Programming Examples that Expose Efficiency Issues for the Cell Broadband Engine Architecture

William Lundgren[1] (wlundgren@gedae.com), Rick Pancoast[2] (rick.pancoast@lmco.com), David Erb[3] (djerb@us.ibm.com), Kerry Barnes[1] (kbarnes@gedae.com), James Steed[1] (jsteed@gedae.com)
1: Gedae, Inc., 1247 N. Church St., Suite 5, Moorestown, NJ 08057
2: Lockheed Martin, 199 Borton Landing Rd., Moorestown, NJ 08057 (Pending Corporate Approval)
3: IBM, 11501 Burnet Rd., Austin, TX 78758 (Pending Corporate Approval)

Introduction

The Cell Broadband Engine™ Architecture was developed as a collaboration between Sony, Toshiba, and IBM. The current implementation of the Cell Broadband Engine (Cell/B.E.) processor combines one Power Processing Element (PPE) with 8 identical Synergistic Processing Elements (SPE). The PPE is a dual-threaded PowerPC core with an instruction set that has been extended to include SIMD (Single-Instruction, Multiple-Data) instructions known as VMX. Each SPE contains a high-speed SIMD processor with its own 256 kB local store and DMA engine. The nine cores and the on-chip memory controller and I/O controller are interconnected with the high speed Element Interconnect Bus (EIB) as shown in Figure 1. The EIB provides a measured peak bandwidth of over 200 GB/s at 3.2 GHz.

.

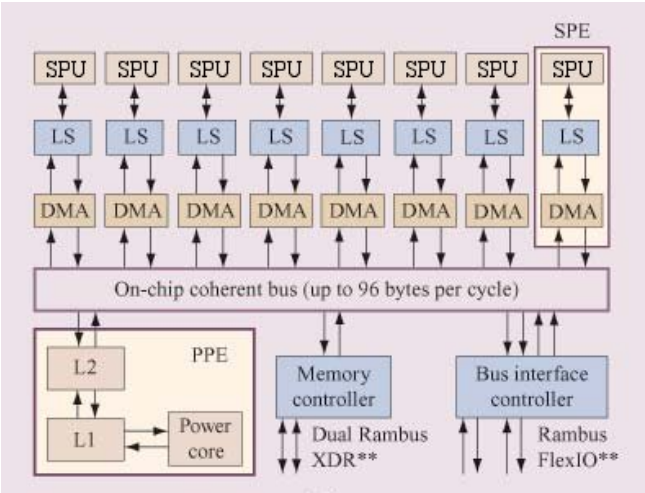

Figure 1 – The current Cell/B.E. Processor combines a PPE core with 8 SPE cores, all interconnected via a high-speed bus.

Because the PPE provides relatively modest performance, the key to using the Cell/B.E. processor efficiently is to take best advantage of the SPEs. While an SPE can process vector arithmetic very efficiently, each SPE has only 256KB of dedicated local storage which holds both instructions and data. Thus when processing a large data set, the developer must either distribute the large data set across the processing cores or store the data in off-chip system memory. For instance, an entire 1k-by-1k complex matrix cannot reside in the SPE local storage, even if the program size is zero. Therefore large data sets will initially reside in system memory, and then the developer will stripmine the data, bringing it chunk-by-chunk to the SPE local storage for processing.

This paper will study how these programming considerations affect the implementation of the synthetic aperture radar (SAR) algorithm. We will study the performance improvement possible by distributing the work to the Cell/B.E. processor's cores, as well as the issues that must be resolved to create that distribution. Additionally we will study a distributed FFT where the data for each vector is distributed across the SPEs. The Gedae programming language and development tools will be used to conduct these experiments on the Cell/B.E. Architecture and analyze the performance.

Distributed SAR Algorithm

To study the practicality and performance of real-world applications on the Cell/B.E. processor, we will port a SAR algorithm to the PPE and SPEs. The SAR algorithm has two key stages: the range processing of the rows of the matrix, and the azimuth processing of the columns of the matrix. To distribute the SAR algorithm we must add three stages: the partitioning of the data to distribute the rows across the processors, a corner turn of the data (distributed matrix transpose) to transition between range and azimuth processing, and the concatenation of the column-based results. In real-world applications, we must also do additional preprocessing to unpack the data, remove the possibility of phase errors, and additional error reduction. With these added processing requirements due to the distribution, the distributed SAR algorithm is as shown in Figure 2.

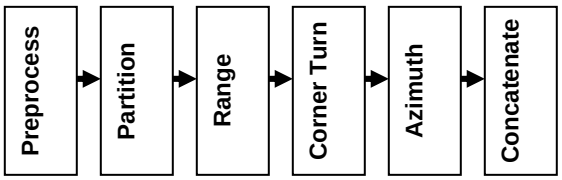

Figure 2 - Stages of the distributed SAR algorithm.

A key issue in implementing this distributed SAR algorithm efficiently on this architecture is the implementation of the corner turn. While the EIB provides a high bandwidth between the SPEs for corner turn operations, the data sets typical for a SAR algorithm cannot fit in the local storage of the SPEs. To do the work of a corner turn on a large SAR data set, the results of the range operation must be transferred to system memory, and then – once full columns

are available – transferred back to the SPEs in transposition. Because this approach places demands on the bandwidth between the SPEs and system memory as opposed to the relatively fast bandwidth between SPEs, there is a concern that the transpose can dominate the time of the algorithm.

A second issue in implementing this algorithm on the this architecture is maximizing the number of vectors that are computed together on the SPE's. The SPE's can perform vector processing with amazing efficiency, so the developer must maximize the size of these vectors to gain as much performance benefit as possible. As the SPE's local storage must be used as both program memory and data memory, its size is a limiting factor. Program overhead must be minimized and unused code must be removed so that as many vectors as possible can be placed in the SPE at one time.

Gedae has been used to implement this distributed SAR algorithm and map it to all 8 SPEs on a Cell/B.E. processor. The Trace Table for this implementation is shown in Figure 3. The top of the table shows the load for each of the 8 SPEs (the black bars showing time the SPE is processing data), and the bottom of the table is collapsed to show the processing of one of the 8 threads in the application, highlighting the range, transpose, and azimuth processing. As shown in the table, the SPE utilization is very high. For comparison, this same Gedae application was run on a quad DSP board with 500MHz PowerPC processors. The PowerPC implementation achieved 3 frames per second. Once optimized routines are incorporated into the Cell/B.E. processor board support package for Gedae, we expect the implementation to achieve over 130 frames per second.

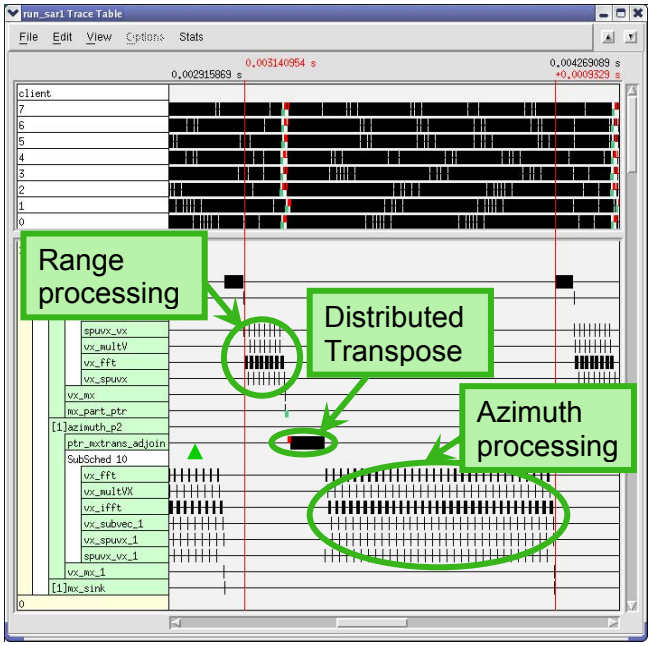

Figure 3 – The Trace Table of the 8 SPE implementation shows the processing is very dense, and the distributed transpose takes a modest portion of the time per data set.

Parallel FFT Algorithm

To study the Cell/B.E. processor's ability to rapidly process data we will also study a parallel implementation of an FFT. In this implementation, the data is distributed across the SPEs' local storage. To perform an FFT on a vector of length N, first we perform √N FFT operations on vectors the size of √N and apply a weight vector to the results. Then a corner turn is performed, and another √N FFT operations of size √N are performed.

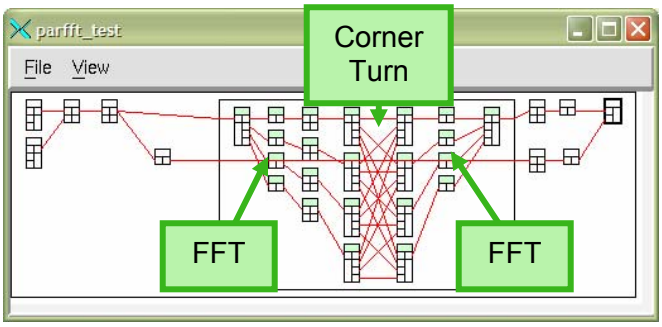

Figure 4 - This parallel FFT implementation distributes the $\log_2$N-by-N/2 butterfly operations to the 8 SPEs.

The key issues in implementing this distributed FFT operation are distributing the work across the processors, implementing the corner turn, and maximizing the amount of data processed in each execution. The data sizes for the FFT allow this corner turn to be implemented without using system memory. As the EIB on the Cell/B.E. processor is implemented as a data ring, structuring the partitioning and mapping of these stages to best take advantage of the ring structure is key to achieving high performance. Several different mapping schemes are investigated using Gedae to determine the best scheme for this algorithm.

Conclusions

Multi-core architectures like the Cell/B.E. Architecture are powerful compute engines. The bus and memory architectures of these multi-core processors play an important role in determining their applicability to a wide variety of problems. Making proper use of these components has a large impact on achieving performance close to the published theoretical maximums of the architectures. Coding for these architectures can easily obfuscate these issues and direct the effort away from exploiting the hardware and towards simply getting the code working. Addressing these issues in a structured, systematic way, such as using Gedae, frees the developer to focus on making best use of the hardware while still achieving high performance with low overhead.

References

[1] IBM, Sony Computer Entertainment, Toshiba. *Cell Broadband Engine Programming Handbook*, Version 1.1, April 2007. <http://www.ibm.com>.

[2] J. A. Kahle, et al., "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, Vol. 49, No. 4/5, 2005.