

An introduction to compiling for the Cell Broadband Engine architecture, Part 4: Partitioning large tasks

Divide and conquer: Parallelizing partitioned jobs

Skill Level: Intermediate

[Power Architecture editors](#)
developerWorks
IBM

07 Feb 2006

This tutorial, fourth and penultimate in the "[An introduction to compiling for the Cell Broadband Engine™ architecture](#)" series, discusses ways to partition code to run across the multiple cores available in a Cell Broadband Engine™ (Cell BE) processor. It gives particular attention to efficient partitioning of code to allow larger programs or data sets to be manipulated using the 256KB of local store available on the Synergistic Processor Elements (SPEs).

Section 1. Before you start

About this tutorial

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.

Prerequisites

See the previous parts in this series:

- [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

And wrap up the series with 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

Section 2. Programming to exploit Cell BE features

Partitioning and parallelization

Exploiting the performance of the Cell BE processor requires efficient use of a broad array of resources. To use the SPEs, the application must be partitioned into components handled by the Power Processor Element (PPE) and components run on the SPEs.

Partitioning can be done entirely manually, entirely automatically, or based on user directives (such as `#pragma` directives).

Programming the PPE and SPEs

The PPE can be targeted with C or Fortran, both of which have intrinsics to access VMX functionality. Furthermore, the compiler can automatically vectorize some code to take advantage of VMX, even without programmer intervention.

Code for the SPEs can be written in any language supported by the compiler; the downloadable releases of XL and GCC support C and C++, or hand-written assembly. In higher level languages, auto-vectorization is possible, or the developer can use SPE intrinsics to write vector code manually.

Parallelizing across multiple SPEs is possible, and necessary to get maximum performance; this requires using appropriate programming models, with or without

help from the compiler.

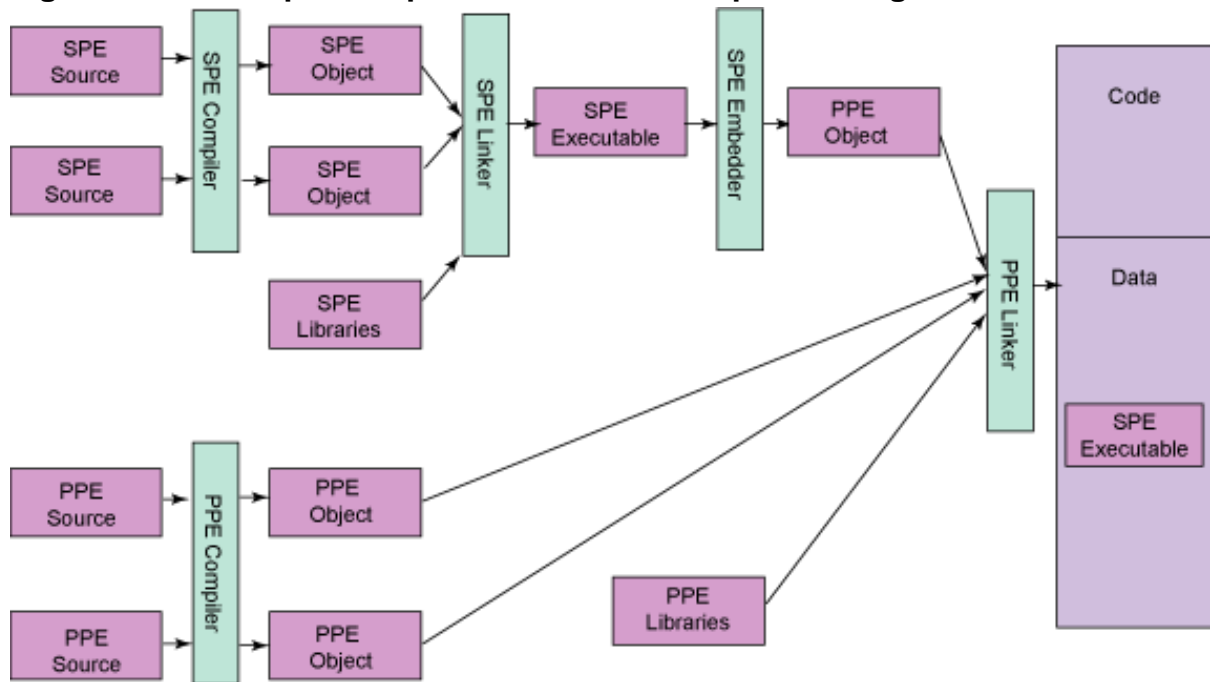
Section 3. Manually programming the Cell BE

It is possible to manually partition work between the PPE and SPEs. For instance, if you have an easily vectorized operation in the middle of some source, it's not hard at all to write an SPE version of it and just manually offload the function to the SPE. This section reviews considerations of manually controlling the SPEs and PPE, to introduce the points where a compiler toolchain can help.

Manual compilation and binding

The following figure illustrates how a manually partitioned and organized program could be built using separate compilers and linkers for SPE and PPE code.

Figure 1. The compilation process with manual partitioning



How manual compilation works

Source specific to the SPEs is compiled into objects, linked together with libraries, and built into SPE executables by a linker. An additional phase then converts these

executables into embeddable objects for use by the PPE linker. On the PPE side, source is once again compiled into objects, and linked against libraries; however, the linker also incorporates the embeddable objects created by the SPE embedder. The result is a normal executable, containing code and data, but some of the data is actually SPE executables.

Performance issues: data transfer

Data transfer is one of the key issues in getting reasonable performance out of the Cell BE. Communication and computation have to overlap for efficient use of resources; otherwise, the processing elements are idle while waiting for the bus, and the bus is idle while waiting for processing. Transfers from system memory have latency which swamps the actual time required for calculation, so efficient operation requires prefetching. The limited size of local store on each SPE requires streaming of data in and out in many cases, which in turn requires reuse of buffer space.

Performance issues: parallel exploitation

The Cell BE processor provides multiple levels of parallelism. The four-way Single Instruction Multiple Data (SIMD) operations used on both the PPE and SPEs allow for many tasks to be parallelized. Furthermore, it is possible to parallelize tasks between the SPEs and the PPE. Effectively using the Cell BE processor requires a strategy for maximizing the use of all eight SPEs, and effective use of vector operations.

Data transfer: size

Identifying the ideal buffer size requires some consideration. Too large a buffer will impose unacceptable latency in filling the buffer and consumes too much of the limited storage available in local store. Too small a buffer leaves the processor waiting for new transfers to begin. Memory operations have significant latency, but high bandwidth, so small transfers are often inefficient.

The best buffer size is determined by the active set of arrays or streams of data, by the "window," or range of indexes, in use in each of them, and finally by the amount of processing needed for each element. Simpler loops might benefit from larger data sets to work through between memory accesses.

Data transfer: index range or "window"

As an example to explore the concept of the "window," consider the following loop:

```
for (i = 0; i < n; ++i)
    A[i] = B[i] + B[i+10];
```

This operation needs a larger "slice" of array B available at a time than it does of A.

Data transfer: timing

The ideal time to transfer data depends on many factors. The following code sample processes a large number of array elements:

```
for (i = 0; i < 100; ++i) {
    /* request 10000 elements of A here? */
    for (j = 0; j < 100; ++j) {
        /* request 100 elements of A here? */
        for (k = 0; k < 100; ++k) {
            x = i * 10000 + j * 100 + k;
            ... = A[x];
        }
    }
}
```

Requesting 10,000 elements of A at the first comment might consume too much storage space for too long, squeezing tight resources. Requesting 100 at the second comment might leave too little overlap between memory access and computation. The best choice primarily depends on available buffer sizes.

Double buffering

One key strategy in maximizing the overlap of computation with memory transfers is double buffering. This is especially important with the SPEs, where main memory access is fast but has large latency. Instead of waiting for a buffer to fill, then processing it, then waiting for it to fill again, you can have one buffer fill while you process another, then process the first buffer while the second fills. The same techniques can be used on output; fill one buffer while another is being copied out to memory.

Manual double buffering

The following code shows a manually coded double-buffering approach on a fairly simple loop. The A and B arrays are now replaced with four buffers, two for A and two for B, each holding 100 items.

```
get B1;
wait B1;
for (j = 0; j < 100; ++j) {
    if (j even) {
        get B2;
        for (i = 0; i < 99; ++i)
            A1[i] = x * B1[i] + B1[i + 1];
        wait B2;
        A1[99] = x * B1[99] + B2[0];
        put A1;
        wait A2;
    } else {
        get B1;
        for (i = 0; i < 99; ++i)
            A2[i] = x * B2[i] + B2[i + 1];
        wait B1;
        A2[99] = x * B2[99] + B1[0];
        put A2;
        wait A1;
    }
}
/* report success to PPE */
```

Section 4. How can the compiler help?

The above overview of issues a developer might face in manually partitioning work onto the Cell BE suggests a few places where the compiler could, in principle, help dramatically. For instance, the compiler can make parallelism easier in many ways, from autovectorization of code, to sorting out which code needs to run on which SPEs, to helping with data caching and buffer management. The EIB has huge bandwidth, but some thrashing can occur, and a loaded system will have some latency; the developer can manually try to ensure that data are prefetched, but the compiler can also help.

Partitioning and parallelization

Parallel systems have always presented a complex programming task. Heterogeneous processors, such as the Cell BE processor, increase the complexity of this task. A partitioning approach needs to consider the characteristics of the different processors. Some code will run faster on the PPE than on the SPEs, and vice versa.

The processors in the Cell BE have different processor performance, different system services (which is to say, none on the SPEs), and different memory latencies. Because the local store for each SPE is not a shared part of main system memory, the compiler has to keep track of locality of code and data; furthermore, the

small size of local store, contrasted with main memory, imposes different requirements on SPE code.

One useful concept is to view the PPE as primarily a service provider, with the bulk of actual computation happening on the SPEs.

One way or another, the compiler can help alleviate the challenge of developing for a particularly complicated architecture.

Section 5. Possible parallel models for the Cell BE processor

When targeting the Cell BE processor, consider two basic approaches. One is to build a model around the PPE, with a single thread which uses two layers of parallelization (one eight-way, on top of four-way SIMD on each processor). In effect, the top-level threads have 32-way parallelization. This model is similar to the loop-level parallelization done for OpenMP. The other is a programming model centered around the SPEs, with eight threads on each Cell BE processor (one per SPE), each of which has four-way SIMD instructions available. This model, an extension of SIMD instructions, is sometimes called a Single Program Multiple Data model. More detailed discussion of these models follows.

The PPE-centric programming model

This model uses a combination of SPMD (Single Program Multiple Data) and SIMD parallelism. Each PPE has a single SPMD thread, and is considered to be the controlling execution engine; the SPEs are treated as a 32-way vector unit. (Although each Cell BE processor has only one PPE, a system could have multiple processors.)

Each SPMD thread in turn manages a group of eight SPE threads. These threads execute chunks of vectorized loops, which are outlined and packaged into an SPE binary. Work is managed on a work queue model; SPEs take new items off the queue as they have resources.

In this model, the program is treated as running on the PPE, with all data in system memory. Code for SIMD loops (for the SPEs) is outlined, and data for them is transferred as needed.

The SPE-centric programming model

In this model, parallelism is primarily exploited by using the SPMD model. Each SPE in the Cell BE processor gets a separate SPMD thread. The SPEs are considered the major execution engine. The PPE is treated as a service provider for the SPEs, while all of the serious computation happens on the SPEs. DMA is used as much as possible for communication between SPEs.

Each SPMD thread in this model has both a PPE component, and an SPE component; these cooperate to execute what is essentially a single program. The PPE component is a Pthread, and the SPE component is an SPE thread. The components require a communication protocol to provide system services to the SPE.

The whole program is treated as running on the SPE. Code partitioning is used for size and functionality. The default location of data is system memory, and local store is used as a software-managed cache.

Section 6. A prototype parallel compiler framework

The prototype Cell BE compiler uses existing compiler infrastructure to partition for functionality. This approach is extended to take advantage of parallelization opportunities offered by the SPEs. A fully automatic approach would face pitfalls, so the user is able to guide partitioning with directives (such as `#pragma` directives) telling the compiler what to do. Partitioning starts with a focus on function, not code size, because most loop bodies will fit comfortably within SPE storage.

The existing compiler supports C and Fortran, so those are the first languages supported. This section discusses the research work being done based on the XL C compiler; other compilers might do some of this differently, but the essential approach taken to compiling for the Cell BE processor should be applicable.

User directives

The user directives adopted to communicate with the Cell BE compiler were based on OpenMP ([see Resources](#)) because of its reasonable acceptance in the parallel programming community. The directives and clauses map easily to parallel execution on SPEs. The `parallel_for`, `parallel_sections` (with possible pipelining functionality) and `parallel_regions` map well onto the Cell BE architecture.

A relaxed memory consistency model is incorporated into the softcache approach, which uses part of the local store to provide a cache of items stored in main memory (see [Part 5](#)).

In addition to user directives, the compiler has existing support for automatically parallelizing some code; the user directives are a way to hint to the compiler about where to apply these algorithms. The initial set of user directives served as a good way to evaluate more generalized user pragmas. This serves as a stepping stone to better, fully automatic parallelization.

Compiler framework

The XL compiler already had a robust infrastructure for much of the new work. OpenMP and auto-parallelization are currently supported in product compilers. The support for dependence analysis, interprocedural analysis, and profile-directed feedback are all very useful; for instance, dependence analysis can feed into the work needed to optimize SPE instruction chains.

The XL compiler already has aggressive loop optimization and loop restructuring; these features are directly relevant to a vectorizing compiler potentially targeting the Cell BE architecture. It has support for partitioning functions for size, or by target.

The XL compiler already supports multiple input languages and targets a variety of different architectures, which is very useful when trying to add support for a new architecture.

There are some new implementation details to cope with, such as the SPE's 128 vector registers and single register class.

The next few panels refer to [Figure 2](#) below. The following terms and acronyms are used:

.code

Wcode/Wcode+ - An intermediate code representation used by the IBM compiler.

FE - Front End. A compiler for a given language that produces intermediate code in the Wcode format.

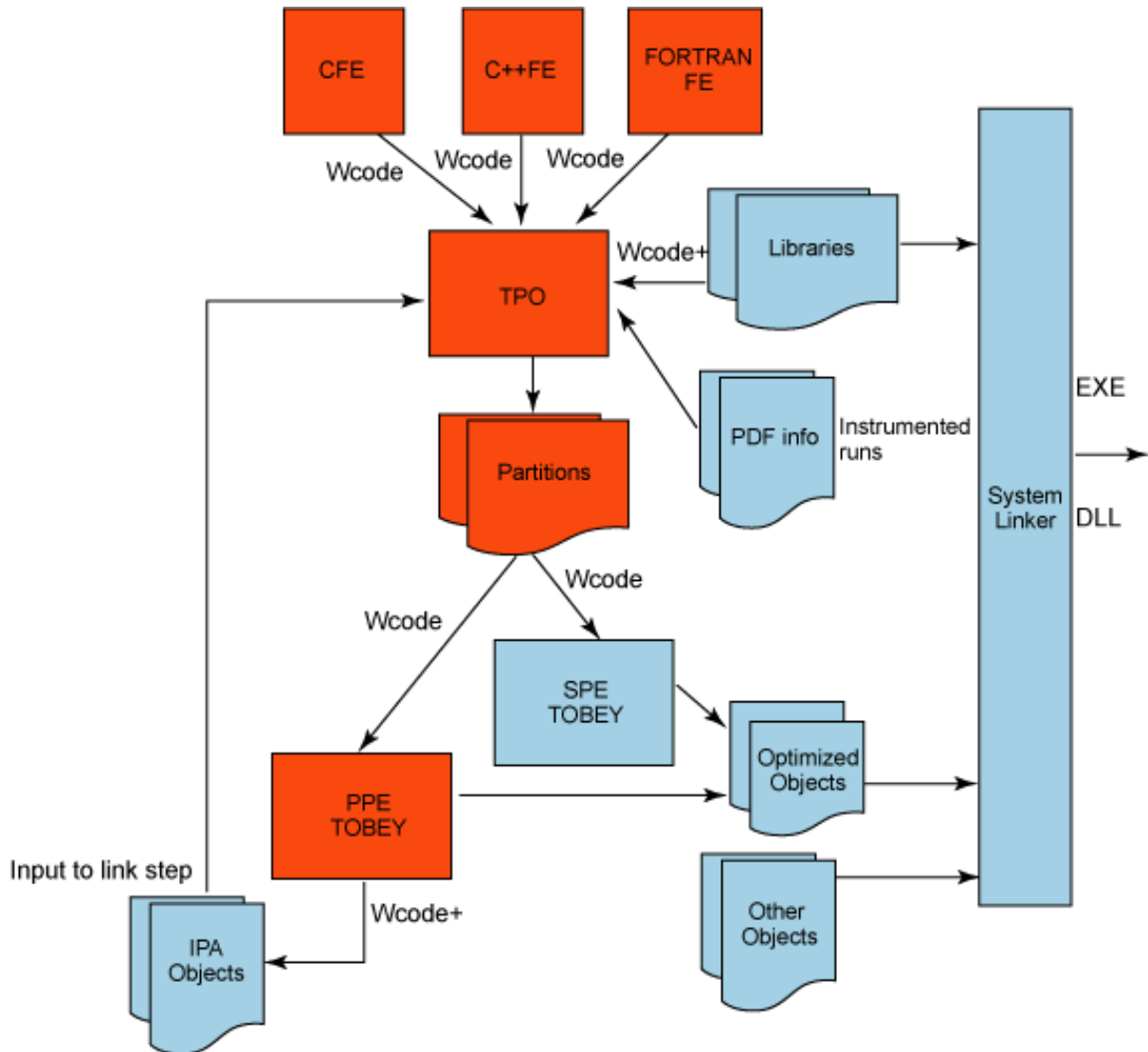
TOBEY - The part of the IBM compiler which deals with some of the special requirements of instruction scheduling for pipelined and multi-issue processors. (Different for PPE and SPE.)

TPO - Toronto Portable Optimizer. The central optimizer of the XL compiler.

IPA - Interprocedural analysis

.endcode

Figure 2. The single-source compiler framework



More about the compiler framework

These are the thousand words the previous figure is worth. The front ends (C, C++, and Fortran) generate intermediate code, called w-code. More w-code is brought in from libraries, and all of this code, along with performance data from instrumented runs, is fed into the TPO phase. This generates partitioned w-code to be fed to the SPE and PPE TOBEY optimizers. The PPE code is, in some cases, run back through interprocedural analysis to give input to the linking pass of the TPO optimizer. The various optimized outputs are put into object files, then linked together.

The TPO passes, in more detail

There are two TPO passes. In the compile pass, the code generated by the front end is run through an Asynchronous Software Thread Integration (ASTI) pass before being passed to the TPO pass. The TPO pass generates w-code, which TOBEY modifies slightly to reschedule it.

In the link pass, whole segments of already optimized w-code are reevaluated and filtered again, and then passed through TOBEY to produce optimized object files. The linker links the various object files together.

A single-source compiler

The IBM research compiler is able to run as a single-source compiler, although this functionality is not yet established in distribution versions of XL C and GCC.

With the single-source compiler, rather than writing separate SPE code and PPE code, the user writes code targeting the Cell BE architecture. The compiler, guided by these directives, takes care of partitioning code between the PPE and SPE. Data transfers between the processor elements are handled entirely by the compiler. The compiler identifies accesses to system memory in SPE functions and uses software caching or static buffers to transfer this data to and from SPE local stores as needed.

The compiler is in charge of code size considerations. Code partitioning might be extended some to handle this based on functionality, rather than just size.

An example of single-source execution

As an example of how this works, consider the following code fragment:

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

Compiled using the single-source compiler, this would end up generating code to run on the SPEs using a software cache for the array contents, something like this:

```
for (k = LB; k < UB; ++k) {
  /* DMA 100 B elements into B' */
  for (j = 0; j < 100; ++j) {
    A'[j] = cache_lookup(x) * B'[j];
  }
  /* DMA 100 A elements out of A' */
}
```

Section 7. Partitioning using OpenMP

Overview

A single-source program contains C or Fortran with user directives or pragmas hinting at partitioning. The compiler outlines all code within the pragmas into separate functions compiled for the SPE, then replaces the outlined code with a call to the parallel runtime, and compiles this wrapper code for the PPE. The master thread continues to execute on the PPE and participate in workshare constructs.

The PPE runtime places outlined functions on a work queue containing information about the scope of a job (such as number of iterations to execute, or the size of data set to work on in each iteration). The runtime has up to eight SPE threads to pull work items (outlined parallel functions) from the queue and execute them on SPEs. The PPE can then wait for an SPE task to complete, or continue executing code while the SPEs work.

The outlining process

Outlining happens early in pass one of TPO, right after intra-procedural optimization. It does not depend on knowledge of the target processor or processors. Instead, it creates an extra node in the call graph. There must be no jumps outside the region being outlined (interprocedural jumps). Procedures are nested to access shared variables. Alias sets for outlines must accurately represent uses and definitions, otherwise later optimizations might move (or eliminate) loads and stores.

Some variables might be turned into automatics for the new procedure, which has to transfer alias relationships.

The cloning process

After a function has been outlined, the next stage is cloning. In pass two of TPO, the outlined function is cloned, and one version is specialized into a PPE version, and one into an SPE version. All called functions must also be cloned, and SPE call sites are modified to call the SPE versions of cloned subroutines.

The partitioning pass creates SPE and PPE partitions and invokes a lower-level optimizer for machine-specific optimization.

Cloning outlined functions for heterogeneous processors

The cloning process assumes that outlined functions are nested, for instance, that local variables may be found in the parent's stack frame. For heterogeneous cloning, that means accessing the stack of a separate processor. Even if you had the system memory address of the parent stack frame in the PPE, you couldn't compute the offset when executing in a different compilation path.

To resolve this, all such references are identified and gathered into a structure (within which offsets are known), and this structure can have its address computed at runtime and passed as a parameter to the SPE routine, which can then use the softcache code to access these variables in the SPE function.

More about cloning

Cloning has substantial implications. The same techniques that potentially allow the same function to run simultaneously on the PPE and an SPE could allow it to run simultaneously on six or eight SPEs as well. With no programmer-visible change in the code, there's potential for a huge amount of parallelism.

PPE runtime

The first OpenMP construct sets up the runtime system. This involves creating SPE threads and a work queue, and getting the DMA queue addresses. The address of the work queue is sent to each SPE, and global options are set. After the partitioning and scheduling setup is complete, a `setup_done` signal is sent to the SPE runtime.

Each OpenMP parallel/work-sharing construct invokes the runtime system. The PPE itself calls outlined procedures to do its part of the work. The runtime uses barrier synchronization.

SPE runtime

The SPE runtime is much simpler than the PPE runtime. It sits in an infinite loop waiting for signals from the PPE runtime. When there's work to be done, the runtime fetches work items from the work queue, using DMA. Depending on the work type, it then either invokes an SPE-outlined procedure directly, or translates the address of an SPE-outlined procedure from a PPE-outlined procedure. Once again, a barrier object handles synchronization.

Communicating between the PPE and SPE runtime

The PPE uses DMA signals to communicate work to be done to the SPEs. The SPEs use mailboxes to notify the PPE of the completion of tasks. The PPE takes items off the work queue when it doesn't have any management tasks to handle, helping share some of the load.

Items in the work queue indicate the function to be run, the lower and upper bounds, and the work type to be done; the SPEs pick these items up and run them, then go back to checking the queue again.

Runtime interaction

The PPE runtime is responsible for partitioning and scheduling decisions, and handling synchronization. So, PPE functions tend to be stubs which set up the work queue, and then outlined functions get executed on the SPEs or on the PPE as appropriate.

Limitations

This design doesn't really provide for nested parallelism. Parallel constructs with system calls end up serialized, because system calls execute on the PPE. The aggregated SPE binary isn't partitioned yet. In general, some features are still under development.

Tune in next time

This tutorial referred in passing to techniques for sharing memory between the SPEs and main memory (and potentially each other). The [next tutorial](#), the last in the series, takes a detailed look at some of the memory management techniques and code developed for this, such as the softcache code.

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.

This Part 4 is based on the section "Parallelization and partitioning in the Single

Source Cell compiler."

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.
- 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-related downloads](#) including the IBM Full System Simulator, an evaluation copy of the Visual Age XL C compiler for Cell, and the Cell 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- 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.