An introduction to compiling for the Cell Broadband Engine architecture, Part 1: A bird's-eye view

Overview

Skill Level: Intermediate

Power Architecture editors (dwpower@us.ibm.com)
developerWorks
IBM

07 Feb 2006

Meet the Cell Broadband Engine™ (Cell BE) processor from a compiler-writer's perspective, and get a bird's-eye view of a number of the unique challenges it poses in this first tutorial of a five-part series. Part 1 provides useful background information relevant to the other tutorials in the series.

Section 1. Before you start

About this tutorial

This tutorial presents an overview of the concepts essential to understanding Cell Broadband Engine (Cell BE) architecture and how a compiler can implement solutions to automatically distribute code to the Synergistic Processing Elements (SPEs) of Cell BE architecture. It is based mainly on the experiences of IBM Research in creating the Octopiler optimizing compiler, but also includes input from the Visual Age XL compiler team; as a series, it should be useful to other compiler writers, but also to anyone who would like a clearer understanding of Cell BE architecture in general. It introduces the local store memory of SPEs, the instruction buffer, auto-SIMDizing code, and the software cache approach to accessing irregular data. Subsequent tutorials discuss these topics in greater detail.

The topics covered in this five-part series include:

- Part 1: Overview: The Cell BE architecture and some of the issues faced in compiler design
- Part 2: Optimizing for the SPE: Optimizations used on the SPEs, such as how the compiler translates scalar code for a vector-only processor
- Part 3: Making the most of SIMD: How a compiler can effectively generate SIMD code for two different architectures (the SPE and VMX), accommodating the various technical constraints of the processors
- Part 4: Partitioning large tasks: How the compiler, or the user, can divide tasks up between the SPEs and the main processor
- Part 5: Managing memory: Techniques used, by the compiler or the programmer, to give the SPEs access to data that can't fit in local storage

Prerequisites

Basic familiarity with computer architecture is helpful, but nearly any programmer in possession of a fully functional brain should be able to follow along.

Section 2. Overview

Tutorial objectives

This series provides an understanding of the Cell BE architecture, a basic intuition for programming issues on it, insight into the compiler challenges presented by it, and an understanding of the techniques and solutions proposed by the IBM compiler. Part 1 offers the background material and overview information assumed by the other tutorials in this series.

About the Cell Broadband Engine architecture

The Cell BE processor has 241 million transistors, and a total size of 235mm ^{2}. Its theoretical performance runs around 200GFlops (single-precision) at 3.2GHz, and the internal bus has about 200GB/sec of bandwidth at the same clock speed.

The Cell BE processor has a general-purpose CPU, called the Power Processor

Element (PPE), which is essentially a PowerPC® CPU, used to run full-fledged OSs and provide services. There are also eight Synergistic Processor Elements (SPEs) which are optimized for computation density. See Resources for more information about Cell BE architecture.

Architecture overview

The Cell BE processor is a heterogeneous multicore engine, with one multithreaded PowerPC processor core and up to eight cores running an ISA designed for computationally intensive tasks. The SPEs have fast access to 256KB local store, and globally coherent DMA for transferring data from main memory.

The Cell BE processor has pervasive single instruction, multiple data (SIMD) functionality. The PPE has the VMX (or AltiVec) vector processing extensions to the PowerPC ISA, while the SPEs have a SIMD-only instruction set.

The bandwidth of the Cell BE processor is unusually high, with a practical internal bandwidth of 200GB/sec, and a dual XDR controller providing 25.6GB/sec of memory bandwidth. Two more configurable I/O interfaces offer another 76.8GB/sec of bandwidth. (All of these numbers assume a 3.2GHz clock rate.)

Architecture: EIB

The Cell BE processor is built around the Element Interconnect Bus (EIB), which is a high-speed internal bus for moving data between components of the processor, giving each functional unit (each of the eight SPEs, the PPE, the memory interface, and the two I/O units) bandwidth of 25.6GB/sec each way. The EIB's theoretical bandwidth, of 100-200GB/sec, can only be attained by fairly well-tuned code.

Architecture: SPE

The SPEs run a custom SIMD-only instruction set. Each SPE has 128 general-purpose registers, each 128 bits wide, used for vector operations. The SPEs have no direct support for scalar operations; scalar operations must be specially handled by the compiler. Each SPE has a 256KB local store, which is used to hold code and data being processed by the SPE. The Cell BE processor provides DMA support for memory transfers to and from SPE local store, whether the data are then moved to main memory, another SPE, the PPE, or the I/O subsystem. The local store is not a cache, although it is possible to use part of it to provide copies of main memory storage.

Architecture: PPE

The PPE of the Cell BE processor is a dual-threaded 64-bit PowerPC implementation. It is not a dual-core processor, but it has hardware support for two threads, each with an independent register file. The PPE is primarily used in a supervisory role, or to provide OS services, for code being run on the SPEs.

Section 3. About the compiler

The optimizing compiler for the Cell BE processor

When tutorials in this series state that "the compiler" does something, they are describing features of the research compiler, some of which are also found in XL or in GCC. Over time, all of these features, or improved features based on their designs, are expected to become available in commercial compiler architectures.

Compiler objectives

The ideal compiler balances performance and productivity, easing the burden of development for a novel architecture, without locking out the expert programmer.

One goal of the compiler is exploitation across a range of programming styles. The compiler offers automatic vectorization, but also provides intrinsics to give direct access to the SIMD functionality of the processors. The compilers for both PPE and SPE units are highly optimized, supporting C, C++, and Fortran.

Using a single source tree for a program combines respectable performance with ease of use. The most appropriate language and user directives are applied to exploit several levels of parallelism (SIMD instructions, multithreading, and the multiple cores available for execution). A single shared memory image for the executable is much more manageable than a collection of separate files would be. Code can then be partitioned as needed for size and functionality.

Compiler background

The Cell BE processor's novel design, offering "heterogeneous" parallelism (as the processor cores do not run the same instruction set), offers unique challenges for compiler development. The PPE is based on the well-established PowerPC architecture, but the SPEs are not. The SPE processors have design qualities that require special treatment from the compiler; for instance, they have no hardware

branch prediction, a dual-issue architecture, and single-ported memory. In effect, the compiler must contain compilers for both the PPE and SPE.

For the compiler to effectively support the Cell BE processor, it has to support parallelization across the heterogeneous processors. The compiler has to be able to recognize partitions between the PPE and SPEs, and has to provide a usable abstraction of the non-uniform memory.

Additionally, the compiler needs automatic vectorization techniques applicable both to the SIMD instructions of the PPE and the somewhat different SIMD instructions of the SPEs.

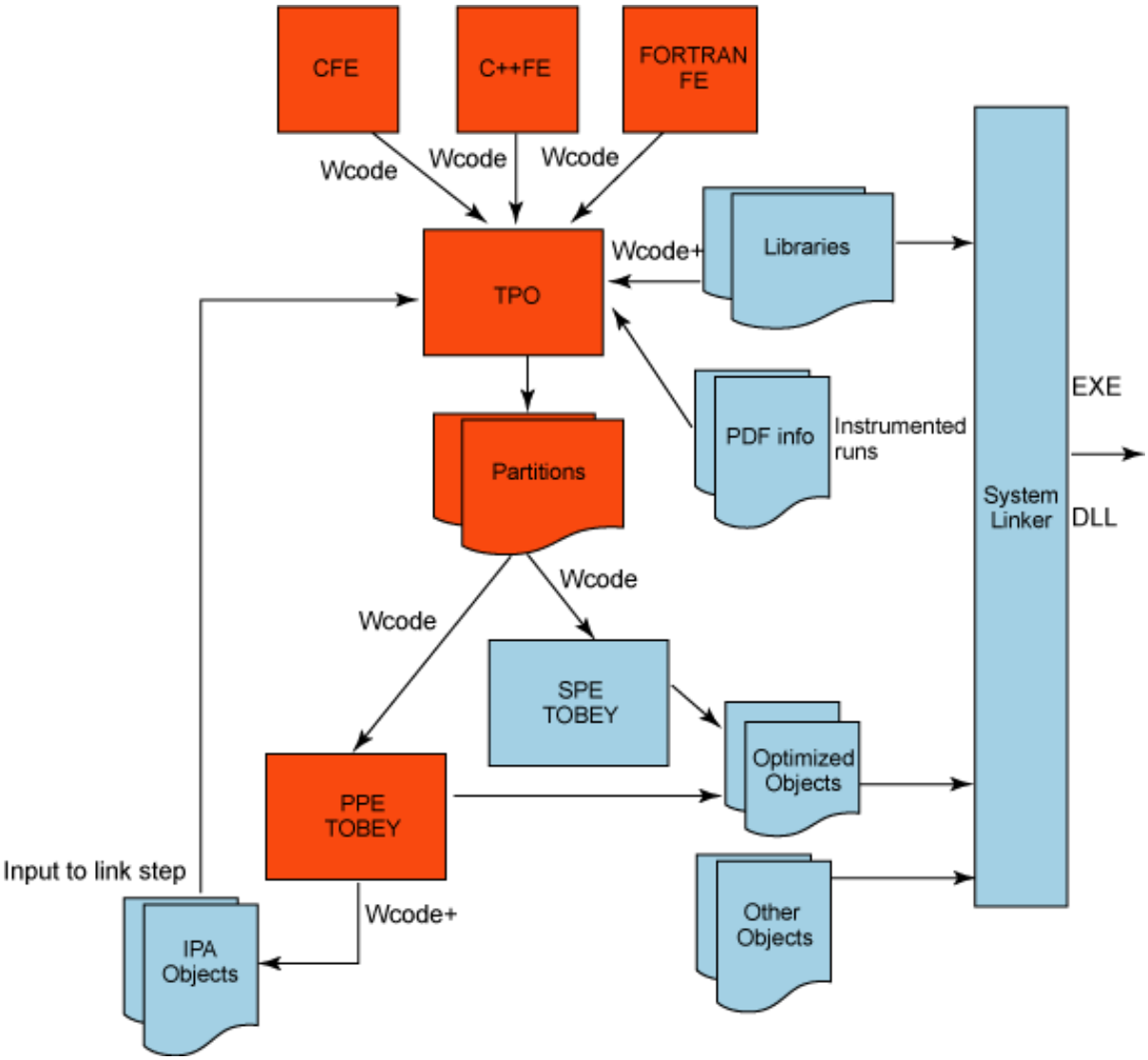
Compiler versions

The compiler discussed in this series is a research prototype compiler only. It is integrated with the product code base, and has been developed in collaboration with the compiler development group, but many research issues are still in progress. Some of the features are based on the OpenMP specification. The XL compiler has some, but not all, of the features described in this series; the XL C compiler for the Cell BE processor is currently available as an evaluation download from alphaWorks (see Resources).

Compiler framework

The following figure illustrates the framework for the compiler as it currently stands.

Figure 1. The structure of the Cell BE compiler



Supporting a broad range of expertise

The Cell BE compiler is intended to support a broad range of expertise and needs: programmers might choose to leave tasks to the compiler, improving productivity, or might take a more hands-on approach and potentially improve performance. Next, we'll discuss the compiler technologies used to automate each of the three major examples of the range of expertise assumed.

Different ways of applying programmer expertise

The first is the program level. An expert programmer can write hand-tuned programs for the SPEs and PPE, writing code specific to each. A more casual programmer can simply write generic code, leaving the compiler to optimize and tune automatically

for each ISA.

The second is SIMD instructions. Programmers might explicitly code SIMD instructions, use SIMD/alignment directives to guide the compiler, or leave simdization entirely up to the compiler.

The third is memory management. Programmers might explicitly parallelize code, controlling the movement of data from one core to another, use a single program/shared memory abstraction, or take advantage of the compiler's automatic parallelization.

Section 4. Automatic SPE tuning

Relevant architectural issues

The SPE has a number of qualities which add complexity to programs, which the compiler can take care of automatically.

First, the SPE's dual-issue architecture requires careful instruction scheduling for maximum performance. Second, local memory is single-ported, so memory operations (whether DMA or loads into registers) can potentially cause instruction starvation. Finally, the branch architecture has no hardware branch prediction, leaving it to software to manage hints and predication to reduce the cost of branches.

Feature 1: Dual-issue architecture

On the SPE, two instructions can be issued every cycle, but only if one of them is a memory operation, and one is an arithmetic operation. Furthermore, alignment matters; the dual-issue works only if the arithmetic operation is in the "even" channel (multiples of eight bytes) and the memory operation is in the "odd" channel (four byte offsets).

The compiler has several features allowing it to improve bundling of instructions. Pairs of instructions which are not dependent, and simply come in the wrong order, can be swapped. In cases where a swap is not possible, it might be necessary to insert NOP (Not and OPeration) operators to align instructions for best performance. The bundler is integrated with the scheduler for the SPE, because changes at this level might change the ideal places for branch hints and instruction fetches. These optimizations have a particularly significant effect on the performance of generated

code.

Feature 2: Single-ported local memory

Local memory is single-ported, which makes the hardware less expensive. The port is asymmetric, providing 16 bytes at a time for load/store operations, but 128 bytes at a time for instruction fetch (IFETCH) and DMA operations. It has static priorities: DMA requests take priority, followed by load/store operations, followed by IFETCH. The latency faced when loading instructions into a buffer can be significant; as a result, the processor can easily starve for instructions without careful planning.

Instruction starvation situation

There are two instruction buffers, with up to 64 ops along the fall-through path. When the first buffer is half-empty, a refill can be initiated (it will not take effect before the first buffer is empty). When the MEM port is continuously used, however, no instructions will be in the buffer, and the processor will grind to a halt until new instructions can be fetched.

Instruction starvation prevention

The SPE has an explicit IFETCH operator, implemented using the branch hint (HBR) instruction with the P flag, which initiates an instruction fetch. The compiler's instruction scheduler, which organizes code for efficient use of CPU resources, monitors the starvation situation, especially when the MEM port is continuously used, and inserts an IFETCH op fairly early. This has the impact that the scheduler must keep track of code layout.

Feature 3: Software-assisted branch architecture

The SPE's branch architecture has no hardware branch-predictor. It does have compare/select ops for predication, and a software-managed branch hint. (Only one hint can be active at a time.) The overhead of branching can be reduced by predicating small if/then/else operations, and hinting predictably taken branches. Hint opcodes are discussed in Section 8 of the SPU Instruction Set Architecture documentation (see Resources for a link).

Hitting branches and instruction starvation prevention

The SPE's HINT instruction fetches the branch target into the HINT buffer. There is no penalty for correctly predicted branches, but the HINT instruction must be 15

cycles, and at least eight ops, before the actual branch. The compiler automatically inserts hints when beneficial.

This does impact instruction starvation: the window for an effective IFETCH is smaller after a correctly hinted branch.

Section 5. Automatic simdization

SIMD computation

SIMD operations process multiple data per operation, for instance, loading four members of an array at once, then adding them to four members of another array all at once, instead of loading and adding sequentially.

Successful simdization

Successful simdization requires extracting parallelism at the loop level, basic-block level, and short-loop level. Furthermore, key constraints must be satisfied: SIMD operations on the Cell BE processor have alignment constraints (only loading and storing at 16-byte boundaries), and must involve objects of the same size. Finally, multiple targets (VMX and the SPE) are targeted.

Example of SIMD-parallelism extraction

At the loop level, SIMD instructions can be generated for code similar to the following:

```
for (i = 0; i < 256; ++i)
        a[i] = ...;
```

The compiler handles misaligned data and pattern recognition (such as reduction and linear recursion), and leverages existing loop transformations.

Example of SIMD constraints

The SIMD units in the SPE (and in the VMX unit of the PPE) perform loads and

stores only at 16-byte boundaries. Given two aligned arrays, b and c, trying to add `b[i+1]` to `c[i+0]` will not work, because the desired array members will not be in the same position within an SIMD register.

When alignments within inputs do not match, the compiler inserts code to realign the data correctly to produce the desired results.

Automatic simdization for the Cell BE processor

The IBM compiler uses an integrated approach: potential opportunities for simdization are extracted at multiple levels, and then adapted to satisfy all SIMD constraints. This is done using an abstraction, a virtual SIMD vector, as glue.

Alignment overhead is minimized by postponing data reorganization when possible. The compiler can handle compile-time and runtime alignment, although compile-time alignment handling is generally more efficient. Prologues and epilogues generated for loops are simdized, too. It helps that memory access on the SPE never generates exceptions due to alignment or other issues; incorrect code may produce surprising results, but the processor won't abort it.

In general, computations can get full throughput or close to it, even in the presence of data conversions.

Section 6. Shared memory and single program abstraction

Cell BE memory and DMA architecture

The Cell BE memory and DMA architecture are fairly flexible. The local stores of the SPEs are mapped into the global address space. The PPE can access (through DMA) local stores on the SPEs, and can set access rights. The SPEs can initiate DMA to any global address, including the local stores of other SPEs; The MMU does translation. All units may be masters; there are no designated slaves.

Competing for the SPE local store

The local store is very fast, so the compiler needs to provide support when the local store may be full. The compiler handles three possible cases, each of which might

be relevant to different programming models (see "Unleashing the power of Cell Broadband Engine: A programming model approach" in Resources).

If the SPE code itself is too large, the compiler partitions code, and a partition manager pulls in code as needed.

If data with regular access patterns is too large, the compiler stages data in and out using static buffering, and can hide latencies by using double buffering.

If data with irregular accesses is present (such as indirection, or runtime pointer usage), a software cache can be used to pull data in and out; this is something of a last resort.

Design of the software cache

The software cache is designed to handle data with irregular accesses. These data generally cannot reside permanently in the SPE's local memory, so they must reside in global memory. When accessed, the global store address is translated to a local store address (in the cache). Data that is not already present must be obtained, possibly flushing something else from the cache first.

The software cache is managed by the SPEs in the local store, and generates DMA requests for transfers to and from global memory. The cache itself is four-way set associative to naturally use the SIMD instructions of the SPE.

Software cache architecture

The software cache directory is a 128-set, four-way set associative array with pointers to data lines, using 16KB of data. Data is stored in a separate structure, holding 512 lines, each of 128 bytes, for a total of 64KB of data.

Software cache access

To translate a global address, the SPE splits it into an address offset, a set, and the chunk of the address used by tags in the cache directory; the latter is splatted into a full four-way SIMD register, which is then compared against a set of four tags in the cache directory. If there is a match, the data pointer is obtained, and the address within that cache line is determined by the address offset left over from the initial global address. The hit latency is about 20 cycles, which is a tiny fraction of the cost of a miss.

Section 7. Single source compiler, using OpenMP

The Cell BE compiler is a single-source compiler, using OpenMP directives to control compilation. For instance, a developer might write code like this:

```
#pragma Omp parallel for
for (i = 0; i < 10000; ++i)
        A[i] = x * B[i];
```

The compiler then outlines the omp region, creating a new function that performs only this work. This is cloned into two versions, one to run on the PPE, and one to run on the SPE.

At runtime, the SPE version of the code uses DMA to obtain chunks of data, and uses the software cache if necessary for lookups (for instance, of the value x in the above example). On the PPE, the OpenMP runtime is initialized, which begins handing off pieces of the loop to the SPEs. The PPE may also participate in calculations if desired.

Conclusions

The Cell Broadband Engine architecture offers a very dense computing architecture, with heterogeneous parallelism.

The compiler presents the application writer with a wide range of tools. These tools range from support to allow a developer to hand-tune code to extract maximum possible performance, to support for largely automatic management of the Cell BE processor, giving maximum programmer productivity.

The compiler has demonstrated reasonable speedups using automatic tuning, simdization, and support for the shared-memory abstraction.

Tune in next time

Part 2 of this tutorial series takes a much closer look at SPE optimizations, such as instruction fetch timing and branch hinting.

Acknowledgement

This tutorial series is based on the original presentation Optimizing Compiler for the Cell Processor given at PACT 2005 by Alexandre Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter Oden, Daniel Prener, Janice Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind of IBM Research.

Part 1 is based on the section "Cell Architecture and Compilation Techniques."

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc.

Downloads

- Product: Full-System Simulator for the CBE
- Product: XL C Alpha Edition for the CBE
- Product: CBE Software Sample and Library Source Code
- Product: GCC Toolchain for the CBE
- Product: CBE SPE Management Library
- Product: Linux kernel patch for the CBE
- Product: Fedora Core 4
- Product: SDK installation script

Resources

Learn

- This introduction to compiling for the Cell Broadband Engine tutorial series is based on a presentation by IBM Research originally given at PACT 2005.
- The IBM Research Octopiler is neither your grandfather's compiler, nor your grandfather's Oldsmobile.
- You will also be interested in "Unleashing the power of Cell Broadband Engine: A programming model approach" (developerWorks, 2005).
- The Cell Broadband Engine documentation section of the IBM Semiconductor Solutions Technical Library lists specifications, user manuals, and articles of general interest -- including the SPU Instruction Set Architecture documentation.
- Learn more about OpenMP: a portable, scalable, cross-platform model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.
- The IBM developerWorks Cell Broadband Engine resource center is your destination for all things Cell BE.
- Peruse the developerWorks Power Architecture technology zone archives.

Get products and technologies

- Get Cell BE-related downloads, including the IBM Full System Simulator, an evaluation copy of the Visual Age XL C compiler for Cell BE, and the Cell BE SDK from IBM alphaWorks.
- Download a copy of the GCC compiler for Cell from the Barcelona Supercomputer Center.
- Get custom: IBM Engineering & Technology Services can help you with Cell BE- and custom-processor based solutions.
- Find more Power Architecture-related downloads in one page.
- Get a free subscription to the Power Architecture Community Newsletter.

Discuss

- Participate in the discussion forum for this content.
- Send a letter to the editor.

About the author

Power Architecture editors

The developerWorks Power Architecture editors welcome your comments on this article. E-mail them at dwpower@us.ibm.com.

Trademarks

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc.