Accelerated DSP Functions on the CBE for the X-Midas Toolkit

Joseph A. Jezak[1], Charles Berdanier[2], Steven P. Levitan[3] and Donald M. Chiarulli[4]
[1]*Computer Engineering Graduate Program, University of Pittsburgh*
[2]*Department of Electrical Engineering, Wright State University*
[3]*Department of Electrical and Computer Engineering, University of Pittsburgh,*
[4]*Department of Computer Science, University of Pittsburgh*

***Abstract* - We characterize the impact of the Cell Broadband Engine architecture, on commonly used radar DSP algorithms. We use the capabilities of the CBE to accelerate several key computational kernels including Matrix Multiplication, Matrix Inversion, and the Finite Impulse Response (FIR) filter. These algorithms are implemented and benchmarked as library routines within the X-Midas Toolkit. We observe speedups as large as 1200x for complex matrix multiplication, but speedups of 40x to 60x are more typical. We find that system I/O overhead within the X-Midas toolkit severely limited the performance of the applications.**

I. Introduction

The Cell Broadband Engine (CBE) is a heterogeneous chip-multi-processor designed to be the main processor for the Sony Playstation 3 video game console [1, 2]. Each CBE chip integrates a 64-bit PowerPC core (PPU) with eight 128-bit vector-processing units (SPUs). Each SPU provides a four way, single instruction, multiple data stream (SIMD) processor with a small local memory and two instruction pipelines, one for register load/stores, and another for 4-way vectorized arithmetic and logic. A memory flow controller (MFC) in each SPU supports parallel block data transfers under software control on a multichannel ring bus between the main memory and the SPU local store. Running at a clock rate of 3.2GHz, the CBE has a theoretical maximum performance of 256 GFLOPS.

X-Midas is a software toolbox widely used by DoD and other groups for modeling and prototyping DSP applications [3]. The X-Midas interface is based on a scripting language, similar to the GNU Octave and Matlab interfaces [4, 5]. The scripting language is based on keyword commands that allow the execution of sequences of library applications or built in commands called "intrinsics". Unlike other mathematical programming platforms, it is designed specifically for DSP application prototyping. It includes many features intended for signal analysis, including file formats with implicit time information, data organization for windowing, and many other useful built in routines.

The goal of this project was to measure the impact of the CBE architecture on specific DSP algorithms and on the overall performance of radar applications benchmarked in the X-Midas Toolkit. In this paper, we report on the integration of Cell Broadband Engine accelerated Matrix Multiply, Matrix Inversion, and Convolution/FIR Filter functions to the X-Midas DSP Toolkit.

This research was partially supported by USAF/AFRL under MYRAIAD task order #520-10-10

II. CBE Optimized Algorithms

Optimized parallel software on the CBE must be written to explicitly incorporate low level hardware specific optimizations with fundamental parallel programming models. Examples of these hardware specific optimizations include (1) software controlled *scheduling and memory flow control* between the SPU and the local store, (2) *memory data organization* to optimize vector computations of complex arrays, and (3) *instruction pipeline optimization* to balance execution of computation with register management. Each of these techniques is useful for optimizing signal processing, as well as other parallel applications on the CBE. The accelerated functions that we implemented, as described below, provide specific examples of all three of these optimizations.

A. Block Matrix Multiply

In the block matrix multiply algorithm each matrix is partitioned into blocks and each result block is computed independently and in parallel. The diagram in Figure 1 shows a model of the block matrix multiply algorithm for a 4 by 4 matrix, in which sets of block computations are assigned to two separate SPUs. Here SPU 0 computes R1, R2, R5, and R6 while SPU 1 computes R11, R12, R15, and R16.

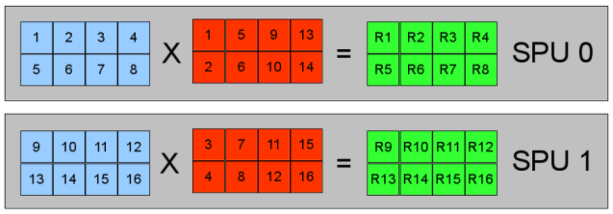

Figure 1. Matrix Multiply Block Assignment

The algorithm is easily *memory flow optimized*. By dividing the matrix into blocks of size N, each computational block dispatched to an SPU requires data transfers on the order of 3N, one block (N) for matrix A, one block (N) for matrix B, and the result block. Computationally, each block costs on the order of $O(N^2)$ floating point operations. Since the computation time rapidly outweighs the transfer time, it is relatively easy to hide all of the latency associated with data transfers by doing them in parallel with other computations. This interval also allows for a transposition of each block to be done on the PPU and similarly hidden behind the parallel computation time.

As an additional problem, matrix B must be transposed so that the data elements can be operated on with vector operations. In Figure 2, we show the calculation of result element R1 with the transposed matrix. As can be seen by transposing the matrix the elements in matrix B required to calculate the result, are now located adjacent to each other, which allows for the use of vector operations. This *memory data organization* allows optimum use of the vector units. The final result for this element is found by summing the elements

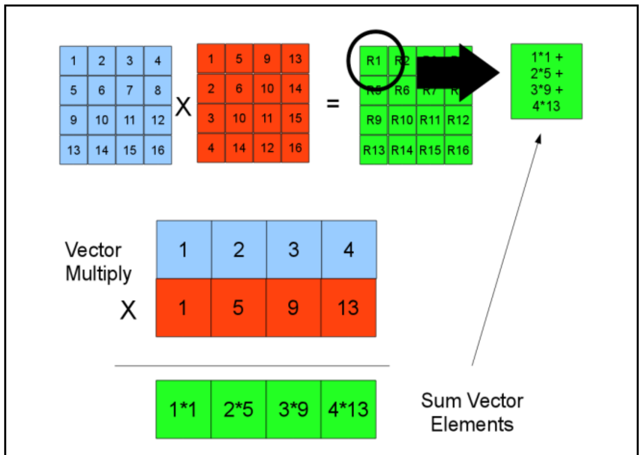

Figure 2. Block Matrix Multiply Vector Operations

of the result vector.

The transpose operation is performed at the block level on the PPU before the DMA transfer occurs and is overlapped with the computation time spent in the SPUs using the *memory flow control* optimization model. In each SPU, on each sequence cycle, a block is computed while the next block is transferred in and the last block is transferred back to the main memory, an implementation of the triple buffering is shown in Figure 3. During the computation phase, the elements of a block from matrix A are multiplied by transposed elements from a block in matrix B. The resulting vectors are then added to find the final result block. Thus, in each SPU, on each sequence cycle, a block is computed in parallel with an incoming and an outgoing DMA transfer. This allows the SPU vector processor to stay continually active without wait time for incoming or outgoing data transfers.

To further optimize the instruction flow, complex arrays were arranged into separate contiguous blocks containing the real and imaginary parts respectively. Using this format rather than the conventional interleaved organization. Vectorized complex MADD operations were completed in four instructions without any intervening instructions necessary to reorganize the vectors. Moreover, by unrolling most of the inner loops and hand *optimizing the computational and load/store pipelines*, we were able to implement a block matrix multiply algorithm with a computational rate that was very near the theoretic maximum for the CBE. Outside of X-Midas we observe a speedup of 3400x [10].

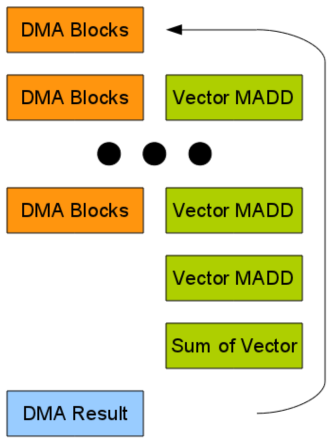

Figure 3. Pipelined DMA/Computation

B. Matrix Inversion

The Matrix Inversion algorithm implemented for this project is a variation of the Gauss-Jordan-Floyd-Warshall method originally published for the CBE by Yokoyama and diagramed in Figure 4 where a matrix is decomposed into four blocks that can be computed independently [6].

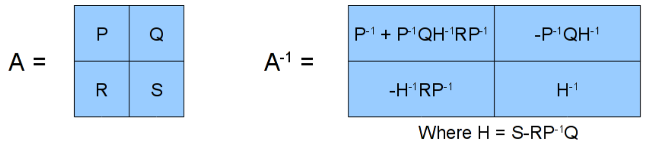

Figure 4. Gauss-Jordan-Floyd-Warshall Method

In the CBE implementation of this algorithm, the matrix is broken into blocks that can be computed in parallel. As shown in Figure 5, these blocks are referred to by color, indicating which stage of the algorithm is being performed on each block during each iteration of the algorithm. In each iteration, a new black block positioned along the diagonal is inverted on the PPU using LAPACK routines optimized for the PPE [7]. Next, the red blocks are computed in parallel on the SPUs as a matrix multiply with the current black block. The next step is to complete the white blocks, also in parallel on the SPUs, as the product of the red block and blue block for corresponding row and column. Finally, the blue blocks are computed in parallel as the product of each blue block with the current black block.

The key to optimization of this algorithm is the *scheduling and memory flow control* of the parallel computations at each

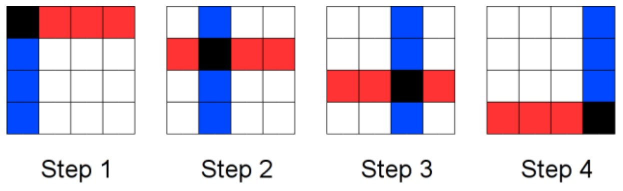

Figure 5. Matrix Inversion Blocking Demonstration on a 4x4 Block Matrix

stage. Synchronization barriers are required for iteration and between the inner loops of red, white, and blue block computations.

C. Convolution/FIR Filter

The FIR algorithm is a simple convolution between a fixed set of taps and an input data stream. In our CBE implementation, the taps are split among the available SPUs, and padded with zeros so that each SPU has an equal number of taps. The computation proceeds as a pipeline executing in parallel across the SPUs. As with our previous algorithms, *software memory flow* control and triple buffering hides all memory latency behind the parallel computations.

As with the matrix multiply, real and complex parts of each value are stored in separate buffers to optimize the complex MADD operation. However, we also optimize data flow, of shifted values as each of the four lanes in a 128-bit vector are convolved against each other. This *instruction pipeline optimization* is solved by allocating four copies of the tap buffers, each shifted by 32-bits relative to the other as shown

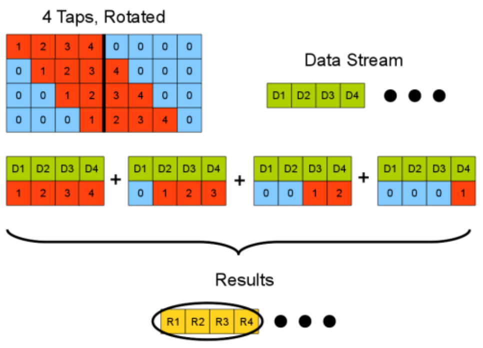

Figure 8. Example of the buffering and convolution operations of 4 taps, buffered and shifted for complete convolution.

in Figure 6. Thus, in four vector MADD operations, the four lanes of the 128-bit vector are convolved without any need for intervening shift or shuffle operations, and thus keeping the processor maximally active.

The FIR operation also required a special optimization of the *memory flow control* programming. In order to ensure that

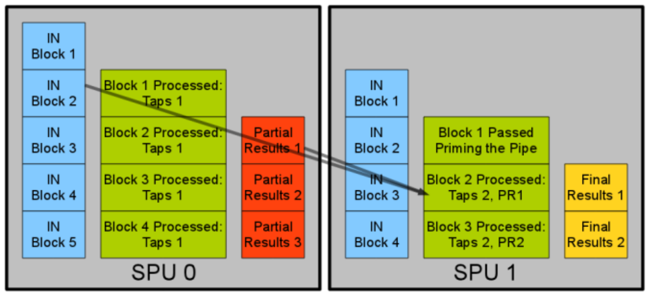
IN Block 1
IN Block 2
IN Block 3
IN Block 4
IN Block 5
Block 1 Processed: Taps 1
Block 2 Processed: Taps 1
Block 3 Processed: Taps 1
Block 4 Processed: Taps 1
Partial Results 1
Partial Results 2
Partial Results 3
SPU 0
IN Block 1
IN Block 2
IN Block 3
IN Block 4
Block 1 Passed Priming the Pipe
Block 2 Processed: Taps 2, PR1
Block 3 Processed: Taps 2, PR2
Final Results 1
Final Results 2
SPU 1

Figure 6. Data movement between two pipeline stages of the accelerated FIR implementation

each result is convolved with every input in the current set, the computational pipeline moves at two speeds simultaneously. As shown in figure 7, input data is double buffered in input and computational buffers, results are triple buffered across, input, computational, and result buffers. Thus, it takes 2 cycles for data to pass through an individual SPU and 3 cycles for results, thus ensuring that each pipeline stage in each SPU is computing a unique result.

III. Benchmark Results

Benchmarks were performed on an IBM cell Blade Model JS20 with 1GB of RDRAM memory shared by two CBE chips. Each chip has one PPU and eight SPU processors running at 3.2GHz. The processors are served by a 40GB IDE disk and are running Redhat Linux.

For the 2D Matrix functions, *Block Multiply* and *Invert,* the accelerated functions were benchmarked against single threaded X-Midas host functions written for comparison, while the Convolution/FIR filter was benchmarked against the CONVOLVE X-Midas function. Benchmark timing is based on wall clock time recorded using the X-Midas *TIMER* intrinsic. Thus, this data includes the X-Midas overhead for parsing the script, starting the application or built in intrinsic and for X-Midas memory management.

A. Block Matrix Multiply

The performance of the Block Matrix Multiply application as an X-Midas intrinsic is shown in Figure 8. As reported in

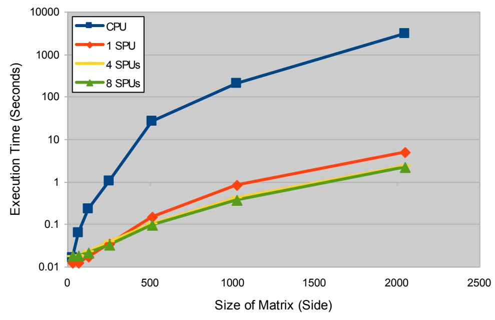

Figure 9. X-Midas Real Matrix Multiply Execution Time

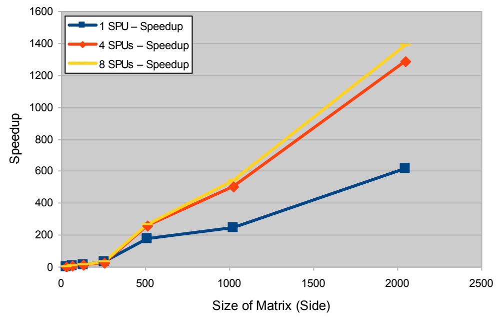

Figure 7. Speedup data for X-Midas implementation of real valued Matrix Multiplication over various matrix sizes

[10] standalone speedups for matrix multiply are quite significant, reaching near theoretical peak performance and showing more than three orders of magnitude improvement for the 8 and 16 SPU cases. As shown in Figure 9, this accelerated code when run within X-Midas loses some of this performance as an X-Midas intrinsic measured by our benchmark, even for relatively large matrices. Moreover, memory management overhead within X-Midas appears to grow faster than computation time for this application in X-Midas for the 1GB primary memory available on the Cell Blade used for testing. This made it impossible to test large matrices where better speedups might have been expected.

B. Matrix Inversion

The additional computational load and smaller speedup of matrix inversion makes it far less sensitive to X-Midas overhead. Figure 10 is a plot the execution time for inverting complex valued matrices of sizes of 32x32 to 1024x1024. The blue trace is runtime for the single threaded PPU implementation and the red, yellow, and green traces are the runtimes of 1, 4, and 8 SPU versions respectively.

In Figure 11, the data from the previous figure is used to find the speedup of the accelerated versions versus the PPU implementation. The largest matrix tested 1024x1024 showed a 45x speedup and a smaller, 128x128 matrix, showed a respectable 5x speedup.

C. Convolution/FIR Filter

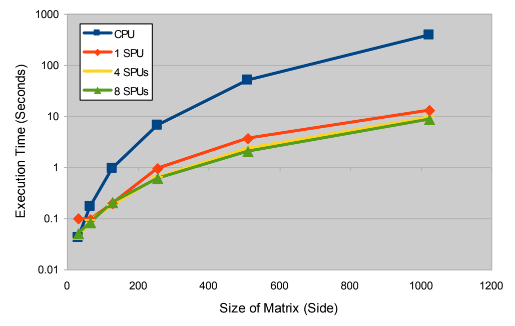

Figure 10. Execution Time for Complex Valued Matrix Inversion within X-Midas

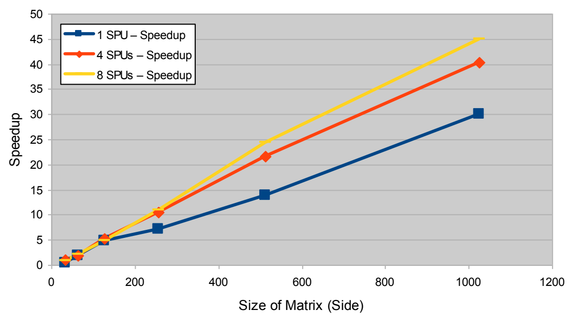

Figure 11. Speedup Compared to a single thread PPU version Complex valued Matrix Inversion within X-Midas

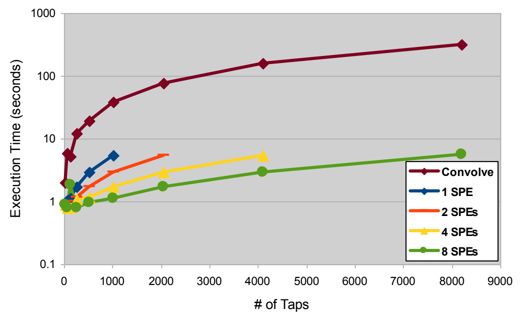

Figure 12. X-Midas FIR Filter Execution Time for Real Taps with Complex Data

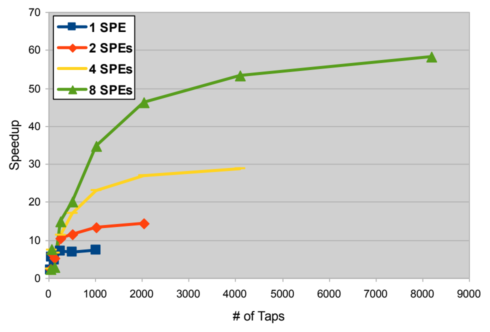

Figure 13. X-Midas FIR Filter Speedup for Real Taps with Complex Data compared with Convolve

We implemented a FIR filter using a convolution algorithm and compared it to the CONVOLVE intrinsic in X-Midas. All tests were run with a million element complex data set and varying numbers of real number filter taps.

In Figure 12, the execution time for the CONVOLVE intrinsic and the accelerated FIR function are plotted. Datasets we tested for 16 to 8192 taps. However, limitations on the size of the local store in the SPUs, restricted the applications to 1024 taps per SPU. Runtime for the *convolve* intrinsic is shown in blue, with runtimes for 1 to 8 SPUs shown below. Figure 13 shows the data as a speedup curve relative to the *convolve* intrinsic. The 8 SPU implementation performed the best, reaching nearly 60x faster for the 8192 tap implementation. Moreover, the speedups are nearly linear over the number of taps, indicating that many of the optimizations for data transfers and register management were successful in completely hiding the latency associated with these operations.

D. Disk Caching Anomalies in X-Midas

When benchmarking each of the accelerated functions in X-Midas we encountered a number of anomalies in X-Midas runtimes that could not be consistently explained by conventional sources of overhead. Seemingly at random, a large spike in the execution time of the function being tested would be observed as in the FIR benchmark result in Figure 14.

To determine the cause of this problem, the accelerated function was augmented with additional execution time

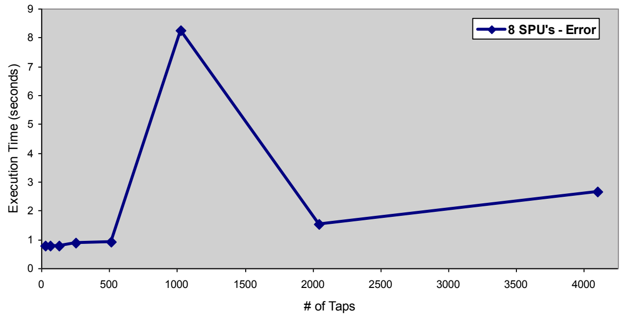

Figure 14. X-Midas Caching Causing Unpredictable Results with a FIR Benchmark

counters to allow benchmarking in three separate stages of execution, the setup and read phase, the kernel of computation phase, and the cleanup and write-back phase. These execution time counters provided us with insight as to which stage was the cause of the issue, as in the counter results from the above FIR test, shown in Figure 15.

It was immediately clear that disk I/O was causing transient spikes in the execution time since the runtime of the core computational kernel was not changing, but there were large differences in the sections with disk activity. Further testing revealed that the X-Midas caching system was executing a sync() operation, flushing its caches when these spikes occurred. Disabling this mechanism and forcing immediate write back on all disk operations resulted in speeds that were the same as observed during these spikes for all test runs. The poor write speeds of the disk in the CBE Blade, along with pressure on the disk caching mechanisms due to low global memory totals are probably responsible for these spikes. Additional system memory and writing over the network or to a faster storage device would likely help to prevent these spikes. In any case, while these results are not indicative of the performance under normal operating conditions, they should still be taken into account when considering the overall performance of the accelerator.

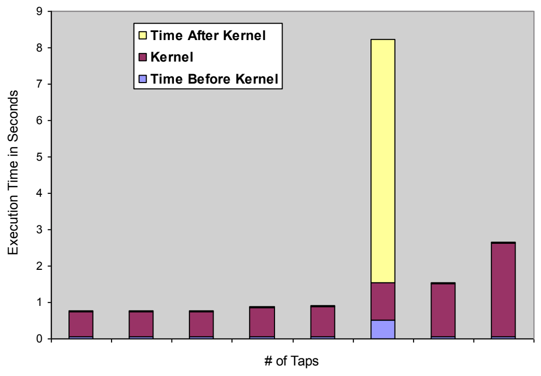

Figure 15. Breakdown of the execution time of each segment of execution

Designs that avoid writing back to disk, such as the X-Midas piping system or network storage, should be considered when writing scripts for the CBE Blades with hard disks to avoid this problem.

E. Summary and Conclusions

In this study we have ported the X-Midas platform to the CBE and demonstrated that significant acceleration of DSP functions for the X-Midas Toolkit. For streaming applications, which chain the accelerated DSP operations in a pipeline parallel fashion, the CBE can provide a significant performance improvement over non-accelerated applications. However, the system overhead and memory management functions in X-Midas are significant and may limit the results attained for certain algorithms. Standalone applications, carefully optimized using the three methods to exploit low level hardware features of the CBE, as presented here, are the most effective means for exploiting this architecture.

References

[1] A. J. Kahle, N. M. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, Introduction to the Cell Multiprocessor. IBM Journal of Research and Development, July 2005, Vol. 49, pp. 589-604.

[2] R. C. Johns and A. D. Brokenshire, Introduction to the Cell Broadband Engine Architecture. IBM Journal of Research and Development, Sept. 2007, Vol. 51, pp. 503-519.

[3] Doty, G. and Jones, D. X-Midas User's Guide. Tuscon, AZ : Rincon Research Corp., 1996. MRSL. [Online] http://mrsl.com

[4] Eaton, J.W. GNU Octave. [Online] http://www.gnu.org/software/octave

[5] The Mathworks Inc. MATLAB. [Online] http://www.mathworks.com/products/matlab

[6] S. Yokoyama, , K. Matsumoto, and S.G. Sedukhin, Matrix Inversion on the Cell/B.E. Processor. IEEE HPCC, 2009. High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on. pp. 148-153.

[7] Scarlett, Kane. Discover the LAPACK Library. [Online] IBM, Sept. 02, 2008. http://www.ibm.com/developerworks/power/library/pa-trlapack1/index.html

[8] IBM Corp. IBM Compilers. [Online] 2008. http://publib.boulder.ibm.com/infocenter/cellcomp/v9v111/index.jsp

[9] Core Asset Manager. Midas Programmer's Guide. Tuscon, AZ : Rincon Research Corp.

[10] J. A. Jezak, Realtime signal processing benchmarking on the cell broadband engine. M.S. Thesis, Computer Engieering Program, University of Pittsburgh, 2011.