Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance

Level: Intermediate

Daniel A. Brokenshire (brokensh@us.ibm.com), Senior Technical Staff Member, IBM STI Design Center

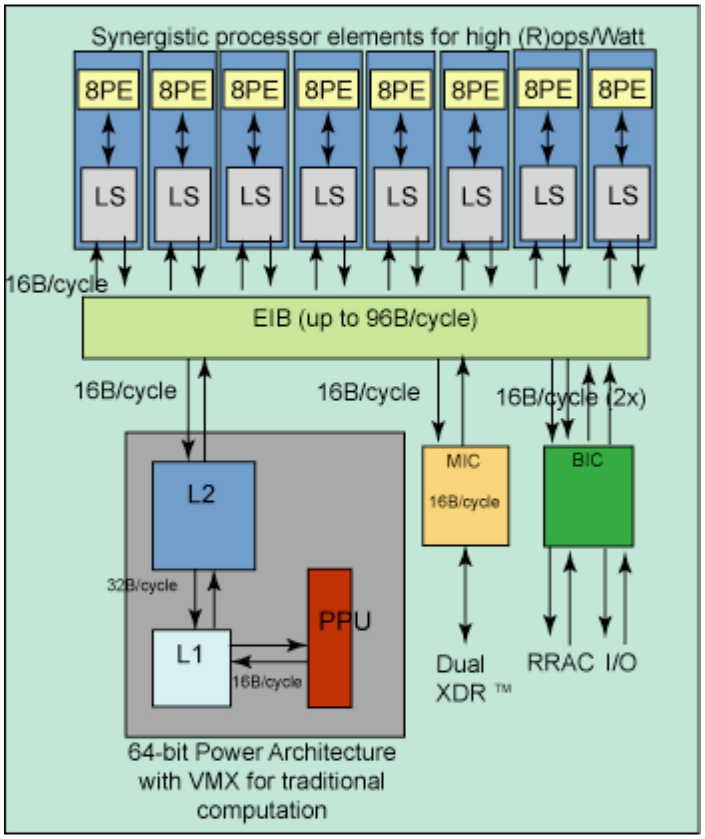
27 Jun 2006

Unlike on conventional processors, you can achieve near theoretical-maximum performance for real applications on the Cell Broadband Engine™ (Cell BE) processor. For this, you must be aware of the Cell BE processor's architectural characteristics: get to know them better with these 25 tips to optimal application performance.

The Cell BE architecture consists of a 64-bit Power-compliant Processing Element (PPE) and eight Synergistic Processing Elements (SPEs), loosely coupled through a coherent memory sub-system. Through their programmability, the SPEs provide computational density advantage over conventional processors while still providing greater flexibility than traditional fixed function ASICs. The SPEs' instruction set and supported data types accommodate efficient computation for a wide variety of applications including network processing, cryptography, high performance computing (HPC), graphics geometry processing, transcoding, and others.

Unlike conventional processors, near theoretical-maximum performance can be achieved for real applications on the Cell Broadband Engine processor. However, the programmer, code generation tools, or both must be aware of the architectural characteristics of the processor to achieve optimal performance. These characteristics include multiple heterogeneous execution units, Single Instruction Multiple Data (SIMD) processing engines, limited local store, software managed cache, memory access latencies, dual instruction issue rules, both large and wide register files, quad-word memory accesses, branch prediction, and synchronization facilities.

Figure 1. The architecture of the Cell BE processor



This article offers software strategies and techniques to exploit the architectural features of the Cell BE processor to achieve super-linear performance improvement over today's conventional general-purpose processors. Sample workload performance results demonstrate the effectiveness of the technologies presented.

The article presents 25 programming tips to achieve optimal application performance. Tips 1-4 cover overall Cell BE programming practices; tips 4-6 are specific to the PPE; tips 7-14 cover effective memory subsystem practices; and tips 15-25 provide SPE-specific programming strategies.

Cell BE system practices

Tip 1: Offload as much work onto the SPEs as possible

The Cell BE processor has eight times as many SPEs as PPEs. Even computations that are not well suited to the SPEs (for example, branchy scalar code) might be best offloaded. Use the PPE as the control processor -- orchestrating SPE execution and assisting the SPEs with exceptional events such as virtual memory management (VMM) misses. Use the SPEs as data plane processors performing all the heavy computational lifting.

Tip 2: Choose a partitioning and work allocation strategy that minimizes atomic operations and synchronization events

Autonomous, asynchronous computing is critical in maximizing computational throughput. One possible strategy is to let the SPEs algorithmically partition the work. For example, consider an image-processing application in which the scan lines are processed by *n* SPEs. Each SPE can algorithmically compute its work partition by solving the following simple equation:

```
height = CEIL(image_height / n);
first_line = height * spe_num
```

```
last_line = MIN(image_height, first_line+height)-1
```

Alternatively, the control processor can place work requests into a work queue or a set of work queues. If the work requests are computationally variable and non-predictable, the SPEs can self-arbitrate for work from a single queue by using atomic operations. When tasks are predictable in duration, the control processor can distribute the work amongst separate queues for each SPE.

Consider all domains in which to partition the problem. For example, many applications can be partitioned in both space and time.

Tip 3: Accommodate potential data type differences

The PPE and SPEs may not use equivalently sized data types. The PPE can be either ILP32 (integers, longs, and pointers are 32 bits), or LP64 (integers are 32 bits, longs and pointers are 64 bits). The SPE is always ILP32. Therefore, 64-bit system applications should be careful when creating shared (PPE and SPE accessed) data structures so that compatible types and their associated alignment are preserved.

The PPE is optimized for both single and double-precision arithmetic. However, the first Cell BE implementation is significantly more efficient at computing single-precision floating results. Every cycle, a four-way SIMD, single-precision, floating-point multiply-accumulate can be issued with a latency (results available) of six cycles. In contrast, a two-way SIMD, double-precision, floating-point multiply-accumulation operation can be issued every seven cycles with a total latency of 13 cycles. If double-precision arithmetic is not required, use single precision.

PPE programming practices

Tip 4: Exploit multithreading

The PPE is two-way multithreaded in that two threads of architectural state are maintained. However, the threads are not completely independent because many execution resources are shared by the threads to reduce the hardware implementation cost. As such, multithreading is strongly encouraged when:

1. The threads experience significant L1 and L2 cache misses. This commonly occurs in applications that perform lots of pointer chasing or scattered array/vector accesses.
2. The threads contain poorly pipelined fixed-point or floating-point operations. An execution stall occurs when there are dependencies between operations (for instance, the inputs of an operation depend upon a previous operation).

Tip 5: Self-manage cache

Applications that have predictable data access patterns can exploit this knowledge by pre-loading the caches so that data is available when it is needed. The PPE supports two forms of *dcbt* (data cache block touch) instructions: classic (th=0) and enhanced (th=8). The classic form pre-fetches a single cache line from memory into both the L1 and L2. The enhanced form pre-fetches up to a page of memory into the L2.

Note that the VMX data stream instructions (dst, dstst, and dststt) have no effect on the Cell BE processor and should not be used for pre-loading the cache.

Tip 6: Avoid microcoded opcodes

Programmers should avoid microcoded instruction opcodes by using a compiler that supports compilation for the PPE machine type. When a microcoded op is encountered by either HW thread, both threads are stalled.

PPE microcoded ops include:

- CR recording instructions (Rc=1)
- Shift and rotate instructions where the shift amount is source from a register (as opposed to an immediate value encoded in the instruction)
- Load and store algebraic, multiple, and string instructions
- Misaligned accesses including Dcache accesses that cross 32-byte boundaries and double floating-point loads and store to an odd word boundary

Memory subsystem programming practices

Tip 7: Make efficient use of programmer-managed data transfers

The Cell BE architecture encourages SPE programmers to manually initiate all transfers in and out of the SPE's local store (LS). This forces the programmer to be aware of all data accesses and encourages thought regarding application data access patterns.

Consider, for instance, the large FFT example presented by A. Chow at GSPx 2005 (see Resources). A straightforward FFT implementation would require 24 ($\log_2(16M)$) passes through the data in order to perform the FFT. Such an implementation would be memory bound. To reduce the memory accesses, Chow utilized a variation of the stride-by-1 algorithm proposed by D. H. Bailey based upon Stockham's self-sorting FFT in which eight butterfly stages are processed at once. Understanding and compensating for the memory access patterns reduced the number of passes through the data to three.

Tip 8: Design data structures for efficient access

To achieve efficient SPE data accesses, programmers should consider data alignment, access patterns, and location.

The SPE's memory flow controller (MFC) supports transfers of 1, 2, 4, 8, and n*16 (up to 16k) bytes; transfers less than 16 bytes must be naturally aligned and have the same quad-word offset for the source and destination addresses.

The Element Interconnect Bus (EIB) overhead is minimized if transfers are at least 128 bytes, and transfers greater than or equal to 128 bytes should be cache-line aligned (aligned to 128 bytes).

Avoid PPE pre-accesses to large data sets, so that most SPE-initiated data transfers come from system memory, instead of the PPE's L2 cache. MFC transfers from system memory have high bandwidth and moderate latency, whereas transfers from the L2 have moderate bandwidth and low latency.

Tip 9: Initiate DMAs from the SPE

Instead of the PPE *pushing* data to the SPE using the SPE's proxy command queue, let the SPE *pull* the data using SPE to initiate the DMAs. This is done for these reasons:

1. There are eight times more SPEs than PPEs.
2. The SPE command queue is twice as deep as the proxy command queue.
3. Consumer-managed transfers are easier to synchronize.
4. The number of cycles to initiate a transfer from the SPE is smaller than the number of cycles to initiate the same transfer from the PPE.

Tip 10: Lock to avoid thrashing

Consider locking an address range into the L2 cache using the Cell BE processor's Replacement

Management Table (RMT) feature. This allows frequently accessed data buffers from getting cast out, and it keeps streamed data from casting out other cached data.

Tip 11: Allocate large data sets from large pages to reduce page table and TLB thrashing

The Cell BE processor supports three concurrent page sizes -- 4KB and any two of 64KB, 1MB, or 16MB. Allocating large data sets from large pages reduces frequent TLB reloading penalties. TLB trashing can be significant. For example, the 16M-point FFT sample provided in the Cell BE Software Development Kit (SDK; see Resources), runs up to three times faster when the data buffers are allocated from 16MB pages instead of 4KB pages.

Tip 12: Maintain synchronization variables in their own reservation block

Atomic operations operate on reservation blocks corresponding to 128-byte cache lines. As a result, synchronization variables should be placed in their own cache line so that other non-atomic loads and stores do not cause inadvertent lost reservations.

Tip 13: Uniformly distribute memory bank accesses

The Cell BE memory subsystem has 16 banks, interleaved on cache line boundaries. Addresses 2KB apart access the same bank. System memory throughput is maximized if all memory banks are uniformly accessed.

The initial implementation of the 16M-point FFT (see Resources) allocated the two (real and imaginary) 16M-element floating point arrays on page (16MB) boundaries. With eight SPEs accessing memory in a regular power of two manner, only half (eight) of the 16 memory banks were being accessed at a time. Since the implementation was memory bound, improving the distribution of memory banks access was required to further improve performance. The solution was to offset the imaginary array by 1K (eight banks) so that all 16 memory banks are simultaneously accessed. This simple change resulted in a 25% improvement in performance.

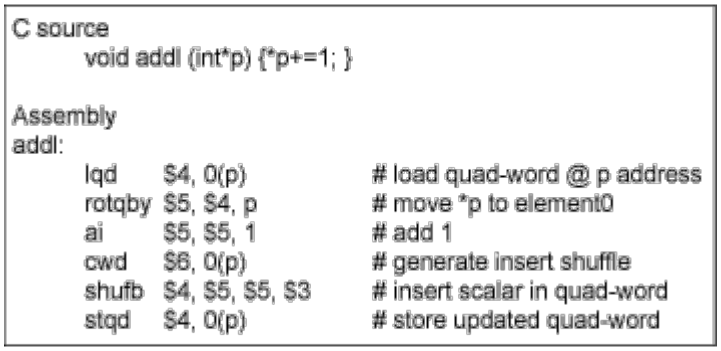
Tip 14: Stay on-chip

The EIB provides significantly more bandwidth than system memory. Many applications might be memory bound and therefore can benefit from keeping data transfers, communications, and synchronization on chip and not consume memory bandwidth. This includes (1) exploiting LS to LS DMA transfers when sharing data between SPEs and (2) utilizing mailboxes, signal notification registers for small data communications and synchronization.

SPE programming practices

Tip 15: Avoid external scalars

The SPE only accesses local store a quad-word at a time. As such, scalar (subquad-word) loads and stores require several dependent instructions. This is due to the fact that scalar loads are rotated into the preferred vector element, and stores require a read, scalar insert, write operation. The following code sample demonstrates this overhead:



```
C source
        void addl (int*p) {*p+=1; }

Assembly
addl:
        lqd     $4, 0(p)            # load quad-word @ p address
        rotqby  $5, $4, p           # move *p to element0
        ai      $5, $5, 1           # add 1
        cwd     $6, 0(p)            # generate insert shuffle
        shufb   $4, $5, $5, $3      # insert scalar in quad-word
        stqd    $4, 0(p)            # store updated quad-word
```

Several strategies make scalar code (code that is not appropriate for vectorization) more efficient. These include:

- Change the scalars to quad-word vectors. This might seem wasteful; however, if you consider the three extra instructions associated with loading and storing scalars, this trade-off can actually have a positive impact on code size.
- Cluster scalars into groups and load multiple scalars at a time using a quad-word memory access. Manually extract or insert the scalars on an as-needed basis. This will eliminate redundant loads and stores.

Consider an implementation of RC4 -- a character-based encryption algorithm that utilizes a 256-byte dynamic state table. Because each iteration of the algorithm is dependent upon the previous iteration, it can not be parallelized. However, by exploiting the strategies outlined above, you can still achieve significant performance enhancements. The 256-byte state table is expanded into a 256 quad-word state table in which all 16 elements of the unsigned character vector contain the same byte value. The input and output messages are fetched and stored a quad-word at a time to eliminate the extraneous loads and stores and their respective latencies. These changes have resulted in an 86% improvement in performance.

Tip 16: Exploit SIMD

When programmers write code in a high-level language (for example, C or C++), they must rely on compiler technology to auto-vectorize the code to exploit SIMD capability of the SPE. However, the flexibility of these languages makes it extremely difficult to achieve optimal results for many applications. This has resulted in programmers having to resort to using assembly for performance-critical sections to achieve the results they desire. Writing assembly on the SPE can be a daunting task for large, complicated code because of the large register file and specific issue rules.

A compromise solution is *intrinsics*. Intrinsics are essentially inline assembly with C function call syntax. They provide the programmer explicit control of the instructions used, but (unlike assembly) eliminate many of the optimization tasks that compilers are good at. These include register coloring, instruction scheduling, data loads and stores, looping and branching, and literal vector construction.

Tip 17: Understand the instruction set and issue rules

When coding using intrinsics, you should understand the instruction set. This includes understanding:

1. How the intrinsics map to instructions
2. The dynamic range and sign of immediate values to avoid extraneous literal construction
3. Branch instructions to formulate efficient conditionals
4. How vector literals are constructed

Just as important, programmers should understand instruction issue rules and latencies. The SPE contains two instruction pipelines with instructions pre-assigned to execute on only one of the pipelines. Two instructions are issued every clock assuming:

- there are no dependencies
- operands are available
- the even-addressed instruction (the least significant 3 address bits are 000) is a pipeline 0 instruction
- the odd-addressed instruction (the least significant 3 address bits are 100) is a pipeline 1 instruction, and
- the instructions are ordered pipeline 0 followed by pipeline 1.

Therefore, choosing instructions wisely can improve dual-issue rates.

Table 1. Instructions for the even pipeline

Pipeline 0 (even) Instructions	Latency (clocks)
Single precision floating-point ops	6
Double precision floating-point ops	6+7
Integer multiplies Integer/float convert Interpolate estimate	7
Immediate loads Logical ops Integer add/subtract Sign extend Count leading zero Select bits Carry/borrow generate	2
Element rotates and shifts Special bytes operations	4

Table 2. Instructions for the odd pipeline

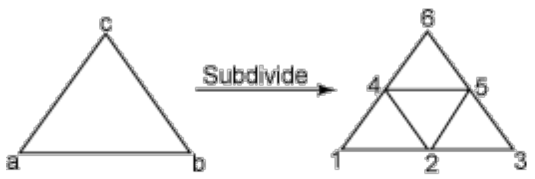
Pipeline 1 (odd) Instructions	Latency (clocks)
Loads and stores Branch hints Channel operations Moves to/from SPRs	6
Shuffle bytes Quad-word rotates and shifts Estimate (reciprocal, reciprocal sqrt) Gather bits Form select mask Generate insertion control Branches	4

Tip 18: Choose optimal SIMD strategy

There are multiple ways to SIMD an algorithm. The chosen method should be appropriate for the algorithm, data organization, and local store budget.

Consider, for example, subdivision surfaces in which the single triangle defined by floating-point vertices a, b, and c in Figure 2 below is subdivided into multiple triangles.

Figure 2. Point subdivision illustrated



There are several methods of vectoring this algorithm. They include:

1. *Evaluate subdivision vertices one at a time.*
   This is a traditional scalar method of performing subdivision. Vertices are typically represented in a "vec-across" format (that is, each three or four component vertex is maintained in a single SIMD vector). This method of representing a 3-D vertex is very natural and often produces small code. However, when applied, it typically produces less efficient code and generally requires significant loop unrolling to improve its efficiency. If the vertices contain less components than the vector can hold (for example, three component vertices), then further efficiencies are compromised.
2. *Evaluate subdivision vertices of four independent triangles one at a time.*
   An easier method of SIMDizing code is to program as if it were scalar and populate the vectors with independent data. Vertex data is represented in "parallel-array" format, in which each vertex component is maintained in a separate array. If the input data is in the vec-across format, the spu_shuffle intrinsic can be used to put the data into a parallel-array format.
3. *Evaluate four subdivision vertices for a single triangle at a time.*
   In this case the vertex control data is replicated across the vectors and unique weighting factors are maintained in each element of the vector so that four subdivided vertices can be computed in parallel. Inefficiency can result when the number of subdivision vertices is not a multiple of four.
   All three of these SIMD strategies have been applied to Curved Point-Normal triangle subdivision (see Resources). The following metrics were extracted to compare and contrast the performance, code side, and efficiency of each of these techniques. The data shows that the optimum solution is strategy 2.

Table 3. Performance of various SIMD-conversion strategies

Metric	**Strategy**				
	1			**2**	**3**
Unroll factor	none	2	4	none	none
Normalized perf	1.0	1.26	1.54	1.70	1.34
CPI	1.10	1.04	0.90	0.99	0.96
Dual issue %	9.1	12.6	18.5	13.6	11.9
Dependency stall %	14.1	13.6	5.8	4.7	2.6
Registers used	72	112	127	113	107
Text size (bytes)	1472	2496	4480	512	1856

Tip 19: Unroll and pipeline loops

Loops are the foundation in nearly all programs (especially streaming applications). If the number of loop iterations is constant, then consider removing the loop altogether. If the number of loop iterations is variable, consider unrolling the loop as long as the loop is relatively independent (that is, an iteration of the loop is not dependent upon the previous iteration). The SPE has a large register file, and significant loop unrolling can be accomplished before register spilling occurs. That is when the instantaneous number of active variables exceeds the size of the register file. Unrolled loops provide additional instructions for the optimizer to efficiently schedule.

When unrolling loops, additional local variables might be required to eliminate "false dependencies" amongst the loop variables. Failure to eliminate these false dependencies can cause unrolled loops to not be interleaved by the compiler.

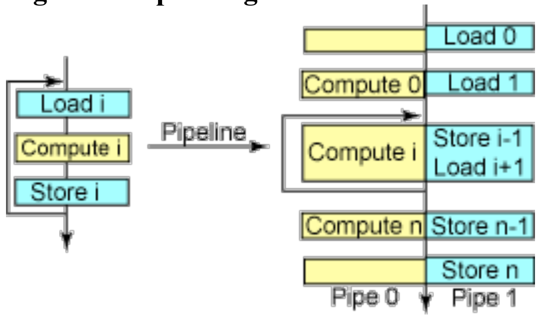
The following data shows the effectiveness of loop unrolling a sample workload that performed OpenGL-like coordinate transformation and lighting.

Table 4. Performance metrics for unrolling loops

Metric	**Unroll Factor**			
	1	**2**	**4**	**8**
Normalized perf	1.0	1.52	1.73	1.66
CPI	1.35	0.91	0.76	0.67
Dual issue %	3.3	19.8	34.3	35.8
Dependency stall %	27.2	5.9	0.9	1.5
Registers used	78	103	128	128
Text size (bytes)	768	1344	3076	5252

Most loops generally have the same basic structure. Per iteration, they load input data, perform computation, and finally store the results. Since loads, stores, quad-word rotates, and shuffles execute on pipeline 1, and most computation instructions execute on pipeline 0, loops can be software pipelined to improve dual-issue rates by computing at the same time as loading and storing data.

Figure 3. Pipelining and dual-issue



Applying pipelining to the previously referenced vertex transformation and lighting workload resulted in the following improved metrics:

Table 5. Performance results for pipelining

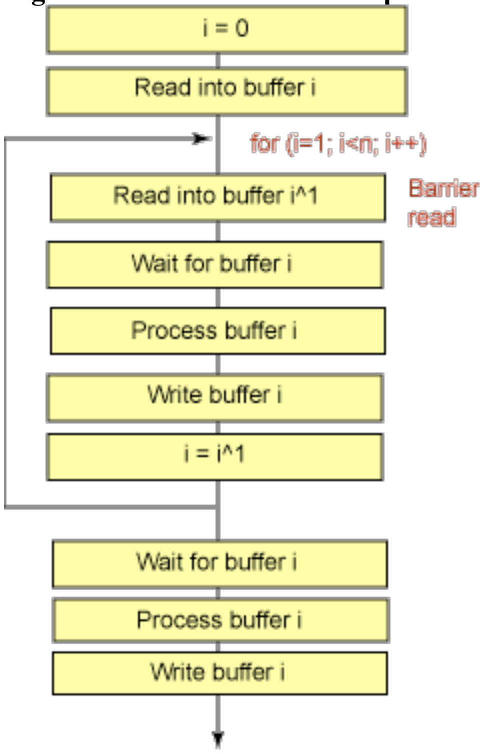
Metric	**Unroll Factor**	
	2	**4**
Normalized performance	1.82	1.83
CPI	0.75	0.69
Dual issue %	36.7	47.8
Dependency stall %	1.3	0.5
Registers used	114	128

Text size (bytes)	2436	3468
Speedup vs. non-pipelined	1.16	1.08

Tip 20: Overlap data movement with computation

To hide data-access latencies, utilize double- (or multi) buffering techniques. You can can either double buffer the data (this is more typical), or double buffer the code, depending upon the code and data sizes and data access patterns.

Figure 4. A double-buffer loop



When double buffering, apply these general rules:

- Use unique DMA tag IDs, one for each multibuffer.
- Use *fenced* commands to order the DMAs within a tag group.
- Use *barrier* commands to order DMAs within the queue.

Tip 21: Eliminate or reduce branches

Branches are relatively expensive (up to 18-19 cycles when mis-predicted). Not only are they expensive from their issuance and stalls due to mis-predicts, they create a boundary for scheduling optimization. Therefore, you should avoid their use, if at all possible.

The secret to eliminating branches is exploiting the *select bits* instruction. For example, an if-then-else statement can be made branchless by computing the results of both the *then* and *else* clauses and using *select bits* to choose the result as a function of the conditional. For example, the conditional:

```
if (a > b) c += 1;
else d = a+b;
```

can be made branchless as follows:

```
select = spu_cmpgt(a, b);

c_plus_1 = spu_add(c, 1);

a_plus_b = spu_add(a, b);

c = spu_sel(c, c_plus_1, select);

d = spu_sel(a_plus_b, d, select);
```

When you can not eliminate branches, then attempt to reduce the number of branch mis-predicts. You can accomplish this by either utilizing feedback-directed optimization technologies or programmer-directed branch prediction. The programmer can explicitly direct branch prediction using the `__builtin_expect` language extension. Considering the previous example, you can direct the compiler that "*a*" is more likely not larger than "*b*," thereby preferencing the else condition by adding a *__builtin_expect* as follows:

```
 if (__builtin_expect((a > b), 0))    c += 1;

else         d = a+b;
```

Tip 22: Avoid integer multiplies

The SPE contains only a 16x16 bit multiplier. Therefore, performing a 32-bit integer multiply takes five instructions -- three 16-bit multiplies and two adds -- to accumulate the partial products.

To avoid extraneous multiply cycles, follow these rules:

- If the operands are less than 16 bits in size, cast them to unsigned shorts prior to multiplication to take advantage of the native multiplier.
- Make sure to also cast constants since they have an implicit type of int.
- Keep array elements a power-of-two size to avoid multiplication when indexing.
- Consider using a macro to explicitly perform 32-bit integer multiplies to avoid inadvertent introduction of signed extends and masks due to casting.

Tip 23: Use offset pointers

The PowerPC® processor supports *load/store with update* instructions. These instructions allow you to sequentially index through an array without the need of additional instructions to increment the array pointer.

The SPE does not support this instruction form. Instead, you should exploit the d-form by specifying small literal array offsets from the base array pointer.

What is a d-form?
The SPU instruction set (as well as the PowerPC instruction set) supports multiple forms of load and store instructions. The types include a-form, d-form, and x-form. The differences between these forms are the method in which the load/store target addresses are constructed. For the a-form, they are constructed from an immediate address (in other words, the address is encoded in the instruction). For the x-form, they are constructed from the sum of two registers. For the d-form, they are constructed from the sum of a register and an immediate offset.

Tip 24: Consider computing versus using pre-computed results

For years, programmers have been improving application performance by developing tables of pre-computed values. For SPE programs, this is typically not ideal. Tables do not SIMDize well and consume valuable LS space. As such, you should consider exploiting the SPE's huge computational capacity by instead computing values on the fly.

Tip 25: Design for limited local store

The SPE local store is a limited resource. 256Kbytes is available for program, stack, local data structures, and DMA buffers. Many of the optimization techniques presented in this paper add additional pressure on this limited resource. As such, all optimizations might not be possible for a given application: programmers might be forced to utilize only a few optimizations due to the limited local store constraint.

Often you can reduce local store pressure by dynamically managing the program store using code overlays -- also known as plugins.

Conclusion

The Cell BE processor is a very powerful processing complex. Specialized programming techniques employed either directly by the programmer or indirectly through tools (like compilers) can pay great dividends toward application performance. Armed with the strategies and techniques outlined in this article, you can realize the full potential of the Cell Broadband Engine processor.

Special thanks go to the members of the software and performance STI Design Center teams for developing the workloads and performance results cited in this paper.

Resources

Learn

- Read Cell Broadband Architecture and its first implementation by T. Chen, R. Raghavan, J. Dale, and E. Iwata.

- Curved PN Triangles (in PDF format) by A. Vlachos et al., from the Proceedings of the 2001 Symposium on Interactive 3D Graphics, 2001, discusses substituting a three-sided cubic Bezier patch for regular old triangles.

- The IBM Semiconductor solutions technical library's Cell Broadband Engine documentation section lists specifications, user manuals, and more. Of particular interest for readers of this article are:
    - Cell Broadband Engine Architecture, Version 1.0
    - PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology, Version 2.06c
    - Synergistic Processor Unit Instruction Set Architecture, Version 1.0
    - SPU C/C++ Language Extensions, Version 2.1

  As well, the Cell Broadband Engine Programming Handbook is an indispensable resource.
- A Programming Example: Large FFT on the Cell Broadband Engine (in PDF format) was presented at GSPx 2005 by A. Chow et al. It, too, is available from the Semiconductor solutions technical library.

- Find all Cell BE-related articles, discussion forums, downloads, and more at the IBM developerWorks Cell Broadband Engine resource center: your definitive resource for all things Cell BE.

- Keep abreast of all the Cell -- and other Power Architecture-related news: subscribe to the Power

Architecture Community Newsletter.

Get products and technologies

- Get Cell BE: Contact IBM E&TS for custom Cell BE-based or custom processor-based solutions.
- Get the alphaWorks Cell Broadband Engine downloads -- including the IBM Full System Simulator and the IBM Cell Broadband Engine Software Sample and Library Source Code.
- See all Power-related downloads on one page.

Discuss

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

About the author

Daniel A. Brokenshire is a Senior Technical Staff Member in the IBM STI Design Center (Austin, Texas). His responsibilities include the development of programming standards, language extensions, and reusable software libraries for the Cell BE processor. He received a BS in computer science and BS and MS degrees in electrical engineering, all from Oregon State University. Prior to his work on the Cell BE processor, he was involved in the development of 3-D graphics products for both Tektronix, Inc. and IBM.