RC24244 (W0704-106) April 18, 2007
Computer Science

IBM Research Report

Accelerating FFT Performance Using the Cell BE Processor

Jizhu Lu, Acie Nobles*, Michael Perrone
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*IBM Sales & Distribution

Accelerating FFT Performance Using the Cell BE Processor

Jizhu Lu, Acie Nobles and Michael Perrone, IBM Corporation*

Summary

The Fast Fourier Transform (FFT) is a key computational workload for seismic imaging. Acceleration of this workload can significantly reduce the run-time of seismic imaging applications. In this abstract, we describe methods for accelerating single-precision, two-dimensional, complex-to-complex, FFT calculations using the Cell Broadband Engine (BE) processor™. In particular, we focus on the Prime Factor Algorithm (PFA) FFT because of its flexibility and efficiency in handling a variety of FFT sizes. Our results demonstrate 12x to 18x improvements over state-of-the-art solutions on dual-core, Intel Woodcrest and AMD Opteron processors.

Introduction

Fast Fourier Transforms (FFT's) are important to numerous engineering and scientific applications, including seismic image processing where 2D FFT's are widely used. Of the many FFT algorithmic that exist, one generally finds a trade off between flexibility and efficiency. For example, extremely efficient algorithms are known FFT's that are powers of two in length; however, one must be willing to constrain oneself to sizes like 1024 and 2048 or pad one's data to the next power of two and the thereby lower the algorithm's efficiency. As a compromise, some practitioners rely on algorithms which might be less efficient in particular cases but which provide better coverage of allowed FFT sizes.

One such algorithm is the Prime Factor Algorithm (PFA) developed by Temperton (1985) which allows efficient computation of FFT's whose size is equal to a multiple of a fixed set of "prime" factors.[1] A scalar implementation of this algorithm can be found at the Colorado School of Mines Seismic Un*x website. For this paper, we have chosen to implement the PFA FFT on the Cell BE Processor to evaluate the potential performance improvements from the processor's eight parallel compute cores, data management flexibility and native vector registers. Our implementation is complex-to-complex, single-precision, two-dimensional and includes ten "prime" factors: 2, 3, 4, 5, 7, 8, 9, 11, 13 and 16.

In the following sections, we describe the relevant aspects of the Cell BE Processor; motivate the need for vectorization of the data; describe the vectorization of the code; review the algorithms that were implemented; discuss the experimental results and summarize with conclusions.

[1] Although called "prime" the factors implemented in any particular case are actually co-prime; so 4 and 16 would never occur together but 4 and 9 can.

Cell BE Processor Overview

The Cell BE Processor™ was designed for the Sony PlayStation3™ and as such is ideally suited for the highly parallelizable, vectorizable, single-precision calculations found in computer gaming. For the same reasons, it is also well suited for general image processing. Here we only highlight the aspects of processor that are essential to the algorithm implementation issues in this paper.[2]

The processor has nine cores: Eight used as compute cores while the ninth runs the OS, spawns compute threads and handles some synchronization. Each of the compute cores runs autonomously once its thread has been spawned which allows us to take full advantage of the highly parallelizable nature of the PFA FFT. Each of the compute cores is a "native" vector processor in that each of its 128 general purpose registers can operate on 4 single-precision floating point numbers simultaneously. Thus vector operations are preferred since scalar operations will be commensurately slower. Each compute core has its own dedicated Memory Flow Controller (MFC) which enables Direct Memory Access (DMA) transfers (puts & gets) of data between coherent, shared main memory and its dedicated 256KB Local Store (LS). As a compute core processes data, it uses the MFC's DMA feature to move data to its LS for processing. These transfers can be done concurrently with computation. For efficient performance, algorithms are generally parallelized over multiple compute cores, vectorized for the vector registers and multibuffered to hide the latency of moving data between main memory and the eight LS's.

Vectorization of the Data

We say that the MxN 2D data is in "scalar" format if the first M entries in the matrix correspond to one trace of length M; the next M correspond to the next trace and so on for N traces. In this case, when the data is loaded into a vector register, it will have 4 consecutive values from one trace. We say that the data is in "vector" format, or is "vectorized", if 4 distinct traces have been interleaved in memory. In this case, when data is loaded into a vector register, each float element of the register will come from a

[2] Detailed Cell BE Processor information can be found at: http://www-128.ibm.com/developerworks/power/cell/index.html

different trace but each will have the same index in the trace it came from (e.g., a register might contain the 17th element from traces 0, 1, 2 and 3). The transformation between scalar and vector formats is depicted in Figure 1.

The motivation for this transform is that it simplifies virtually all subsequent computation and much of the memory access patterns. This simplification results in dramatic performance improvements. However additional performance potential is left unrealized if one has to pay the cost of the initial vectorization and subsequent scalarization every time the data is visited. If on the other hand the entire data is vectorized once at the beginning of the seismic imaging process and all subsequent operations (FFT and otherwise) are done in vector format, then the transform overhead can be amortized and neglected. This one-time-transfer approach will have benefits beyond this FFT implantation as all vectorizable portions of the code will benefit from this approach while scalar portion will not be significantly impacted. In our results, we compare the vectorized PFA FFT performance with and without the repeated data vectorization and scalarization transforms, to measure the potential value of adopting a vectorized data format throughout the seismic processing pipeline.

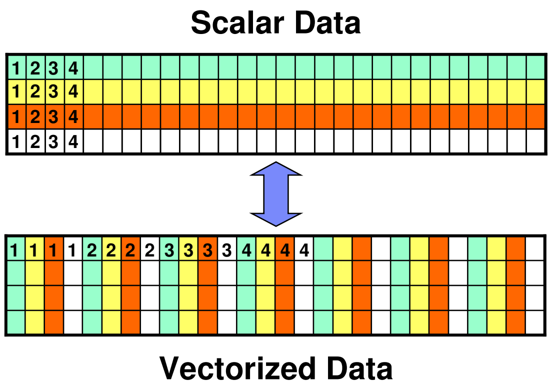

Figure 1: Vectorization/Scalarization transforms for 4 traces of length 28. Memory stride-one proceeds to the right.

Vectorization of the Code

With the data in the vector format, we now have to "vectorize" the code. Here is sample of scalar PFA code for prime factor 3 in which z[] contains the data:

```
for (l=0; l<m; l++) {
    t1r = z[j01]+z[j2];
    t1i = z[j01+1]+z[j2+1];
    y1r = z[j00]-0.5*t1r;
    y1i = z[j00+1]-0.5*t1i;
    y2r = c1*(z[j01]-z[j2]);
    y2i = c1*(z[j01+1]-z[j2+1]);
    z[j00] = z[j00]+t1r;
    z[j00+1] = z[j00+1]+t1i;
    z[j01] = y1r-y2i;
    z[j01+1] = y1i+y2r;
    z[j2] = y1r+y2i;
    z[j2+1] = y1i-y2r;
    jt = j2+2;
    j2 = j01+2;
    j01 = j00+2;
    j00 = jt;
}
```

Using the vectorized data, this loop can be modified to handle 4 traces at a time as follows:

```
for (l=0; l<m; l++) {
  t1r = spu_add(z[j01],z[j2]);
  t1i = spu_add(z[j01+1],z[j2+1]);
  y1r = spu_sub(z[j00],spu_mul(P5,t1r));
  y1i = spu_sub(z[j00+1],spu_mul(P5,t1i));
  y2r = spu_mul(c1,spu_sub(z[j01],z[j2]));
  y2i = spu_mul(c1,spu_sub(z[j01+1],z[j2+1]));
  z[j00]   = spu_add(z[j00],t1r);
  z[j00+1] = spu_add(z[j00+1],t1i);
  z[j01]   = spu_sub(y1r,y2i);
  z[j01+1] = spu_add(y1i,y2r);
  z[j2]    = spu_add(y1r,y2i);
  z[j2+1]  = spu_sub(y1i,y2r);

  iwrk01 = spu_insert(j01,iwrk01,0);
  iwrk01 = spu_insert(j00,iwrk01,1);
  iwrk01 = spu_insert(j2,iwrk01,2);
  iwrk01 = spu_add(iwrk01,itwo);
  j2  = spu_extract(iwrk01,0);
  j01 = spu_extract(iwrk01,1);
  j00 = spu_extract(iwrk01,2);
}
```

If the loop variable is not a multiple of four, then code has to be added to handle the edge condition, similarly for the j01 and j2 additional data alignment might be needed.

Algorithm Implementations

A 2D FFT can be decomposed into two sets of 1D FFT's - one for each dimension. In this approach, 1D FFT's are performed for all traces in one dimension, the data is transposed, the 1D FFT's are repeated on the other dimension and then a final transpose is done to restore the data to its original orientation. We used the same approach here with additional considerations to take advantage of the Cell processor's strengths. This approach led to two implementations of the PFA FFT based on the number of image transfer steps from main memory to LS and back.

The 3-Step Algorithm is as follows:

1. First DMA Step: Transfer 4 traces to LS buffer; vectorize buffer; run vectorized PFA FFT on buffer; transpose each 4x4 "tile" in buffer and transfer to main memory; repeat until all image data have been processed.
2. Second DMA Step: Transfer 1 buffer from main memory to LS; run vectorized PFA FFT on buffer; transpose buffer and transfer to main memory; repeat until all image buffers have been processed.
3. Third DMA Step: Transfer 1 buffer from main memory to LS; scalarize; transfer to main memory; repeat until all image buffers have been processed.

In Figure 2 below, we depict the data layout in main memory before and after the second step of the 3-Step algorithm. The 4x4 tile size allows us to efficiently write the data so that it has the correct memory position for the next step.

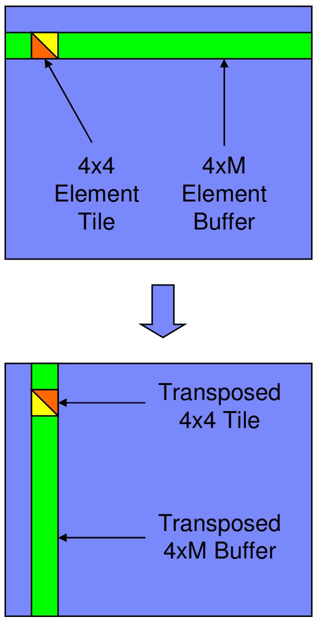

Figure 2: High-level view of data flow in algorithm.

The 2-Step Algorithm combines the second and third steps of the 3-Step algorithm to reduce the number of transits of the image data to and from main memory:

1. The first DMA Step: Transfer 4 traces to LS buffer; vectorize buffer; run vectorized PFA FFT on buffer; transpose each 4x4 block in buffer and transfer to main memory; repeat until all image data have been processed.
2. The second DMA Step: Transfer 2 buffers from main memory to LS; run vectorized PFA FFT on both buffers; transpose, scalarize and transfer data to main memory; repeat until all image buffers have been processed.

For both of these algorithms it is implicit that the steps are parallelized over multiple compute cores where each core is processing a separate subset set of image. This requires synchronization at the end of each 1D FFT process so that we are guaranteed that the FFT is complete before starting the FFT of the transposed data.

Note that because the 2-Step algorithm moves less data, it requires less bandwidth but because it merges the transpose and scalarization operations, it benefits less from removing the data vectorization and scalarization transforms. This effect can be seen in our performance results below.

Experimental Results

The 2-Step and 3-Step algorithms described above were optimized by hand and run on a 3.2 GHz Cell BE blade with two Cell BE chips, 1GB RAM and 18x16MB huge memory pages. Each Cell BE chip ran on its own set of image data 20480 times for a total of 40960, 2D FFT's performed for each image size. Each algorithm was also run with the corresponding data vectorization and scalarization transforms removed to test the performance when the data is already in vectorized format. In the following we refer to these four cases as 2-Step with data vectorization, 2-Step without, 3-Step with and 3-Step without (or 2SW, 2SWO, 3SW and 3SWO, respectively). The images sizes were chosen to be representative of those commonly used seismic imaging.

In order to enable a blade-to-blade performance comparison, we ran 2D, complex-to-complex, single-precision FFT's on Intel and AMD blades. These runs used dual-core, dual-socket Woodcrest and Opteron blades running at 3.0 GHz and 2.4 GHz, respectively. Each of the four cores on these blades ran on its own set of image data 10240 times for a total of 40960, 2D FFT's performed for each image size. In order to make these comparisons as fair as possible, we used the optimized math libraries provided by Intel and AMD for comparison: On the Woodcrest blade, we used the FFT implementation in Intel's Math Kernel Library (MKL) and for the Opteron blade we used the FFT implementation in AMD's Core Math Library (ACML).

Figure 3 compares the execution time in seconds for a variety of 2D image sizes among the four algorithms described in this paper running on Cell blades and the 2D FFT algorithms running on Woodcrest and Opteron. From Figure 3, we can see that the Cell BE processor is significantly faster then either the Intel Woodcrest or the

AMD Opteron in a blade-to-blade comparison.[3] The 3-Step algorithm without repeated data vectorization transforms (3SWO) is the best overall but if a complete vectorization of all data is not possible, then the 2-Step algorithm (2SW) is the best overall performer. The 2SW case is about 7% faster than the 3SW case.

Figures 4 and 5 compare the speedup of the Cell BE algorithms described in this paper relative to the AMD Opteron and Intel Woodcrest FFT performance, respectively. From these figures, we can see that the PFA FFT on Cell can outperforms the ACML FFT on dual-core Opteron by a speedup of over **14x** in the **2SW** case and by nearly **18x** in the **3SWO** case; and relative to the MKL FFT on dual-core Woodcrest, we see that the PFA FFT on Cell can outperforms by a speedup of nearly **9x** in the **2SW** case and over **12x** in the **3SWO** case. Note that in Figure 4, the speedup is fairly constant over a wide range of image sizes while in Figure 5 we see the speedup increasing with image size particularly for the 3SW and 3SWO cases. We anticipate this trend will continue until for larger images sizes. This will be the focus of other studies.

Conclusions

In this paper we have presented four algorithms for accelerating 2D, complex-to-complex, single-precision FFT calculations on the Cell BE Processor. We have described the various implementation issues and have compared the performance of these algorithms with state-of-the-art FFT implementations from Intel and AMD running on dual-core, dual-socket Woodcrest and Opteron blades, respectively. Our comparisons demonstrate a significant opportunity for accelerating the algorithms that are of key importance to this conference's audience.

We believe that as multicore processor design advances, it will become increasingly important that researchers pay close attention to the special algorithmic requirements in order to obtain highly-efficient code on such processors. We hope that this paper will motivate other researchers to explore implementing additional algorithms on the Cell BE Processor and that the lessons learned here will be useful in those explorations.

Acknowledgments

The authors thank Billy Robinson of the IBM Corporation for his assistance in generating the performance comparison timings for the Woodcrest and Opteron processors.

[3] It should be noted that this advantage persists at the chip-to-chip level but for size reasons those results are not included here.

Execution Time Performance Comparison

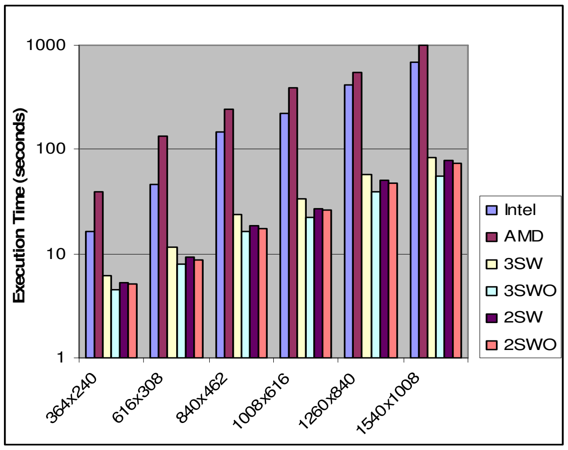

Figure 3

Cell Performance Speedup over AMD Opteron

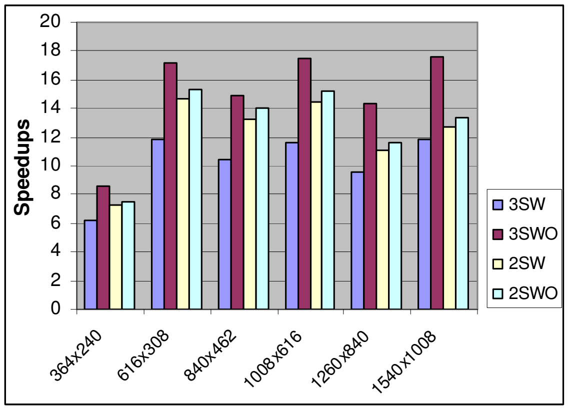

Figure 4

Cell Performance Speedup over Intel Woodcrest

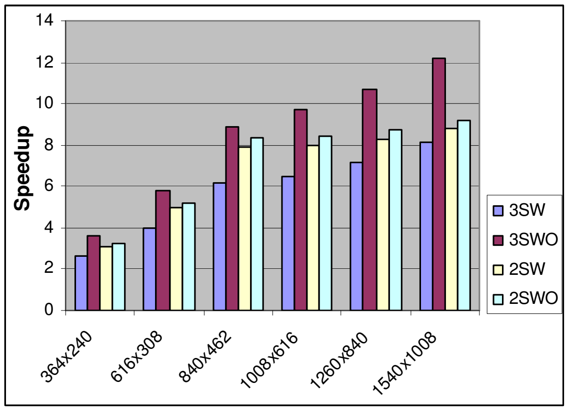

Figure 5