the previous FFT problem in the test series, so that this address is resident in the local store of each SPE at the beginning of the problem. This accommodation for PFFTC is intended to make its timing results more comparable with those of the other packages, because the other packages do not perform the step of retrieving unique problem information for each FFT problem that is solved in their test series. Instead, the other FFT packages transmit information about the location of problem input buffers once before beginning their main timing loop and then solve this same FFT problem repeatedly.

The PFFTC package accepts command line parameters to specify how many partitions should be created and how many SPEs should be allocated. For each problem size, we evaluated every possible combination of partition counts and SPE allocations. We report the best average runtime over all of these combinations as the runtime for each problem size for PFFTC.

Additionally, for the PFFTC algorithm we ran a series of 1,000 tests for each problem size that we considered to verify the correctness of our algorithm. Each test took as input an appropriately sized array of normally-distributed random single precision complex input data with a mean of zero and a standard deviation of one and performed an FFT identity operation on the test input data by running a forward FFT followed immediately by an inverse FFT on the output of the original forward FFT problem. The partition and SPE count parameters for each test were chosen to match those that were found to provide the fastest runtime for each problem size. This sequence of operations returns the original input values exactly if the component FFT operations introduce no errors at all, so any differences that occur between the input and output are due to the implementation of the FFT algorithm. We computed the inf-norm of the differences between the output of our FFT identity operation and the original input for each test case, and report the maximum inf-norm observed across the 1,000 tests for each problem size as a measure of the total error introduced by our algorithm's implementation.

5 RESULTS

In this section, we report and analyze the performance and accuracy results of our empirical testing for the PFFTC package and prior Cell BE FFT packages. We also discuss the effect of recursion depth cut-off in the PFFTC partitioning stage and the impact of SPE allocation on overall PFFTC performance.

Using the metric that an FFT problem requires $5*N*\log_2(N)$ floating point operations, our timing results indicate that PFFTC attains a peak processing rate of 33.6 GFLOPS. As shown in Figure 6, PFFTC achieves speedups ranging from 31% to 62% over FFTC [1], which had the highest previously reported FFT performance for problems with 1,024-16,384 data points.

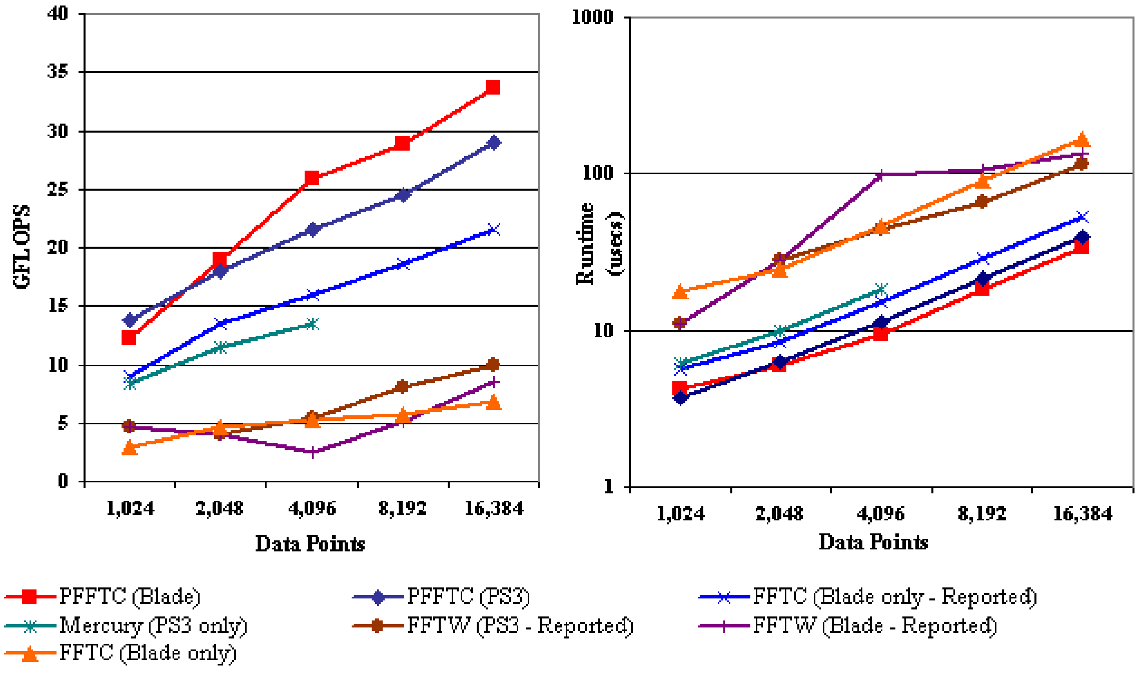

Figure 6: FFT GFLOPS and Runtime vs. Problem Size.

Given that at most $5*N*\log_2(N)$ floating point operations are performed for an FFT problem with N data points, the 25.6 GB/sec main memory bandwidth is a critical bottleneck in Cell BE FFT performance for small problem sizes. Although a Cell BE with eight SPEs is capable of 204.8 GFLOPS of single precision floating point performance, the need for each FFT problem to be moved from the main memory to the SPE local stores and for a result of equal size to be moved back to main memory creates a minimum communication delay for each size of FFT problem that prevents the peak single-precision floating point processing rate from being attained. Even if all FFT computation could be masked with communication operations, the communication requirements alone would lead to the theoretical performance limits shown in Table 1.

Table 1: Maximum Cell BE FFT Performance for Small Problem Sizes.

Problem Size (points)	Data Size (bytes)	2-Way Transfer Time (microseconds)	Floating Point Operations	Maximum GFLOPS
1,024	8,192	0.64	51,200	80
2,048	16,384	1.28	112,640	88
4,096	32,768	2.56	245,760	96
8,192	65,563	5.12	532,480	104
16,384	131,072	10.24	1,146,880	112

In comparison to these maximum theoretical performance values, our PFFTC algorithm attains efficiency ratings of between approximately 15% and 30%, as shown in Table 2.

Table 2: PFFTC Efficiency for Small Problem Sizes.

Problem Size (points)	Maximum GFLOPS	PS3 GFLOPS	Blade GFLOPS	PS3 Efficiency	Blade Efficiency
1,024	80	13.8	12.2	17.3%	15.2%
2,048	88	17.9	18.9	20.3%	21.5%
4,096	96	21.5	25.9	22.4%	27.0%
8,192	104	24.5	28.9	23.6%	27.8%
16,384	112	29.1	33.6	25.9%	30.0%

These results reflect the best performance achieved for each problem size across all possible combinations of partition counts and SPE allocations. As shown in Table 3, we find that a variety of workload distribution strategies allow PFFTC to achieve its best performance for different problem size and platform combinations.

Table 3: Optimal Partitioning and SPE Allocation for Small Problem Sizes on PlayStation 3 and Blade Server.

Problem Size (points)	Optimal PS3 Partitioning / SPE Allocation	Optimal Blade Partitioning / SPE Allocation
1,024	4 / 4	4 / 4
2,048	4 / 4	8 / 8
4,096	4 / 4	8 / 8
8,192	16 / 6	16 / 8
16,384	16 / 6	16 / 8

For smaller problem sizes, communication has a more significant impact on runtime than computation. Thus, we find that PFFTC attains its best performance for these problems using workload distribution strategies that reduce the number of required communication and synchronization messages at the expense of sacrificing some computational power. On the other hand, computation plays the most significant role in the runtime of the larger problem sizes. For these problem sizes, PFFTC attains its best performance by allocating the maximum possible number of SPEs.

For all but the largest of the problem sizes that we considered, PFFTC achieves its best performance when the number of partitions is equal to the number of allocated SPEs. This strategy negates any benefits from double buffering throughout the algorithm, but reduces the total number of messages that must be sent across the bus throughout the algorithm, which is a critical bottleneck for small problems. For the larger problems, sending only a few large sub-problems across the bus would require the algorithm to wait for longer periods while the large sub-problems are distributed among the SPEs at the end of each stage. Also, there is more total computation time required for these problems, so there is a good opportunity to effectively mask some of the sub-problem communication with computation through double buffering. For these problems, we find that PFFTC attains its best performance by maximizing the number of sub-problems and distributing them across the maximum possible number of SPEs.

We note that although the blade server achieves its best performance using more than four SPEs for the 2,048 and 4,096 point problem sizes, the PlayStation 3 system still attains its best performance using only four SPEs for these problem sizes. This occurs because the PlayStation 3 has a maximum of only six SPEs available. If all six SPEs are allocated for these problem sizes the additional SPEs will add communication cost to the overall runtime because they will have to participate in each synchronization operation, but the full benefit of their computational power will not be applied to the problem because neither four, eight or sixteen sub-problems can be distributed evenly across six SPEs.

PFFTC has significantly higher performance than any prior Cell BE FFT package that we evaluated. FFTC reported the next best performance [1], although we were not able to reproduce these results during our testing. After FFTC, the Mercury FFT_ZIPX package achieved nearly the same performance as FFTC's reported results while using only a single SPE, but this package can only solve problem sizes up to 4,096 data points [8]. After the FFT_ZIPX package, the FFTW package had the slowest performance, which was not unexpected given FFTW's emphasis on portability instead of maximum platform-specific performance [6],[7]. We also found FFTC to attain performance similar to FFTW's reported results in our empirical testing. When we contacted the FFTC authors to investigate the performance of their algorithm, they indicated that their reported results were obtained on a pre-production blade server that is no longer available.

With respect to accuracy, we found that PFFTC's implementation solves FFT problems of all the sizes that we considered with only a small amount of error. A summary of the maximum inf-norm results obtained in our testing is shown in Table 4.

Table 4: PFFTC Accuracy Results.

Problem Size (points)	**Maximum Inf-norm Observed on PS3**	**Maximum Inf-norm Observed on Blade**
1,024	0.000066	0.000066
2,048	0.000185	0.000075
4,096	0.000380	0.000164
8,192	0.000168	0.000170
16,384	0.000399	0.000373

We believe that the error reported above occurs mainly because the Cell BE SPEs implement non-compliant floating point arithmetic that is less accurate even than the standard single precision floating point arithmetic that is available on most traditional computing platforms [4]. The exact amount of error reported for each problem size is dependent upon the partitioning and SPE count parameters that are used for each problem size, as these parameters affect the exact values computed for the twiddle factors in the solution stage and combination stage of the PFFTC algorithm, and also the numbers of butterfly stages computed using each of these sets of twiddle factors. However, we believe that the effect of the partitioning and SPE count parameters is small when compared to the total error due to basic floating point rounding errors. Altogether, the PFFTC algorithm reports total error that is very small when compared to the magnitude of its input values and the number of floating point operations that are being performed.

6 CONCLUSIONS AND FUTURE WORK

In summary, we present PFFTC, our high performance FFT implementation for the Cell BE. PFFTC has three main phases to (i) partition the problem into independent sub-problems, (ii) solve the sub-problems in parallel, and (iii) combine the results of the sub-problems to obtain the solution to the original problem. PFFTC includes optimizations for exploiting data transfer parallelism, avoiding unnecessary communication through careful data routing, avoiding data dependency stalls with instruction-level double buffering, and minimizing synchronization overhead through the use of an "asynchronous" signal-based barrier method. These optimization techniques can generally be applied to the optimization of any complex problem that is solved on the Cell BE.

Our PFFTC algorithm currently attains a peak performance of 33.6 GFLOPS for a problem size of 16,384 data points, and attains speedups ranging from 31% to 62% over the fastest previous Cell BE FFT algorithm's reported performance for FFTs with 1,024-16,384 data points [1]. PFFTC thus attains the lowest latency solution of any Cell BE FFT package of which we are currently aware. It also achieves this performance with high accuracy, which is limited only by the precision of the non-IEEE-754 compliant floating point unit present in the Cell BE's SPEs.

If even greater performance is needed, we believe that PFFTC's performance can be improved to as much as 40.3 GFLOPS by replacing our current solution stage FFT kernel with an improved kernel similar to the one used by the Mercury FFT_ZIPX package. In Appendix B, we present analysis modeling the performance of PFFTC and showing how this improved performance measurement is estimated.

Nevertheless, we note that the Cell BE faces a significant memory bottleneck when solving FFT problems in isolation from other operations. This bottleneck limits the Cell BE to just over 54% of its theoretical peak floating point performance for the largest FFT problem sizes that we considered, even assuming that the data transfer for solving multiple FFT problems in sequence can be arranged to fully utilize the main memory bandwidth and that the computation time for solving the FFTs can be reduced to balance exactly with the data transfer time for each problem. If the full potential of the Cell BE is to be realized for complex applications, it will be necessary to find ways to perform more useful computational work on the problem data once it has been brought from main memory to the SPEs than what would be possible by using the PFFTC algorithm as a library call to complete a single step in a larger application.

APPENDIX A: DISCUSSION OF CELL BE BENCHMARK RESULTS

Prior to developing the PFFTC algorithm, we performed a series of benchmark tests to evaluate the performance of various components of the Cell BE. These tests provided insight into the Cell BE architecture as a high-performance computing platform and helped to guide the design of the PFFTC algorithm. We present a brief discussion of some of our benchmark test results in this appendix. The results are grouped into three main areas: (i) basic peak processing performance, (ii) PPE memory hierarchy performance, and (iii) SPE-focused communication performance.

All of our benchmark tests were evaluated on the PlayStation 3 platform only, and were run in single-user mode under Fedora Core 6 (kernel 2.6.16). To reduce overhead from background processes, the ps3fbd process that is run by Linux to refresh the screen buffer on the PlayStation 3 was disabled during the timing for each benchmark. The ps3fbd process persistently consumes about 2% of the available PPE resources and a significant amount of the L2 cache, and thus could have a significant impact on the results of many of our benchmarks if it was not disabled.

We tested our benchmark codes with both the GCC version 4.1.1 compiler (Red Hat 4.1.1-30) and the IBM XLC version 8.2.0.19 compiler that ships with version 2.1 of the IBM Cell BE SDK to evaluate the efficiency of the code generated by each of these compilers. Each compiler was configured to use the –O3 optimization flag when generating benchmark executables.

Peak Processing Performance

We evaluated the peak processing performance of both the PPE and the SPEs using benchmark codes that each contain a simple loop of floating point operations. Each loop is manually unrolled to reduce loop control overhead, and the number of loop iterations is chosen to allow the benchmark to run long enough that the total test runtime is long compared to the tick length of the timer being used. The floating point operations are structured to operate on a small number of variables that fit within in the register file of the processing element being tested and are arranged to avoid any data dependency stalls to maximize the observed performance. All of these benchmarks were written to use SIMD instructions except the PPE double precision benchmarks, for which the PPE does not support SIMD operations.

The performance results obtained by our benchmarks for the floating point fused multiply-add and the basic floating point multiply operation in both single and double precision is shown in Figure 7.

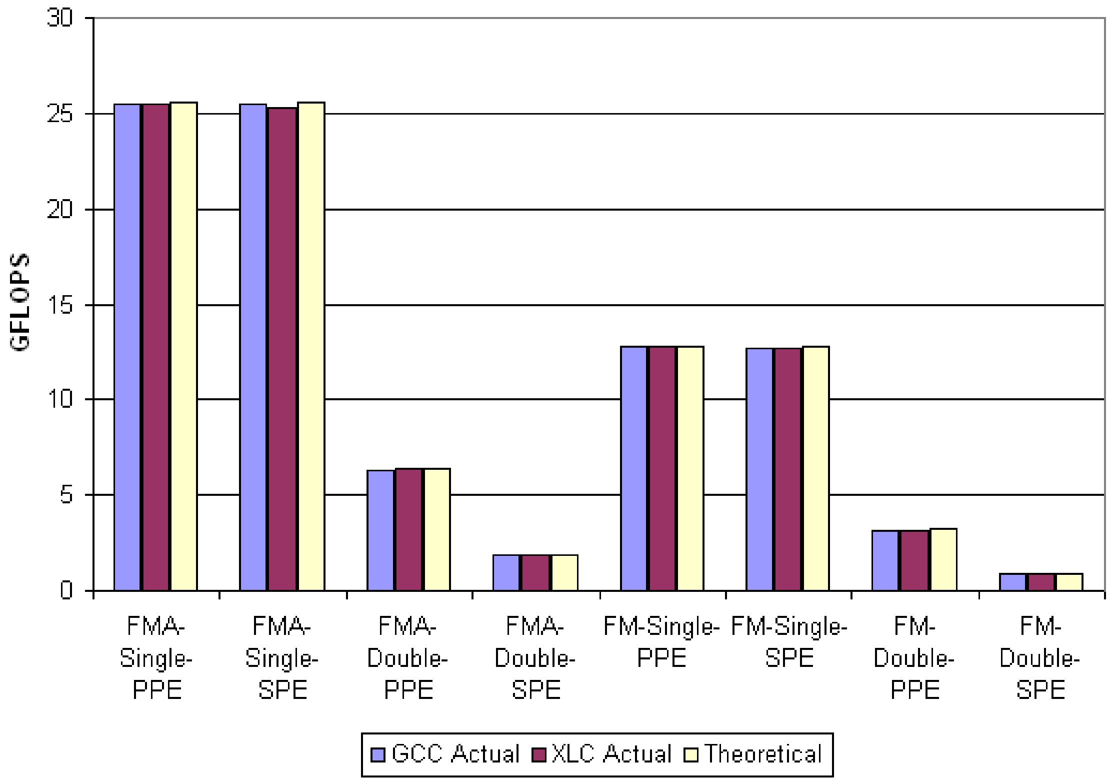

Figure 7: Peak GFLOPS for Basic Floating Point Operations.

As can be seen in Figure 7, GCC and XLC both generate code that attains nearly the full theoretical peak performance possible for both the fused multiply-add instruction and the basic multiply instruction, regardless of what processing element or floating point precision is used. Our results highlight the outstanding single-precision floating point performance of both the PPE and the SPE in the Cell BE, particularly for the fused multiply-add instruction. However, we note that this performance was achieved by hand-coding a program that is specifically designed to avoid memory access stalls. The Cell BE's SPEs are capable of sustaining this near-optimal performance on real world problems by carefully ordering instructions to avoid the 6-cycle latency delay of accessing the single-level SPE local store. However, the PPE's slower cache hierarchy prevents the PPE from attaining performance near its peak processing rate for anything larger than the most trivial of problems.

PPE Memory Hierarchy Performance

For problems that exceed the size of the PPE's register file, the performance of the PPE's cache hierarchy is a critical factor in the PPE's processing rate. We evaluated the impact of the PPE's memory hierarchy by writing a benchmark that performs 1,024 passes over an array, performing SIMD fused multiply-add instructions to update every element of the array in sequence using a manually unrolled loop on each pass. For arrays that are smaller than the L1 cache, the benchmark will only interact with the L1 cache after the first pass over the data is complete, allowing us to evaluate the L1 cache performance for these problem sizes. Because the PPE cache hierarchy uses a least-recently-used replacement policy and array elements are always accessed in sequence, arrays that are larger than the L1 cache but smaller than the L2 cache will cause an L1 cache miss but an L2 cache hit for every memory access, allowing us to evaluate the L2 cache performance for these problem sizes. Likewise, arrays that are larger than the L2 cache will cause an L2 cache miss for every memory access, and we can use these arrays to evaluate the performance of the main memory.

We ran tests for a variety of working set sizes between 1 KB and 1,024 KB using benchmarks generated using both GCC and XLC. For each compiler, we generated one benchmark program that defined the working set size as a constant defined in the benchmark code and ran only a single test for the specified working set size, and we also generated a second benchmark program that performed tests for all working set sizes within a loop and specified the working set size for each test using a program variable. This allowed us to evaluate how effectively each compiler makes use of constants defined in user code to improve its optimizations. Our results for these tests are shown in Figure 8.

As can be seen in Figure 8, the PPE's floating point performance decreases significantly once the working set exceeds the size of the 32 KB L1 data cache, and then again once the working set exceeds the size of the 512 KB L2 cache. Moreover, we note that even when the PPE is processing working sets that fit entirely within the L1 cache, its attains only about 8.8% of the theoretical peak of 25.6 GFLOPS that can be attained by the PPE or by a single SPE. The low efficiency of the PPE in this benchmark highlights the significant gap between the speed of the PPE's floating point unit and all levels of the PPE memory hierarchy. It also provides a strong motivation for using the PPE to coordinate the activities of the more efficient SPEs instead of attempting to use the PPE to perform serious computational work on its own.

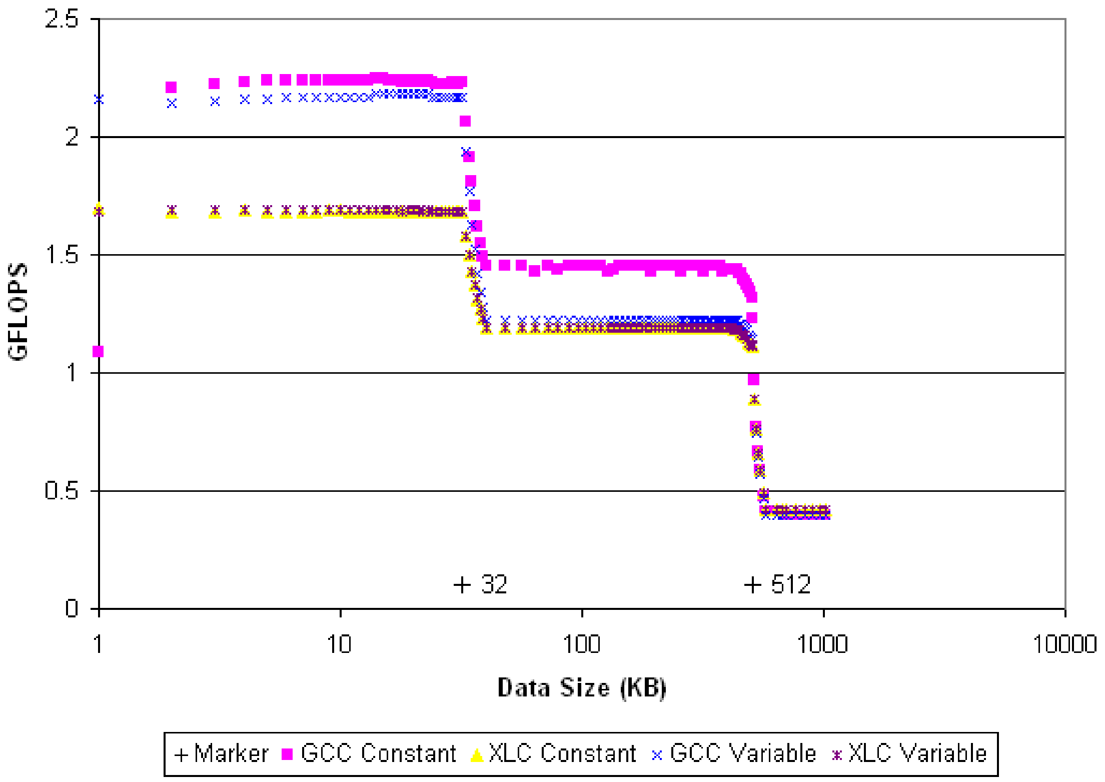

Figure 8: PPE GFLOPS vs. Working Set Size.

Figure 8 also shows that XLC generated benchmarks that have similar performance for all working set sizes regardless of whether the working set size is defined as a constant in the benchmark code or not, whereas GCC generated benchmarks that had better performance when the working set size is defined as a constant in the benchmark code. We believe that this is because GCC uses constants in program code as part of its optimization strategy, whereas XLC avoids these optimizations in favor of other more general optimizations.

Although Figure 8 shows GCC producing more efficient code than XLC for this simple memory hierarchy test, we found that in general XLC produces more efficient code than GCC for complex real-world applications. Also, we note that it is often easier to optimize programs that handle general input when they are compiled using the XLC compiler because the performance of XLC-compiled programs does not fluctuate depending on what constants are used in the program code.

SPE-Focused Communication Performance

Since the SPEs provide the bulk of the Cell BE's processing capabilities for real-world problems, understanding the performance characteristics of the Element Interconnect Bus that connects the SPEs to main memory and to each other is important for implementing algorithms efficiently on the Cell BE. We evaluated the latency of sending mailbox messages between the PPE and each SPE and between each pair of SPEs, the bandwidth of each single SPE for moving data to and from main memory and the PPE's L2 cache, the total bandwidth available for communicating with the main memory and the L2 cache, and the total bandwidth available for transferring data between each pair of SPEs. We also evaluated the impact of interleaving memory reads and writes on main memory bandwidth instead of grouping these operations together so that only read or write operations are being performed at any one time.

We first measured the latency of sending mailbox messages between the PPE and each SPE by writing a benchmark that allocates a single SPE and ping-pongs a mailbox message between the PPE and the allocated SPE repeatedly until the total test runtime is long compared to the timer tick. Although SDK version 2.1 does not support SPE affinity control on the PlayStation 3 platform, the SPEs appear to be allocated in round-robin fashion by the SDK support functions, with the allocation pattern continuing across program executions. This makes it possible to test the latency of sending messages between each individual SPE and the PPE by running our benchmark program repeatedly, because each subsequent test run will allocate and test the next SPE in the allocation sequence. We find that the both XLC and GCC produce benchmarks that have essentially identical performance for this test, and the average latency of sending a message between the PPE and an SPE is approximately 0.129 microseconds. However, each SPE actually has its own unique latency, which is likely due to the physical layout of the SPEs on the Cell BE chip. The pattern of SPE latencies in our test PlayStation 3 system is shown in Figure 9.

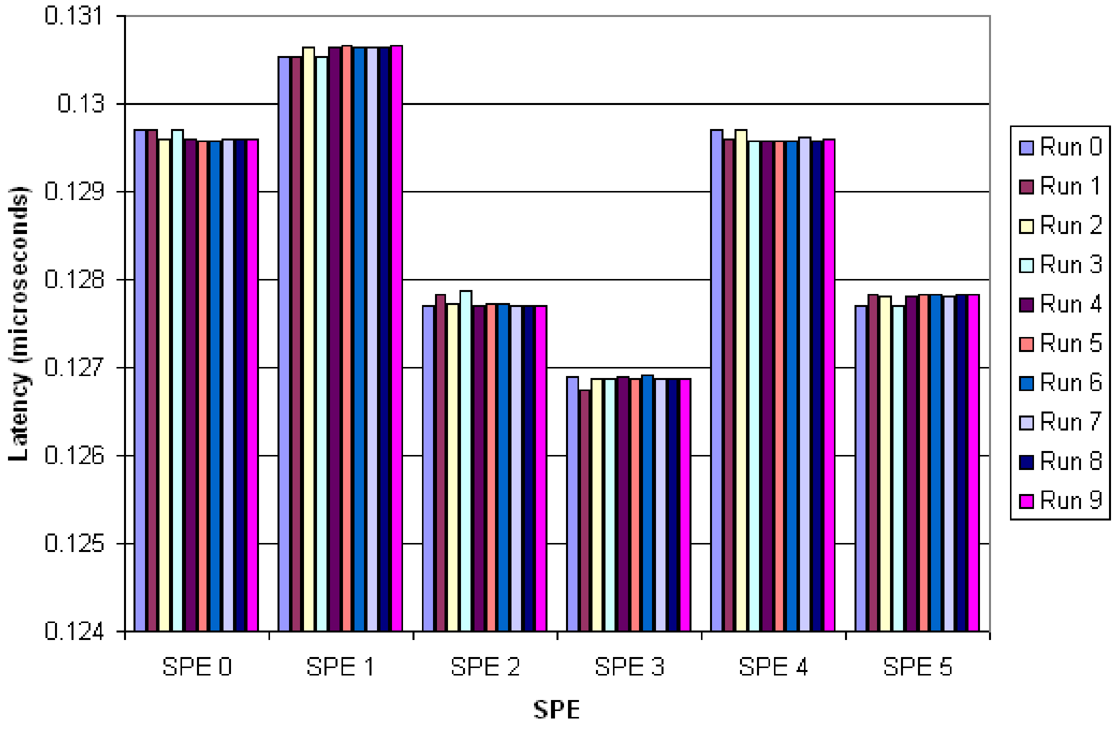

Figure 9: PPE-SPE Mailbox Message Latency.

Next, we extended our benchmark to test sending messages back and forth between all possible pairs of SPEs, with our expanded benchmark program testing each pair of SPEs one at a time in sequence. We found that the choice of compiler again made no difference in the timing results, and we found that the latency of sending messages between any pair of SPEs is approximately 0.064 microseconds. This delay appears to be constant regardless of which pair of SPEs are communicating, and is approximately half of the delay of sending messages between the PPE and any SPE.

After completing the mailbox message latency tests, we also tested the bandwidth available for sending larger messages back and forth between an individual SPE and the main memory and between an individual SPE and the PPE's L2 cache. The main memory benchmarks repeatedly performed a DMA get or put operation between each SPE and the main memory and timed how long the transfer of a known amount of data required, allowing us to compute the effective bandwidth. The L2 cache benchmarks performed the same test, but with the PPE scanning the region of memory being transferred before the timing loop to load it into the L2 cache before the transfers took place. For both sets of tests, we generated our benchmark programs using both the GCC and the XLC compiler, and we also varied the DMA block size used in the transfer.

Our single-SPE DMA transfer tests show that the first SPE allocated by the version 2.1 SDK after system boot-up is closer to the PPE and has higher bandwidth than the other five SPEs, all of which have the same bandwidth. Also, no single SPE is capable of using the full main memory or L2 cache bandwidth that is available in the system by itself. Further, our results show that XLC produces significantly more efficient communication control code than does GCC, as evidenced by the fact that the GCC-compiled benchmarks typically remain bottlenecked by control code for larger DMA block sizes than do the XLC-compiled benchmarks. A summary of the performance results for our single-SPE bandwidth tests is shown in Table 5.

Because a single SPE is not capable of utilizing the full main memory or L2 cache bandwidth by itself, we next extended our DMA bandwidth benchmarks to have all six

Table 5: Summary of Single-SPE DMA Benchmark Results.

Operation	SPE 0 Peak Bandwidth GB/sec	SPE 1-5 Peak Bandwidth GB/sec	Smallest DMA Block Size Attaining Peak Bandwidth for SPE 0 with GCC (KB)	Smallest DMA Block Size Attaining Peak Bandwidth for SPE 0 with XLC (KB)
MM Get	16.5	13.9	256	256
L2 Get	16.5	14.0	512	256
MM Put	23.7	22.2	512	128
L2 Put	23.7	22.2	512	128

SPEs in our PlayStation 3 test system repeatedly perform gets or puts simultaneously. Using this approach, we were able to compute the effective maximum bandwidth of both the main memory and the L2 cache as the total amount of data transferred by all SPEs divided by the total time elapsed during the test. We found that both compilers generated benchmarks that returned essentially identical results, as this test was communication bound because of the large number of SPEs simultaneously performing data transfers. The results of these bandwidth tests are reported in Table 6.

Table 6: Peak Bandwidth of Cell BE Main Memory and PPE L2 Cache.

Memory Subsystem	Get Bandwidth (GB/sec)	Put Bandwidth (GB/sec)
Main Memory	24.9	24.3
L2 Cache	19.5	24.3

We also evaluated the combined behavior of the main memory and the L2 cache by writing a new benchmark that allocates two sets of SPEs and then simultaneously performs data transfers between one set of SPEs and the L2 cache and between the other set of SPEs and the main memory. By varying the number of SPEs in each group and computing the effective bandwidth of the combined memory subsystems as the minimum per-SPE bandwidth observed multiplied by the number of SPEs in the test, we were able to obtain effective bandwidth rates higher than the peak bandwidth observed for either the main memory or the L2 cache alone for the DMA get operation. On the other hand, we found that the DMA put operation remained fixed at 24.3 GB/sec bandwidth regardless of how many SPEs were assigned to each set, suggesting that DMA put operations are written directly into main memory regardless of whether a corresponding memory entry is present in the L2 cache.

Working under the hypothesis that the main memory and the L2 cache can each serve DMA get requests independently, we were able to model the bandwidth reported by our benchmark by computing the minimum effective bandwidth available to any SPE multiplied by the number of SPEs participating in the test. The effective bandwidth of each SPE equals the bandwidth of the memory subsystem communicating with the SPE divided by the number of SPEs in the group of SPEs assigned to that memory subsystem. Basing our model on a 25 GB/sec effective main memory bandwidth and a 20 GB/sec effective L2 cache bandwidth and varying the memory subsystem group allocations among a set of six SPEs, we find that our model predicts the observed bandwidth of our benchmark fairly well. This finding confirms our hypothesis that the main memory and the L2 cache can indeed serve DMA get requests simultaneously. Our results are shown in Figure 10.

In addition to evaluating the SPEs' communication bandwidth with the main memory and the L2 cache, we also evaluated the bandwidth available between each pair of SPEs in our PlayStation 3 test system. We tested the bandwidth for each SPE getting data from every other single SPE in the system and for each SPE putting data into the local store of every other single SPE in the system. For all possible pairs of SPEs, we found the maximum effective bandwidth to be approximately 23.7 GB/sec. The compiler used to create the benchmark executables for these tests again did not have a significant impact on these results.

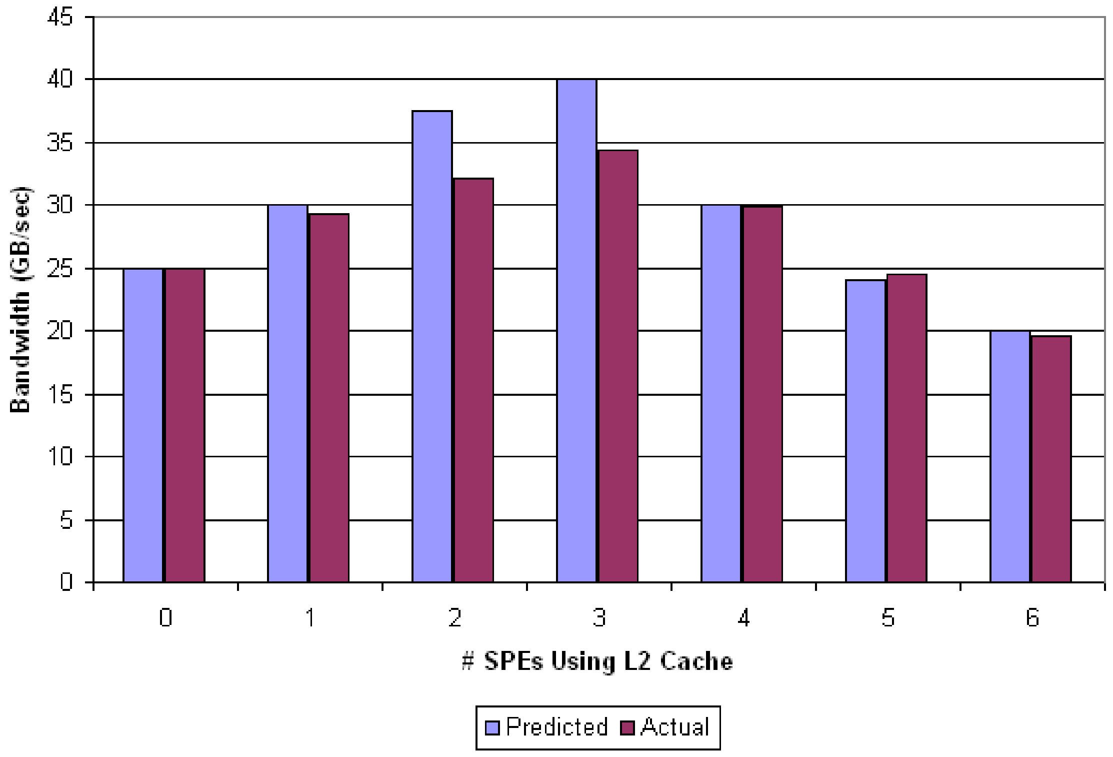

Figure 10: Predicted and Observed Bandwidth for Combined Main Memory and L2 Cache DMA Get Operations.

Finally, we wrote a series of benchmarks to evaluate the impact of SPEs performing interleaved DMA get and put operations when communicating with the main memory, with the L2 cache, with both the main memory and the L2 cache simultaneously, and with other SPEs. The main memory and L2 cache benchmarks for this test were written in the same way as the main memory and L2 cache bandwidth benchmarks discussed earlier, except that the DMA get and put operations are interleaved one after another instead of having each test perform only one type of operation throughout the entire benchmark. The SPE interleaved DMA benchmark was written similarly to the pairwise-SPE communication benchmark discussed earlier, except that the DMA operations are interleaved in this test as well. The bandwidth reported by these benchmarks is computed as the total data transferred divided by the total runtime of the test. As with our earlier tests, we evaluated our benchmark programs as compiled using both GCC and XLC.

The main memory, L2 cache, and combined main memory/L2 cache benchmarks all reported very similar results, indicating that approximately 21.3 GB/sec bandwidth could be sustained for the interleaved DMA transfer tests. The consistency of these results makes sense given our conclusion that DMA put operations are written directly into the main memory – hence, the L2 cache and combined main memory/L2 cache benchmarks would replicate the behavior of the benchmark for the main memory alone for all of the data transfers that they perform except the initial one. The reduction in effective main memory bandwidth that occurs when different DMA operations are interleaved suggests that it is desirable to block similar types of DMA operations together to maximize the available effective memory bandwidth whenever possible.

The SPE interleaved DMA benchmark test reported that every pair of SPEs continues to communicate with an effective bandwidth of 23.7 GB/sec even when different DMA operations are interleaved. This is favorable for application developers, as relieves them of the extra burden of grouping inter-SPE DMA operations together into blocks of similar operations to maintain high performance.

We conclude by emphasizing that the Cell BE is a powerful tool for high performance computing, but it is also a highly complex system that must be thoroughly understood if its full potential is to be exploited. The benchmarks discussed in this appendix provide a brief overview of the behavior of some of some key Cell BE subsystems. During the development of PFFTC, these benchmarks were used to guide the decisions to perform the partitioning stage on two SPEs, to avoid including the PPE as a processing element in the main PFFTC data flow, to avoid staging intermediate problem data in main memory, and to seek an alternative to the traditional tree-based barrier mechanism for inter-SPE synchronization. It is our hope that the results of these benchmarks will find further use in aiding the development of even more complex applications in the future.

APPENDIX B: ANALYSIS OF PREDICTED PFFTC PERFORMANCE USING IMPROVED SOLUTION STAGE KERNEL

The Mercury FFT_ZIPX package achieves near-optimal single-SPE FFT performance [8],[9]. In this appendix, we evaluate how much speedup could be attained by replacing the current PFFTC solution-stage FFT kernel with a more highly optimized kernel similar to the Mercury FFT_ZIPX package.

First, we note that to effectively compare the performance of the PFFTC and FFT_ZIPX kernels, we must separate the runtime of each kernel from the communication overhead of retrieving problem information and storing the result in main memory. To accomplish this, we use the SPE decrementer to time the evaluation of each FFT kernel for individual problems that are already present in the SPE local store. Because the SPEs have deterministic memory access and do not handle interrupts or exceptions, each individual FFT kernel call executes in essentially the exact same amount of time. We report the runtime and GFLOPS results for several typical sub-problem sizes in Table 7.

Table 7: Runtime and GFLOPS Performance of PFFTC Solution Stage Kernel and FFT_ZIPX Kernel for Typical Sub-Problem Sizes on Blade Server System.

Problem Size (points)	PFFTC Solution Stage Kernel Runtime (microseconds) / GFLOPS	FFT_ZIPX Runtime (microseconds) / GFLOPS
256	1.33 / 7.7	0.58 / 17.7
512	2.58 / 8.9	1.15 / 20.0
1,024	5.38 / 9.5	2.53 / 20.2
2,048	11.38 / 9.9	5.31 / 21.2
4,096	24.24 / 10.1	11.84 / 20.8

As Table 7 shows, FFT_ZIPX achieves at least twice the performance of our PFFTC solution stage kernel for each sub-problem size considered. This confirms that there is opportunity for significant speedup by further improving the performance of the PFFTC solution stage kernel.

To determine exactly how much speedup we could expect from such an optimization, we computed the reduction in PFFTC solution stage runtime that could be achieved for each problem size. This value equals the difference between the current PFFTC solution stage FFT kernel runtime and the FFT_ZIPX kernel runtime for the sub-problem size used by the optimal workload distribution parameters for each problem size, multiplied by the maximum number of sub-problems that must be solved on any SPE under the optimal workload distribution parameters for each problem size. The projected savings are reported in Table 8.

Table 8: Projected PFFTC Runtime Savings from Application of Iterative FFT Kernel Similar to FFT_ZIPX on Blade Server System.

Problem Size (points)	Optimal Sub-Problem Size	PFFTC Solution Stage Runtime per Sub-problem (microseconds)	FFT_ZIPX Runtime per Sub-problem (microseconds)	Maximum Sub-problems Solved on any SPE	Projected Savings (microseconds)
1,024	256	1.33	0.58	1	0.75
2,048	256	1.33	0.58	1	0.75
4,096	512	2.58	1.15	1	1.43
8,192	512	2.58	1.15	2	2.86
16,384	1024	5.38	2.53	2	5.70

However, this calculation assumes that the solution stage will remain compute-bound after the solution stage FFT kernel optimization has been applied. If the improved solution stage FFT kernel improves our algorithm enough so that it becomes communication-bound, then the communication delays in the solution stage will prevent all of these savings from being realized. We confirm that our algorithm will in fact remain compute-bound with an improved solution stage FFT kernel by verifying that the unmasked portion of the solution stage FFT kernel runtime is greater than the projected savings for each problem size. The necessary timing for this analysis was obtained by commenting out all of the calls to the solution stage FFT kernel function in our algorithm but leaving all of the communication intact and then re-running our algorithm for each problem size with the previously determined optimal workload distribution parameters. The difference between the runtime for the fully enabled PFFTC algorithm and our test version with the iterative FFT kernel disabled equals the amount of unmasked solution stage FFT kernel runtime. Table 9 shows the results indicating that the unmasked solution stage FFT kernel runtime is greater than the projected savings from the FFT kernel improvement optimization for each problem size considered.

Table 9: Unmasked PFFTC Solution Stage FFT Kernel Runtimes on Blade Server System.

Problem Size (points)	PFFTC Total Runtime (microseconds)	PFFTC Runtime w/out Solution Stage FFT Kernel (microseconds)	Unmasked Solution Stage FFT Kernel Runtime > Projected Runtime Savings (microseconds)
1,024	4.20	2.93	1.27 > 0.75
2,048	5.96	4.80	1.16 > 0.75
4,096	9.48	7.67	1.81 > 1.43
8,192	18.44	14.38	4.06 > 2.86
16,384	34.12	26.06	8.06 > 5.70

Finally, because our algorithm remains compute bound even with the improved solution stage FFT kernel, we can compute its new runtime for each problem size by subtracting the projected runtime savings for each problem size from the current optimal runtime for that problem size. The new projected runtimes and performance values are listed in Table 10.

Table 10: Projected Performance and Efficiency of PFFTC with Improved Solution Stage FFT Kernel on Blade Server System.

Problem Size (points)	Projected Improved PFFTC Runtime (microseconds)	Projected Improved PFFTC GFLOPS	Projected Improved PFFTC Efficiency
1,024	4.20 – 0.75 = 3.45	14.8	18.5%
2,048	5.96 – 0.75 = 5.21	21.6	24.6%
4,096	9.48 – 1.43 = 8.05	30.5	31.8%
8,192	18.44 – 2.86 = 15.58	34.2	32.9%
16,384	34.12 – 5.70 = 28.42	40.3	36.0%

BIBLIOGRAPHY

[1] Bader, D. and Agarwal, V. “FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine.” *College of Computing, Georgia Institute of Technology*, 2007. [Online]. Available: http://www.cc.gatech.edu/~bader/papers/FFTC-HiPC2007.pdf. [Accessed Feb. 25, 2009].

[2] Buttari**,** A., Luszczek, P., Kurzak, J., Dongarra, J., and Bosilca, G. “A Rough Guide to Scientific Computing on the PlayStation 3.” *Innovative Computing Laboratory, University of Tennessee Knoxville*, Tech. Rep. UT-CS-07-595, 2007. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/scop3.pdf. [Accessed: Feb. 25, 2009].

[3] Chen, T., Raghavan, R., Dale, J., and Iwata, E. “Cell Broadband Engine Architecture and its First Implementation,” *IBM Corporation*, 2005. [Online]. Available: http://www.ibm.com/developerworks/power/library/pa-cellperf. [Accessed: Feb. 25, 2009].

[4] IBM Corporation Technical Staff, *Cell Broadband Engine Programming Handbook v. 1.1*, IBM Corporation, 2007.

[5] IBM Corporation, “developerWorks: Cell Broadband Engine Resource Center,” *IBM Corporation*. [Online]. Available: http://www.ibm.com/developerworks/power/cell. [Accessed: Feb. 25, 2009].

[6] Frigo, M. and Johnson, S., “The Design and Implementation of FFTW3,” *Proceedings of the IEEE 93 (2), 216-231 (2005). Invited paper, Special Issue on Program Generation, Optimization and Platform Adaptation*. [Online]. Available: http://www.fftw.org/fftw-paper-ieee.pdf. [Accessed: Feb 25, 2009].

[7] FFTW.org, “Fastest Fourier Transform in the West,” *FFTW.org*. [Online]. Available: http://www.fftw.org/cell. [Accessed: Feb. 25, 2009].

[8] Mercury Computer Systems, Inc., “Algorithm Performance on the Cell Broadband Engine Processor, Revision 1.1.2,” *Mercury Computer Systems, Inc.*, 2006. [Online]. Available: http://www.mc.com/uploadedFiles/Cell-Perf-Simple.pdf. [Accessed: Feb. 25, 2009].

[9] Cico, L., Cooper, R., and Greene, J., “Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor,” Mercury Computer Systems, Inc., 2006. [Online]. Available: http://www.mc.com/uploadedFiles/CellPerfAndProg-3Nov06.pdf. [Accessed: Feb. 25, 2009].