Application Implementation on the Cell B.E. Processor: Techniques Employed

John Freeman (freeman@brsc.com), Diane Brassaw (brassaw@brsc.com), Rich Besler (besler@brsc.com), Brian Few (few@brsc.com), Shelby Davis (davis@brsc.com), Ben Buley (buley@brsc.com) .
Black River Systems Company Inc., 162 Genesee St. Utica NY 13501

Summary

Development of real-time and non real-time algorithms on a Cell Broadband Engine™ (Cell B.E.) is challenging. The multi-core architecture of the Cell B.E. combined with small memory constraints and limited instruction set on the vector engines create a development environment with an interesting set of problems. Black River Systems has ported several algorithms and has developed optimized assembly language functions on the Cell B.E. The objective of this paper is to discuss the techniques we have employed to successfully take advantage of the capabilities of the Cell B.E. processor.

Motivation

The Cell B.E. processor is a unique and interesting heterogeneous processing architecture with impressive single precision floating point performance of over 200 GFLOPS.

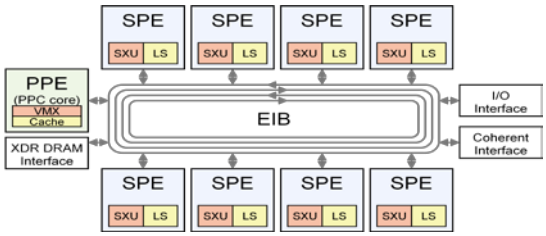

Figure 1: Cell B.E. Architecture

Figure 1 shows the Cell B.E. architecture consisting of a 3.2 GHz Power Processing Element (PPE), 8 enhanced independent Synergistic Processing Elements (SPEs), a high speed Element Interconnect Bus (EIB), high-speed I/O options, and high speed XDR memory. Coupling this performance with the availability of a low cost ($400) development environment in the form of the PlayStation 3 (PS-3) and the availability of higher density platforms in the Mercury 1U server and IBM Blade Center makes the Cell B.E. a very attractive target for compute intensive applications.

Introduction to Programming the Cell

Programming the Cell B.E. is not a trivial process. The first challenge is to figure how to make the algorithm (including data) fit in the small 256K SPE local store.

- When mapping an algorithm you must identify data independent algorithm components and distribute these components across multiple SPE's.
- If you have large FFT's you will want to use the mega-stage/transpose/DMA technique.
- Strip-mine long vectors to/from XDR.
- Use the PPE for shared computation and process management.

The second challenge is to make the algorithm run fast.

- Overlap computation on SPE and data transfer between SPE and XDR DRAM.
- Utilize vector instructions on the PPE and SPE.
- Make use of the extensive register set on the SPE
- Unroll loops to avoid stalls due to operand unavailability
- Analyze inner loop execution. A useful tool to use is Assembly Visualizer (asmVis) provided by IBM as shown in Figure 2 .

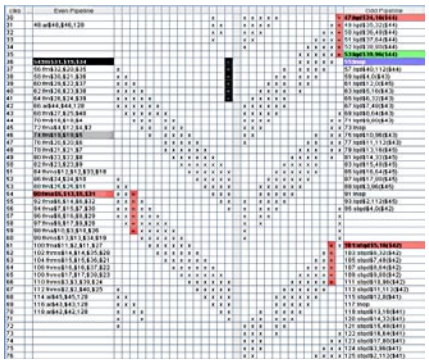

Figure 2: Loop Performance in IBM asmVis

- Use interrupts or mailboxes from SPE to PPE to signal operation completion

The third step is to connect the functional components to provide a cohesive solution. Finally balance the loading on each SPE such that SPE idle time is minimized.

Techniques Employed

Below is a brief discussion of the techniques employed in order to successfully follow the programming approach described and take advantage of the capabilities of the Cell B.E. processor. This includes a low-cost development environment, SPE assembly programming, techniques for loop unrolling, parallel implementations, and performance metrics.

Development Environment

The Sony PlayStation 3 (PS-3) has a 3.2 GHz Cell B.E. Chip with 6 SPEs, 256MB XDR RAM, 40 GB Hard drive, and Gigabit Ethernet and USB connectivity for ~$400. Installed with Fedora 7 Linux, IBM SDK and using GNU GCC and associated debuggers, it provides a low-cost development station for software development. A rebuild of the Linux kernel to allow networking options, trim unneeded sections, and select different memory performance options resulted in a 2 week initial software/hardware setup and an additional ½ day applied

time for additional systems. The migration of applications developed on the PS-3 to other Cell B.E. platforms is seamless with no performance differences noted in the SPE code.

SPE Assembly Programming

The code generated by the C compiler is often suboptimal for the inner loop of a program. When it is important to get every single cycle of performance out of an algorithm, coding the inner loop in assembly is the only way. Since all of the assembly commands on the SPEs are vectorized, it provides an incentive for the developer to think hard about how the implementation is structured, and gives them an opportunity to use SIMD techniques as often as possible. Knowledge of the IBM SDK C intrinsics offers a good step into the more opaque assembly programming environment, but all of the higher level language features like types and memory protection are no longer available to the programmer. On the other hand, the compiler is not doing any transformations to your code, so performance bottlenecks are easier to diagnose.

Techniques for Loop Unrolling

One approach is to use text processing in Java. Using this approach, we are able to specify 'computational blocks' within an assembly implementation of an inner loop and then eliminate stalls by unrolling the loop. This template can be used to unroll a loop as many times as is necessary to eliminate all of the stalls generated by the interdependence of the assembly instructions in the routine. In practice, a loop will rarely need to be unrolled more than six times, as this is the longest computational latency of any of the single precision operations in the SPE instruction set.

A second approach is to use C++ Template Meta Programming to generate semi-automatic loop unrolling. This use of templates allows the compiler to determine the optimal number of registers in use at once. In addition, because the loops were programmatically unrolled with template recursion, all values derived from a combination of constants and loop counts (e.g. address offsets and logical bit masks) were computed at compile time reducing the total number of computations per loop. Since there were no inner loops, the dependency of the data was much easier for the compiler to see therefore allowing the compiler to make better optimizations and concurrently process multiple loop(original) iterations at once. This type of optimization increases the execution speed but may lead to a very large execution image making it infeasible to target to the Cell local store.

Parallel Implementation

Execution using POSIX threading is used to manage each algorithm component with various parallel implementation techniques such as a fixed function pipeline and a queue managed use of the SPEs as an offload engine. In the fixed function pipeline method shown in Figure 3, SPE functions are assigned a PPE proxy thread, simplifying the main program to a scheduler. Queue managed use of SPEs or load balancing makes use of the next available SPE for the next function that needs to be done as shown in Figure 4.

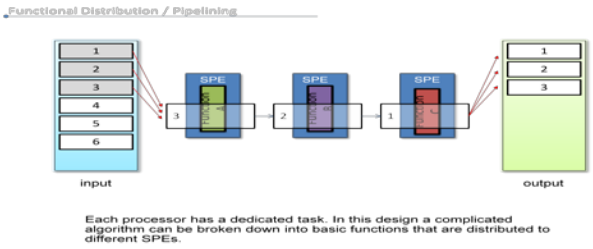

Figure 3: Fixed Function Pipeline

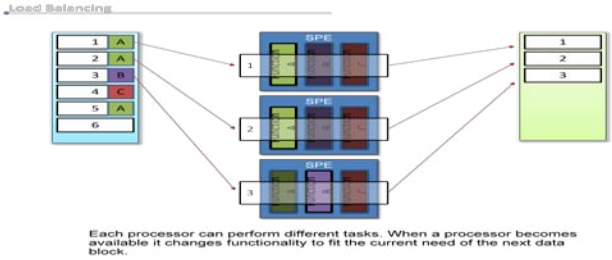

Figure 4: Load Balancing

Performance Metrics

The PPE time-base registers are used to collect event stamps at the start and end of each threads execution and analyzed graphically to assess load balance, latency, idle time, and to optimize the Cell execution as a whole. Since a thread is used to control a SPE, then the SPE execution time and execution placement is also conveyed as shown in Figure 5. Each line represents a thread, some of them running SPE code, some only PPE code. Using the timing metrics and the profiling GUI tool, the developer can analyze where there are bottlenecks, where the code needs further optimization, and if the SPE distribution needs to be changed or more SPEs added.

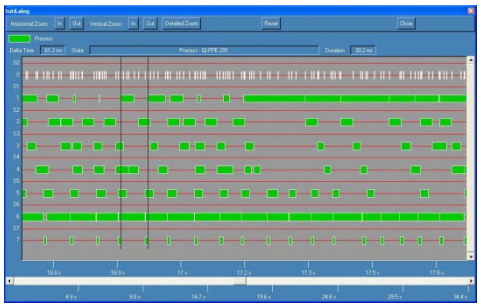

Figure 5: Black River Profiling Tool

Conclusion

Using the techniques described has enabled us to obtain impressive performance on the Cell B.E. processor. For example, one application that originally ran on a 64 node PPC7410 500 MHz system was reduced to running on a 1U Cell server. A 10x decrease in deployed system cost and an increase in system performance was achieved. When paired with the correct applications the Cell B.E. is a very capable processor which can provide significant performance gains over other current processor technologies. Programming for the Cell B.E., while challenging, has proven no more difficult than traditional embedded multi-processor systems. Dealing with the unique features of the Cell B.E. presents an appreciable learning curve but results in performance nearly matching the theoretical peak.