



Building Signal-Processing Applications for the Cell Broadband Engine Processor – Without Sacrificing Performance or Portability

How to use a high-level API to write less code, get better performance, and develop reusable algorithms

Because the performance of signal- and image-processing (SIP) applications is limited by the speed of floating-point processors, the Cell Broadband Engine™ (Cell/B.E.) processor – a “supercomputer on a chip” – with very high floating-point performance potential has attracted tremendous interest in the SIP community.

However, the complex, unconventional nature of the Cell/B.E. architecture makes taking advantage of its performance potential difficult. Conventional programming techniques force software developers to write substantially more code to fully utilize the Cell/B.E. processor than they would for other processors. Furthermore, applications developed for other processors are difficult to move to the Cell/B.E. processor, and vice versa. This combination of increased complexity and lack of portability has created a perception that adopting the Cell/B.E. processor may bring excessive costs, despite the

substantial performance benefits.

Fortunately, by adopting a standardized high-level application programming interface (API), organizations can simultaneously take advantage of the Cell/B.E. processor’s performance, sidestep the architectural complexity of the Cell/B.E. processor, and maintain application portability between the Cell/B.E. processor and other processors.

This white paper describes the Cell/B.E. architecture, explains how to use a high-level API to conveniently take advantage of the Cell/B.E. processor’s performance potential, and then describes how an important radar-processing computation was ported to the Cell/B.E. processor using that API. This computational kernel had been developed for a conventional processor. A ten-fold performance improvement was realized when this application was ported to the Cell/B.E. processor – with no changes to the application code.

1. The Cell/B.E. Architecture

The Cell/B.E. architecture is a novel processor architecture developed jointly by IBM, Sony, and Toshiba to address the memory, frequency and power limits reached by conventional processor design. The Cell/B.E. processor uses an asymmetric multi-core architecture to provide increased processing power and a multi-tiered memory architecture to accelerate data movement, without increasing clock frequency.

One general-purpose processor, the Power Processing Element (PPE), performs operating system and control tasks. Eight additional processors, the Synergistic Processing Elements (SPEs), are specialized high-performance computational engines. A high speed on-chip bus connects the processors to each other and to high-performance memory and I/O interfaces.

Computational Power

Because the PPE uses the standard Power ISA, it is compatible with other popular processors, such as the IBM PowerPC 970. By contrast, the SPEs use a new instruction set, designed to achieve very high floating-point performance and 128 registers, each of which is 128-bits wide.

SPE arithmetic is performed via Single Instruction Multiple Data (SIMD) instructions which can perform as many as eight single-precision floating-point operations simultaneously. Therefore, at 3.2 GHz, the theoretical peak performance of a single SPE is 25.6 GFLOP/s, and all 8 SPEs together can deliver 204.8 GFLOP/s. Including the PPE, total peak performance on the Cell/B.E. processor is a remarkable 230 GFLOP/s using just 80 watts.

I/O Capability

Each SPE has 256 KB of dedicated high-speed local memory (for both program and data) accessible with zero latency. Each SPE has a 25.6 GB/s connection to an internal bus. The internal bus connects the 8 SPEs, main memory, and an external I/O bus. The internal bus has a peak bandwidth of 300 GB/s; the external bus (which can be used to connect two Cell/B.E. processors) delivers 60 GB/s.

The key challenge of programming for the Cell/B.E. processor is making effective use of the eight Synergistic Processing Elements (SPEs).

2. Challenges

Although its asymmetric multi-core design allows the Cell/B.E. processor to deliver a high level of throughput, that design also greatly increases the complexity of programming the architecture. Any multi-core design imposes the usual problems associated with parallel programming, such as efficiently partitioning and balancing computation between CPUs and overlaying computation with communication.

However, the specialized nature of the SPE accentuates these problems. For example, even applications of moderate size may be too large to fit in SPE local memory, forcing explicit transfers between main memory and SPEs. Furthermore, standard programming models, such as the Message Passing Interface¹ (MPI), cannot be used on the SPEs.

Managing Multiple Cores

The nine processors in the Cell/B.E. chip (one PPE and eight SPEs) must operate independently, but in a coordinated fashion, in order to achieve maximum

¹ <http://www.mpi-forum.org>

performance. The PPE is best used as a control processor, managing the complex flow of instructions and data to and from the SPEs, determining which computations to offload to the SPEs, configuring the SPEs to perform them, and then managing their operation. Because the SPEs collectively have much higher direct memory access (DMA) issue rate than the single PPE, it is advantageous for SPEs to manage their own data transfers.

Don't starve the SPEs. With the Cell/B.E. architecture, applications must explicitly stream small chunks of data to SPEs for processing, so it's important to hide communication latency behind computation and fuse operations to eliminate unnecessary transfers between SPEs and main memory.

There are often many ways to factor a particular computation between the PPE and the SPEs. One design might have all eight SPEs running the same application code but working on different portions of the data set. Another design for the same computation might use different SPEs for different portions of the computation, streaming data between the SPEs before returning them to main memory. It is often difficult to tell which approach will be more efficient without implementing both approaches and measuring the results.

Transferring Data

Using the full computational capabilities of the SPEs is only possible if data transfers can be overlaid with computation so that the SPEs are never starved for data. The local store associated with each SPE is not a cache and, therefore, the hardware does not automatically transfer data to or from main memory. Instead, all data transfers

must be explicitly performed via DMA requests issued by the application software. This manual approach improves performance (by simplifying the hardware architecture) and provides predictability (since all local store accesses have uniform access time), but increases programming complexity.

Moreover, MPI itself cannot be directly used from the SPEs due to key differences between the MPI model and the programming model necessary for

managing data transfers to and from the SPEs. MPI is typically used between a set of processors which, between them, share the entire data set. The limited size of the SPE local store means that much of the data may lie in

main memory, rather than being associated with any SPE. This situation demands a "streaming" paradigm, in which data is taken from main memory, sent to the SPEs, processed, and then returned to main memory. This streaming must be done in chunks small enough for the 256K local store, and must be done asynchronously so that communication latency is hidden behind computation.

Another important difference between Cell/B.E. processor and MPI programming is that in a typical MPI application, the operating system automatically manages loading application code into system memory, using the virtual memory system if necessary. On the other hand, a Cell/B.E. application uses the PPE to ensure that the right application code is available on the SPEs. As a result, the programmer often loads one set of routines to the SPEs, processes data, and then loads another set of routines to perform the next processing phase.

Fusing Operations

In order to avoid “starving” the SPEs, programmers need to eliminate unnecessary memory traffic between the SPEs and main memory. For example, consider the case in which two successive operations must be performed on the same data. The simplest approach is to stream the data to the SPE, perform the first operation, return the data to main memory and then repeat that cycle again. Unfortunately, this forces all of the data to move between main memory and the SPEs twice. By “fusing” the first and second operations on the SPE, the amount of data transferred can be cut in half. Because there are a large number of basic operations and because one can fuse more than two operations, the number of possible combinations is vast.

Maintaining Portability

New hardware architectures, such as reconfigurable or polymorphic computing technologies, massively multi-core technologies, GPU accelerators, and other more exotic possibilities all loom on the horizon. Meanwhile, much of the production software running today can trace its lineage back for decades. As a result, many users considering the Cell/B.E. processor have millions of lines of code which may also run in other environments.

The high rate of change in hardware platforms presents application developers with a dilemma: either specialize for maximum performance on today’s silicon or generalize for portability to future architectures. The problem is compounded by conflicting goals of minimizing system costs by fully utilizing hardware capability, and of minimizing software development costs by avoiding

the need to completely rewrite the software for each technology refresh. Maintaining portability and scalability without sacrificing performance is critical.

Performance vs. portability. The goal of getting the most out of hardware is often in tension with the goal of reusing software.

Software reuse is an increasing priority for many major enterprises, the Department of Defense, and other developers. Engineers have recognized that open standards, i.e., freely available specifications that multiple parties can implement, greatly facilitate reuse. For example, by developing applications using MPI, rather than proprietary protocols, the same application can be much more easily moved from one computing platform to another.

Clearly, code specifically targeted for the Cell/B.E. processor is not readily portable to other platforms, as there is no equivalent to the SPEs on a conventional processor. Therefore, when bringing an application written for another environment to the Cell/B.E. architecture, substantial recoding may be required to take advantage of the SPEs. Likewise, an application for the Cell/B.E. processor may need to be drastically reworked to run on conventional processors.

3. Strategies

After considering these challenges, designers must decide how to attack the problem of programming for the Cell/B.E. processor. The most appropriate strategy depends on the fundamental objectives for the project. Is it vital to squeeze every possible bit of performance from the hardware? Or is it more important to have an upgrade path to future processors? Does the budget for the

project allow extensive tuning, or is it important to get good performance with relatively little optimization? In this white paper, we consider three strategies, which are not mutually exclusive.

Direct Access

It is possible to program every detail of the Cell/B.E. processor directly from the application code. Applications can manage communication and computation explicitly, using DMAs and code tuned for the SPEs. Programming the Cell/B.E. processor at this level is analogous to using assembly language on a conventional processor. Because all programs are eventually translated to assembly language, this strategy provides the highest possible performance. However, the costs of writing in assembly language are very high: programmers are far less productive than when writing in a high-level language, portability is entirely lost, and many of the available tools for assisting programmers are simply unavailable. Furthermore, modern

The best strategy for programming the Cell/B.E. processor may be to start with a high-level API, take advantage of the IBM SDK, and use direct, low-level programming of the Cell/B.E. processor only for select, performance-critical code sections.

compilers can often generate code superior to that written by most programmers.

Similarly, on the Cell/B.E. processor, programming at the lowest level clearly could deliver the best possible performance. However, many programmers will find achieving that potential both challenging and time-consuming. As with programming in assembly language, programs written in this way will not be portable to other

systems.

A low-level approach also forces early decisions on how data and processing will be partitioned and mapped onto the architecture. These decisions become hard-wired into the application and are difficult to change later. As a result, this strategy may limit performance as well as portability. If the application is not sufficiently understood during development, the choices made may be suboptimal. The high cost of changing decisions makes exploring tradeoffs prohibitive.

Therefore, most applications should use these low-level techniques only for performance-critical code sections where higher-level techniques have proven unsatisfactory.

IBM Software Development Kit

The second strategy is to use the IBM Cell/B.E. Software Development Kit² (SDK). The SDK provides libraries for computation and communication, including the Accelerated Library

Framework (ALF) for writing data-parallel applications. ALF uses a SPMD task model, running a single computational kernel on all of the SPEs to process a work queue of data. ALF handles data transfers, overlays computation with

communication, and manages concurrency, allowing the application to focus on mathematical functionality. The SDK also includes a cycle-accurate simulator, compilers, sample applications, and other development tools.

The libraries in the SDK provide some benefits relative to the “do-it-yourself” approach described above. In particular, ALF can eliminate much of the difficult DMA programming required.

² <http://www.ibm.com/developerworks/power/cell>

Applications written using the SDK are still tied to the Cell/B.E. processor, however, and portability to other systems remains a substantial challenge. Finally, because the SDK is a general-purpose toolkit, the computational primitives provided are focused on general-purpose mathematics, rather than those specifically needed for SIP applications. Although some basic SIP operations, such as basic fast Fourier transforms (FFTs) are included, the set of available primitives is not sufficient for most SIP applications.

High-Level APIs

SIP application designers already face substantial complexity merely in constructing the complex mathematical algorithms required to process data. Translating these algorithms into robust, efficient code – for any processor – is a formidable task. When the processor in question is the Cell/B.E. processor, programmers must be experts in SIP algorithms and in the complexities of the architecture.

What if developers could express the algorithms in terms of their basic primitives and let the computer do the rest? Algorithm designers think of their algorithms as a series of FFTs, linear algebra operations, and other similar operations. Why not simply write code at that level?

The third strategy – using high-level APIs – is exactly the approach taken by high-productivity solutions, such as MATLAB®.³ These “scripting” environments allow designers to readily experiment with a variety of technical approaches and to quickly determine whether new algorithms yield better results. On the other hand, scripting

languages are often unsuitable for real-time, mission-critical deployments due to high memory requirements, limited performance, and unpredictable execution times.

Therefore, a solution that leverages conventional, pre-existing programming languages (and their attendant compilers, debuggers, and profiling tools) and allows high-level programming can provide a number of key advantages. Because the programmer uses a high-level API, rather than writing explicitly for the SPEs, the application is much easier to write and remains portable to other processors. Furthermore, if the API can efficiently dispatch operations to the SPEs, fuse adjacent operations, and automatically manage data transfers, programs written using this strategy can perform as efficiently as those written using lower-level interfaces.

Advantages of a high-level API:

- Achieve superior performance
- Get to market more quickly
- Preserve portability

4. The VSIPL++ API

The VSIPL++⁴ API is an open standard for high-performance signal- and image-processing defined by the High Performance Embedded Computing Software Initiative (HPEC-SI), a forum of industrial, academic, and governmental partners, with sponsorship from the U.S. Department of Defense.

The VSIPL++ API defines a pure C++ interface for operations including FFTs, filters, solvers for systems of linear equations, and other operations useful in developing radar, sonar, and medical imaging applications.

Because the VSIPL++ API is an open standard, the API specification is

³ <http://www.mathworks.com>

⁴ <http://www.vsipl.org>

freely available and anyone may implement the API. A portable (but unoptimized) “reference implementation” of the API runs on most systems with a C++ compiler, making it easy for users to experiment with VSIP++.

The VSIP++ API is an open standard designed to deliver productivity, portability, performance, and parallelism.

The API design goals were to simultaneously deliver “the three Ps” – productivity, portability, and performance. When using VSIP++, programmers write less code, and the code written is free of low-level system details. Furthermore, the API was carefully designed to allow implementations that use sophisticated techniques, like compile-time and run-time operation dispatch, to deliver maximum performance.

The VSIP++ API also provides direct support for a fourth “P”: parallelism. Unlike VSIP, which was targeted at uniprocessor systems, the VSIP++ API includes support for data distributed across multiple processors.

Programmers develop VSIP++ applications using a Single-Program Multiple-Data (SPMD) model. Each processor operates on data available to it locally, while the VSIP++ implementation moves data between processors. Thus, users are freed from writing MPI code. In many cases, serial applications can be converted to parallel applications just by changing data declarations to indicate how the data should be distributed among processors.

VSIP++ Views

A guiding principle in the design of the VSIP++ API was the separation of logical

and physical attributes of data. As a result, the core algorithm used (i.e., the SIP computations performed) can be separated from details of implementation (such as whether data resides on one or multiple processors).

As an example, consider the radar data collected during a single Coherent Processing Interval (CPI). Logically, this data is a set of `n_pulse` pulses, each containing `n_cells` range cells. A range cell value represents the phase of the returned energy as a complex frequency.

In VSIP++, this data would be represented by a matrix *view* declared as follows:

```
typedef complex<float>
    value_type;
Matrix<value_type>
    data(n_pulse, n_cells);
```

The matrix view encapsulates the *logical* description of the data: it has 2 dimensions, the first dimension is pulses, the second dimension is cells, and the values stored are complex, single-precision values.

Signal processing functionality can now be written to process data. For example, this sequence of statements:

```
Fftm<value_type, value_type,
    col, fft_fwd>
    fftm(Domain<2>(n_pulse,
        n_cells),
        1.0);
data =
    fftm(vmmul<col>(hanning
        (n_pulse),
        data));
```

implements a Doppler filter by first applying a Hanning window to the returns

in each column of range cell returns, then using a multiple-FFT object to transform the returns into Doppler frequencies.

VSIPL++ Blocks

By default, the data in a VSIPL++ Matrix is stored as a row-major array on a single processor. Complete algorithms can be developed without changing this data representation.

However, once the algorithms have been verified, it is possible to optimize application performance by considering how the data is stored in memory. For example, if the columns from data are being transformed with a Doppler filter to find moving targets, it would be advantageous to store data in column-major dimension ordering. That will ensure that successive accesses to the data occur at neighboring memory locations and thereby improve cache performance.

In VSIPL++, this is done by describing the *block type* used by the matrix view to store data:

```
typedef Dense<2,
             value_type,
             col2_type>
    block_type;
Matrix<value_type,
      block_type>
    data(n_pulse, n_cells);
```

The signal processing code (the actual Doppler filter algorithm) need not be modified after this change is made.

Similarly (and again without changing the signal processing code), the block type could be modified to express different packing (contiguous or row/column aligned), different complex formats (interleaved complex or split complex), and even different distribution options across multiple processors (block

distributed, replicated, and so on).

The separation of logical and physical attributes allows signal processing code to be developed focusing on functionality first, and then optimized by exploring performance trade-offs. Decisions about data layout that typically must be made early in development – before the trade-offs are fully understood – can be made much later, after measuring the different alternatives, when using VSIPL++.

With the VSIPL++ API, programmers can implement the necessary SIP functionality first and then optimize their application by experimenting with data layout.

5. Sourcery VSIPL++™

Sourcery VSIPL++ is CodeSourcery's optimized implementation of the VSIPL++ API. CodeSourcery was chosen to lead the development of the VSIPL++ API and was subsequently awarded a contract by the Air Force Research Laboratory to develop Sourcery VSIPL++. (The VSIPL++ reference implementation is in fact a version of Sourcery VSIPL++ that does not include CodeSourcery's proprietary optimizations.) Sourcery VSIPL++ runs on GNU/Linux systems (including multi-processor clusters), on Power systems running the Mercury Computer Operating Environment, and on Microsoft Windows.

Dispatch Engine

The abstractions provided by VSIPL++'s views and blocks, and the high-level functionality provided by VSIPL++'s routines, create the potential for high-performance. To exploit this potential requires mapping the application code onto the available hardware resources. Sourcery VSIPL++ has a powerful *dispatch engine* that uses the attributes of

the data structures and operations to select, and even create, a high-performance computation.

Dispatch uses compile-time and run-time attributes of the data and operation to choose the highest performing implementation. For example, an operation on data known to be aligned at compile-time can be mapped directly to an implementation using SIMD instructions without incurring overheads to check alignment at run-time.

High-Performance Libraries

The dispatch engine in Sourcery VSIPL++ can take advantage of optimized low-level math libraries, such as the Intel® Performance Libraries⁵ (IPP) or Mercury's Scientific Algorithm Library⁶ (SAL). Operations can be dispatched to existing vendor libraries with near zero overhead.

For example, when dispatching operations to SAL on Power Architecture™ MCOE™ systems, Sourcery VSIPL++ is within 1% of the raw performance of SAL. Dispatch can pick and choose between multiple vendor libraries and organic implementations. As a result, VSIPL++ applications fully leverage the investment made in the

existing vendor stack, without sacrificing portability.

Fused Operations

Sourcery VSIPL++ uses expression templates to recognize opportunities for fusing operations. For example, A VSIPL++ application multiplies two vectors A and B and adds them to a third vector C by writing:

$$A * B + C$$

Sourcery VSIPL++ recognizes this idiom as a fused multiplication-and-addition operation. It can therefore make use of low-level math routines designed for exactly this case. Hand-written low-level math routines are not always available for more complex expressions, such as:

$$A * B + C * D$$

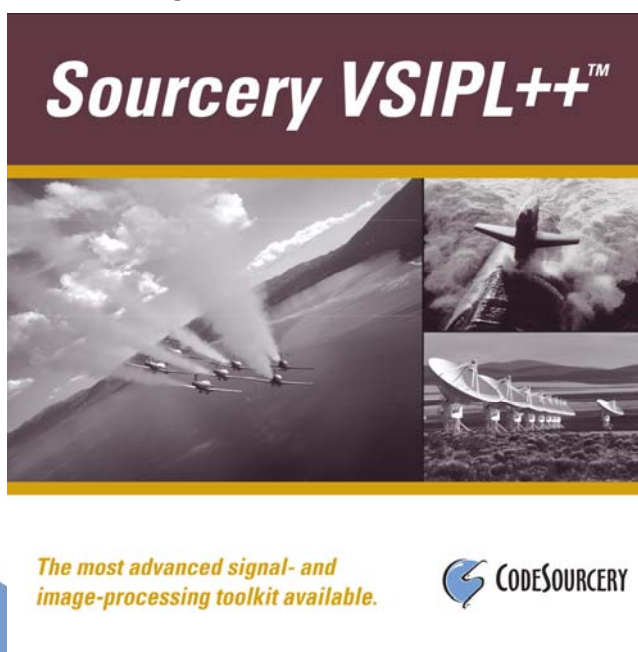
In these cases, Sourcery VSIPL++ uses sophisticated C++ techniques to provide the system C++ compiler with code that interleaves multiple operations. Then, the compiler can generate code that is well-tuned for the target processor.

Multi-Processor Operation

Sourcery VSIPL++ implements the VSIPL++ API specifications for multi-processor machines. On most systems, Sourcery VSIPL++ uses MPI to communicate data between nodes. However, Sourcery VSIPL++ can also use the Mercury Parallel Acceleration System (PAS) on MCOE systems.


6. Sourcery VSIPL++ on the Cell/B.E. Processor

Sourcery VSIPL++ for the Cell/B.E. processor balances simplicity of



Sourcery VSIPL++™

The most advanced signal- and image-processing toolkit available.



⁵ <http://www.intel.com/cd/software/products/sasmo-na/eng/perflib/219780.htm>

⁶ <http://www.mc.com/products/view/index.cfm?id=5&type=software>

programming with optimal utilization of the Cell/B.E. processor's capability. The PPE is used to run the application and to manage computation on the SPEs, which are used as high-performance computation engines. Sourcery VSIPL++ manages use of the SPEs, including double-buffering of communications to hide communication latency behind computation.

Sourcery VSIPL++ for the Cell/B.E. processor maximizes the computational capability of the SPEs and provides linear scalability across multi-Cell systems.

Resource Allocation

Sourcery VSIPL++'s dispatch engine determines which computations run on the SPEs based on a variety of factors, including the layout of data, the operations being performed, and the ratio of computation to communication. Additionally, applications can give Sourcery VSIPL++ hints about SPE resource allocation.

Extensibility

Developers can provide custom SPE computation kernels. Sourcery VSIPL++'s SPE management and streaming allow the developer to focus on computational performance. The Sourcery VSIPL++ dispatch engine allows custom kernels to be used without sacrificing application portability.

Multi-Cell Machines

Sourcery VSIPL++'s data-parallel model allows data to be distributed over multiple Cell/B.E. processors so that computations can be performed in parallel. Sourcery VSIPL++ can seamlessly utilize multiple Cell/B.E. processors, even those not sharing the same address space.

7. Using Sourcery VSIPL++ to Implement Fast Convolution on the Cell/B.E. Processor

CodeSourcery has developed a simple VSIPL++ application which performs a *fast convolution*. Convolutions are used in signal-processing to implement filtering. As expected, because this application uses the VSIPL++ API, no changes to the code were required to move this application between three processors:

Intel Pentium® 4 Xeon®, IBM PowerPC® 970FX, and IBM Cell/B.E. processor.

This section describes the fast convolution algorithm, the VSIPL++ implementation of the algorithm, and compares the performance results that were obtained in each of these three environments.

Fast Convolution

Fast convolution performs computations in the frequency domain instead of the time domain. Because signals can be efficiently transformed between time and frequency domains using the Fast Fourier Transform (FFT), if the convolution kernel is large enough, frequency domain convolution requires fewer operations. Fast convolution for a single vector is computed by transforming both the vector and the convolution kernel into the frequency domain, performing an element-wise vector multiply, and then transforming the result back into the time domain.

In signal-processing applications, fast convolution is typically done on many vectors that together form a data cube. This allows the cost of transforming the kernel coefficients to be amortized.

Because each of the vectors can be operated on independently of the others, it is possible to perform multiple convolutions simultaneously, assuming sufficient computational power is available.

Implementation Using the VSIPL++ API

With Sourcery VSIPL++, fast convolution can be expressed in a few lines of source code using FFTM signal processing objects, which perform multiple FFTs on rows of a matrix.

First, views are declared to hold the data cube and the FFT signal processing objects:

```
typedef complex<float> T;
Vector<T> weights(n_cells);
Matrix<T> data(n_pulses,
              n_cells);
Domain<2> dom(n_pulses,
             n_cells);
Fftm<T, T, row, fft_fwd>
  fwd(dom, 1.0);
Fftm<T, T, row, fft_inv>
  inv(dom, 1.0/
      n_cells);
```

Next, the weights are transformed into the frequency domain via an in-place FFT:

```
fft_ip<fwd_fft>(weights);
```

Finally, the convolution itself is expressed as:

```
data =
  inv(vmmul<row>(weights,
                fwd(data)));
```

This statement performs three steps: First, it uses the fwd FFTM object to transform the rows in matrix data from the time-domain to the frequency-domain. Next, it uses a vector-matrix multiply (vmmul) operation to multiply each row in the frequency-domain data matrix by the

weights vector. Finally, it uses the inv Fftm object to transform data back into the time-domain.

On the Cell/B.E. processor, Sourcery VSIPL++'s dispatch engine recognizes the fast convolution operation expressed by this statement and maps it to a fused fast convolution kernel that runs on the SPEs. This fused kernel streams through data, performing forward FFT, vector-multiply, and inverse FFT one row at a time. Intermediate results are kept within the SPE, eliminating unnecessary memory traffic. This example runs on a single Cell/B.E. processor, taking full advantage of all 8 SPEs (or 16, if another Cell/B.E. processor is connected via the I/O bus).

By modifying the structures to describe how they can be distributed across multiple processors, the same example can take advantage of multiple Cell/B.E. processors connected via a network fabric such as InfiniBand:

With Sourcery VSIPL++, it only takes a few lines of C++ to implement a fast convolution algorithm — even on the Cell/B.E. processor.

```
typedef Dense<2, T,
            row2_major,
            Map<> >
  data_block_type;
typedef Dense<1, T,
            row1_major,
            Global_map<1> >
  weights_block_type;
Map<> map(num_processors());
Vector<T,
        weights_block_type>
  weights(size);
Matrix<T, data_block_type>
  data(rows, size, map);
```

The fast convolution code itself remains unchanged.

Performance on the Cell/B.E. Processor

Even though the application code shown above is entirely portable, and makes no mention of SPEs, DMAs, or other Cell/B.E. technology, it performs very well on the Cell/B.E. processor. As discussed below, this application runs many times faster on the Cell/B.E. processor than it does on conventional processors.

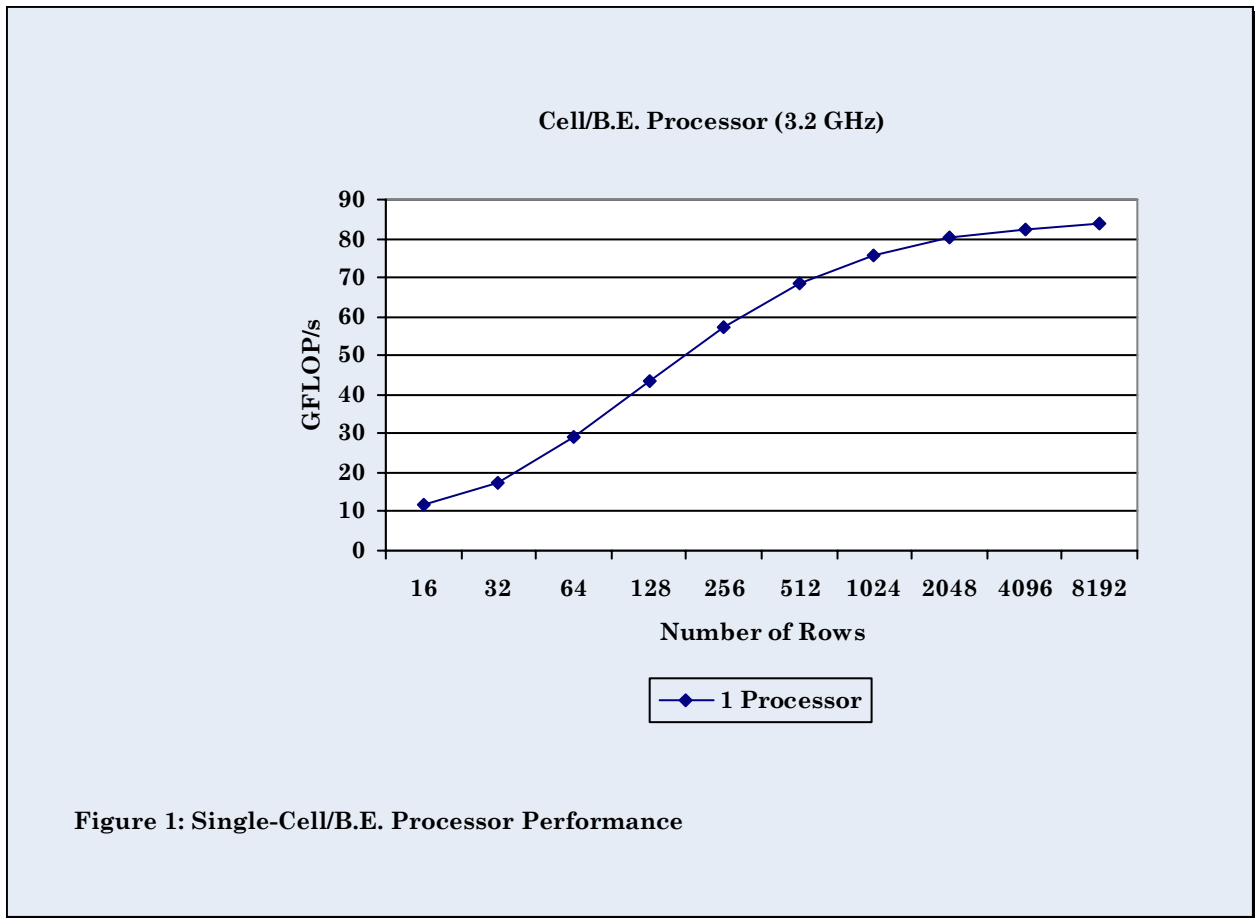
Performance on a Single Cell/B.E. Processor

Figure 1 shows the fast convolution performance on a single Cell/B.E. processor, with a single PPE and all 8 SPEs, using Sourcery VSIPL++ and a modified version of the IBM SDK.

At 4096 rows, 82.6 GFLOP/s is sustained (40.3% of the SPEs' theoretical peak performance of 204.8 GFLOP/s). This corresponds to 11.6 GB/s of memory bandwidth (45.3% of the XDR Interface's peak performance of 25.6 GB/s).

Performance on Multiple Cell/B.E. Processors

Figure 2 compares fast convolution performance for a single Cell/B.E. processor, 2 Cell/B.E. processors located on a single blade in an IBM Blade Server, and 4 Cell/B.E. processors located on two blades. Once data structures have been modified to describe distribution (which introduces almost zero overhead), going from 1 to multiple PPEs is controlled via



run-time parameters to the application.

The 1 PPE (blue) and 2 PPE (pink) lines are identical to those shown in Figure 2. For 4 processors (green), 300 GFLOP/s is sustained at 4096 rows, a 3.6-fold speedup. The speedup is sub-linear only because each Cell/B.E. processor is now processing a smaller, less efficient problem size. When the problem size is scaled with the number of processors, the speedup is over 3.9 times.

Performance on Other Processors

Because the VSIPL++ API is portable, and because Sourcery VSIPL++ runs on multiple architectures, the fast

convolution benchmark runs on other systems as well. A simple recompilation of the application is all that is required.

Figure 3 compares the performance of the fast convolution benchmark running on the Cell/B.E. processor with the same benchmark running on a 3.6 GHz Pentium 4 Xeon and on a 2 GHz PowerPC 970FX processor. On the Pentium 4 Xeon, Sourcery VSIPL++ uses the Intel Performance Primitives (IPP) to implement low-level mathematical operations. On the PowerPC 970FX, Sourcery VSIPL++ uses the open-source Fastest Fourier Transform in the West⁷ (FFTW) library to implement FFTs.

⁷ <http://www.fftw.org>

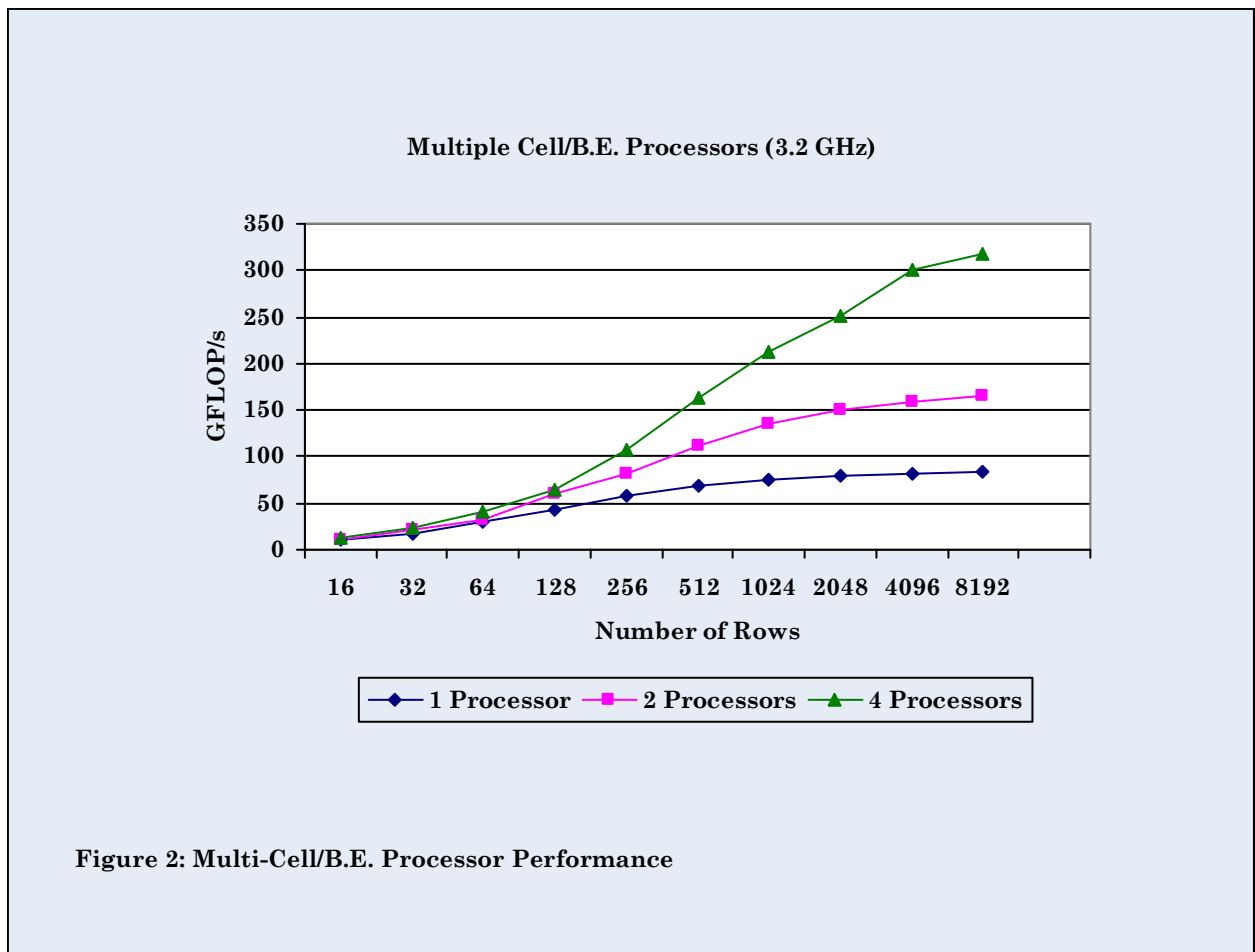
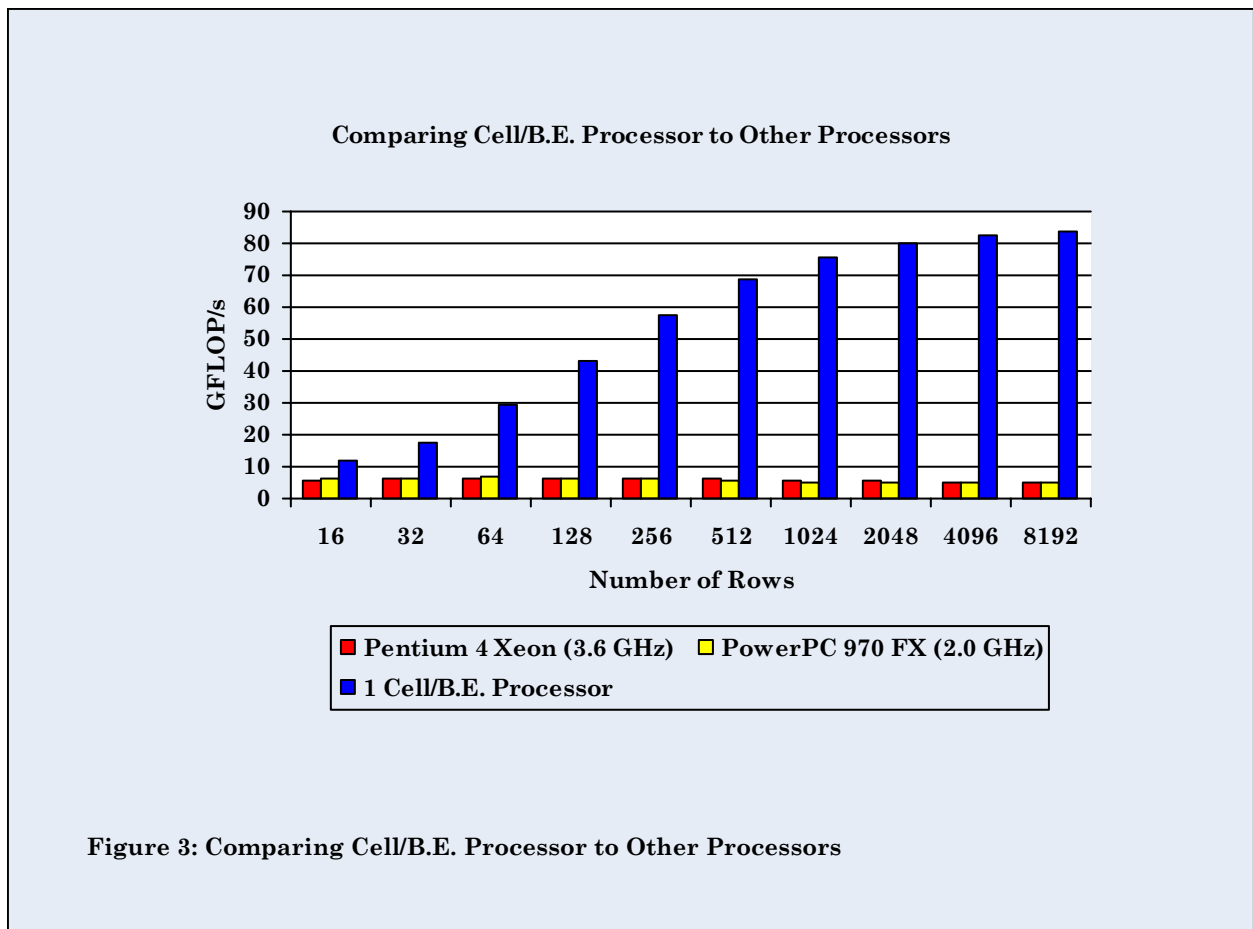


Figure 2: Multi-Cell/B.E. Processor Performance

On the Pentium 4 Xeon and PowerPC 970FX processors, maximum performance is achieved for problem sizes that fit into the available cache. For the Pentium 4 Xeon processor, the best sustained performance is 6.0 GFLOP/s, 41.8% of theoretical peak. (Although hyperthreading is available on these processors, these measurements used only a single thread, as compute-intensive applications generally do not benefit from hyperthreading.) On the PowerPC970 FX, the peak performance achieved is 6.6 GFLOP/s, or 41.2% of theoretical peak.

The Cell/B.E. processor outperforms both of these other processors

in two dimensions. First, the absolute performance is much greater, reaching as high as 82.6 GFLOP/s, more than ten times faster than either of the other processors. Second, while the performance of the Cell/B.E. processor is better at all of the data points shown above, the Cell/B.E. processor does particularly well on larger data sets. The overhead of transferring data to and from the SPEs decreases, relative to the overall computation, as the data sizes increase. In contrast, conventional processors operate less efficiently on larger problems, as these larger problems do not fit into the available cache.



About CodeSourcery

CodeSourcery builds software tools that enable its customers to get the most out of hardware platforms ranging from embedded devices to supercomputers. Sourcery G++, a complete software development environment based on the GNU Toolchain, is the leading choice of software engineers, operating system vendors, semiconductor companies, and device manufacturers. Sourcery VSIPL++, a portable, parallel signal- and image-processing toolkit, boosts performance and dramatically increases productivity for radar, sonar, and imaging applications. CodeSourcery's products and services deliver on the promise of open-source software and open standards. Founded in 1997, CodeSourcery is a privately-held company headquartered in Granite Bay, California.

Fully functional trial versions of Sourcery VSIPL++ are available at no charge. To learn more about CodeSourcery and about Sourcery VSIPL++, visit <http://www.codesourcery.com> or call (888) 776-0262.

<http://www.codesourcery.com>

(888) 776-0262

Sourcery VSIPL++ is a trademark of CodeSourcery, Inc. Cell Broadband Engine (Cell/B.E.) is a trademark of Sony Computer Entertainment, Inc. All other product and service names are the property of their respective owners.