

The little broadband engine that could: More on rendering fractals on the SPE

Use the SPE's effectiveness in vector operations to optimize the example application even further

Skill Level: Intermediate

[Peter Seebach \(developerworks@seebs.net\)](mailto:developerworks@seebs.net)

Freelance writer

Wind River Systems

29 Jul 2008

In the previous article in the series, you learned about the challenges of rendering fractals on the SPE. That article focused on the SPEs copying their rendering results directly into the target data buffer. This article shows you how the fractal generator can be optimized further by taking advantage of the SPE's fondness for vector operations.

Introduction

The previous article introduced a very simple generator for points of the Mandelbrot set, which obtained a noticeable performance boost from using SPEs for calculation, despite using only scalar operations. In this article, you'll see how that generator could be optimized further by taking advantage of the SPE's proclivity for vector operations.

Reviewing what you've learned so far

If you haven't read the earlier article in this series "[Why is my scalar code so slow?](#)", this is a great time to skim through it for a brief introduction. This article assumes you are basically familiar with the vector-only model that the SPEs use.

To make a long story short, there are no scalar operations on SPEs. None. Scalar-like operations can be performed using the *preferred* slot in a vector register: the first slot, also called slot 0. However, data reads and writes are always aligned, so if the datum you want to operate on wasn't already aligned on a 16-byte boundary, the compiler must generate additional code to shift, rotate, mask, and otherwise mangle data to get the datum you asked for into the preferred slot for operations.

On a processor that offers vector operations as an option in addition to scalar operations, the best-case increase from vector operations is normally a speed increase lower than the vector size. For example, using the AltiVec/VMX extensions on the PPE core, you would expect performance on vectors of floats to be somewhat less than four times scalar floating point performance. (Note: Like nearly everything other than the processor specification, this is an oversimplification.)

Understanding vector sizes and data types

In most of the examples in this series, vectors have been four objects. In fact, while all vectors are the same size, they are not always four objects; rather they are always 128 bits. A vector of 32-bit floats holds four objects. A vector of 64-bit doubles holds only two.

For this application, double-precision math is very desirable. However, the output is an array of 32-bit integers, which are 4 to a vector. Therefore, in addition to the normal issues of vectorization, the `x_size` variable in the loop processing command-line arguments guarantees that every row will have a multiple of four items to process and eliminates a complicated special case from the code.

Listing 1. A simplifying assumption

```
case 'x':
    x_size = (int) v;
    /* to make life easier on the SPE */
    if (x_size % 4) {
        x_size &= ~3;
    }
    break;
```

(Don't worry: there are plenty of type hassles left to keep things interesting.)

Note that while many operations can be performed on every data type, there are a few exceptions. For example, the `spu_add` vector intrinsic works on float and double types and on 32-bit integer types, but not on 64-bit integer types. This comes back to haunt your (not so) simple vectorization.

Setting up and understanding relative constants

The `do_mand()` function now iterates through the list in batches of four. Each batch of four is calculated from two pairs of incoming values. Listing 2 shows some of the declarations from the top of the revised `do_mand()`:

Listing 2. Setting things up

```
int i, j;
vector double px0, px1, py;           /* starting x and y values */
    vector double x0, x1, y0, y1;     /* values being iterated over */
vector double nx0, nx1;               /* temporary values */
vector double xsq0, xsq1;             /* x0 and x1 squared */
vector double ysq0, ysq1;             /* y0 and y1 squared */
vector unsigned int done0, done1;     /* checking which */
vector unsigned int d0, d1;           /* values escaped */
int done;
vector double vscale, vmin_x;         /* scale, and starting x value */
vector signed int i0, i1;             /* iteration count */
vector double vcsquared;              /* a vector of the c^2 value */
vector unsigned char shuffler = (vector unsigned char) {
    0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B,
    0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1A, 0x1B
};
/* constants */
vector signed int zero = spu_splats(0), unity = spu_splats(1);
vector double two = spu_splats((double) 2.0);
```

This rather large block of declarations contains a few declarations you can use to hold intermediate values, such as `d0` and `d1`. Declaring pairs of variables rather than small arrays encourages the compiler to use registers, of which the SPE has a plentiful supply. The `vscale` parameter is of particular interest. The value of the `py` parameter is constant throughout the row, so you need to calculate it only once. The original calculation looks like this:

```
py = min_y + ((double) row / count_y) * size_y;
```

Calculating `py` with vector operations is somewhat difficult. There is no division operation available, and the reciprocal estimate instruction isn't accurate enough. But the compiler can do a single division tolerably well, so why not just do that and use the `spu_splats()` intrinsic to get a usable vector?

```
py = spu_splats(min_y + ((double) row / count_y) * size_y);
```

However, that won't solve the problem of needing to calculate the value of each `x` coordinate, which involves a similar calculation:

```
px = min_x + ((double) i / count_x) * size_x;
```

To calculate the starting x values, you need to perform this calculation, but this runs into the same division problem as before. Conveniently, it's possible to shortcut this. Both `count_x` and `size_x` are constants that do not vary within a row. This can be reformulated as $px = (i * scale) + min_x$, and this conveniently maps well onto the SPE's fused multiply-add:

```
px0 = (vector double) { i, i+1 };
px0 = spu_madd(px0, vscale, vmin_x);
```

With this figured out, the rest of the core algorithm is simple to translate with two significant exceptions:

- Determining when no more iterations are needed
- Determining how many iterations each value needed

Both of these require a bit of clever work with the SPE's comparison and selection operators.

Venturing on the road less travelled

In a scalar operation, it is easy to understand branching logic. In a vector operation, though, it is possible that a conditional test will be true of some members of a vector but not of others. The SPE, much like AltiVec/VMX, handles this by using conditional assignments. The `spu_sel` intrinsic selects values from two vectors based on the bits set in a pattern vector. In turn, these are set based on comparison operators, which work on a whole vector.

The SPE's inner loop continues until all four of the values it is working on have escaped or until it has run the maximum number of iterations. Determining whether the loop is done involves calculating the squares of the current x and y values and comparing them to the `vcsquared` vector.

Listing 3. Are you there yet?

```
done0 = (vector unsigned int) spu_cmpgt(vcsquared, spu_add(xsq0, ysq0));
done1 = (vector unsigned int) spu_cmpgt(vcsquared, spu_add(xsq1, ysq1));
d0 = spu_add(done0, spu_rlqwbyte(done0, 8));
d1 = spu_add(done1, spu_rlqwbyte(done1, 8));
done = spu_extract(spu_add(d0, d1), 0) == 0;
```

The `done0` and `done1` vectors start out with 1s in the bits corresponding to slots where c^2 is greater than $x^2 + y^2$. They are then converted into vectors of unsigned int, meaning that each of the 64-bit values is treated as two 32-bit values. These are then added to themselves rotated left by 8 bytes. This means that in `d0`, the first 4 bytes will have the value of 0 if both vector slots were 0, -1 if one vector

slot was not 0, and -2 if both vector slots were not 0. The same goes for `done1` and `d1`. Thus, if adding these two vectors together yields a sum of 0, all 4 comparisons yielded 0. That means that in all 4 cases, x^2+y^2 was greater than c^2 , which means that all 4 values have escaped, and there is no need for more iterations.

However, the `done0` and `done1` vectors have another role to play. Normally the iteration count is incremented each time through the loop. With a vector of values, if the iteration count for every value is incremented until all values are done, the results will be fairly thoroughly wrong. This is where the `spu_sel()` intrinsic comes in.

Listing 4. What's ++ got that you haven't got?

```
i0 = spu_sel(i0, spu_add(i0, unity), done0);  
i1 = spu_sel(i1, spu_add(i1, unity), done1);
```

Where `done0` has 0 bits, the corresponding bits from the original, unmodified `i0` are selected. Where `done0` has 1 bits, the corresponding bits from the modified `i0` are selected. Because `done0` has 1 bits wherever the operation is not yet done (slightly confusing, I admit), this means that the incremented value is selected in the not-done components but that the previous value is selected in the done components. The `unity` vector exists to remove repeated calls to `spu_splats()`.

With this complete, you can now easily write a loop that runs for, at most, the specified number of iterations, stopping earlier when it can, and that records the number of iterations required for each point to escape. These values are stored across two vectors.

This might be a good time to ask: Why are they vectors of 32-bit integers rather than vectors of 64-bit integers? The answer is simple: The `spu_add` intrinsic is not defined for 64-bit integers. Because there's no additional cost to performing operations on four integers instead of two and because the additional range is unneeded, it is easier to simply use 32-bit values. (A disclaimer: This program cannot calculate the Mandelbrot set to a depth of 4.3 billion iterations. So sorry.)

The remaining question is how to get these iteration counts, stored in two vectors, combined into a single vector of four integers. It would be quite possible to do this with a series of `spu_extract` operations and direct assignments into the array of integers. But there is a better way.

Remember the previously unexplained variable named `shuffler`. This is used with the `spu_shuffle` intrinsic, which can populate a vector with arbitrary bytes from two other vectors, as well as constant 0x00, 0xFF, or 0x80 bytes.

Listing 5. The big shuffle

```
vector unsigned char shuffler = (vector unsigned char) {
    0x00, 0x01, 0x02, 0x03,
    0x08, 0x09, 0x0A, 0x0B,
    0x10, 0x11, 0x12, 0x13,
    0x18, 0x19, 0x1A, 0x1B,
};
    /* used to accumulate results */
vector signed int *vb = (vector signed int *) buffer;
[...]
```

```
*vb++ = spu_shuffle(i0, i1, shuffler);
```

This operation fills the first four bytes of the resulting vector (the first integer) with bytes zero through three of the first vector argument (`i0`). Those hold the number of times the first double vector was incremented. The next element (the second integer in the vector) is initialized with bytes eight through eleven of `i0`. These hold the number of times the second point in the `x0/y0` pair was incremented. The next two vectors are assigned from the same byte positions within vector `i1`. The fifth bit in the values in the shuffler array selects the second vector argument as the source.

Evaluating the conversion

And with that, the conversion is complete. The essential pattern of operations is the same. There are a few more explicit instructions involved per pass, but each pass is now handling four items rather than one. The code is substantially longer: the original implementation was about 20 lines of actual code and declarations, while the new version is 64 lines. But does it pay off?

Perhaps surprisingly, the answer is yes. The original naive implementation took about 8 seconds of total SPE time to render a 1600x1200 Mandelbrot set to a depth of 512 iterations. The new version takes a bit less than three seconds to achieve the same results.

The Mandelbrot set was chosen as a nice example of a workload excellently suited to simple SPE optimizations, and it delivered. Still, there are a number of optimizations that would be rendered difficult or impossible by this particular approach. For example, heuristic-based methods that look at small areas (rather than individual scan lines) do not mesh well with this kind of optimization.

Conclusion

The quest for a perfect demonstration of SPE programming techniques continues. The Mandelbrot set renderer is well-suited to simple SIMD optimizations. The relatively simple control loop doesn't seem to need much improvement. It's hard to imagine improving dramatically on something that scales linearly with the number of SPEs in use. It might go a bit faster with double-buffering, but it's pretty fast already. Given that rendering is now taking under one-fourteenth as long as displaying the output, it seems not to be worth a lot of extra work for now.

Resources

Learn

- Use an [RSS feed](#) to request notification for the upcoming articles in this series. (Find out more about [RSS feeds of developerWorks content](#).)
- Check out the other articles in the "[Little broadband engine that could](#)" series.
- Experiment with [Fractint](#), one of the most famous fractal generators.
- Want to plot pixels, but don't feel like writing X code? [GNU plotutils](#) can help.
- Start here: Wikipedia's introduction to the [Mandelbrot set](#) is a good starting point.
- Check out two flavors of documentation on how great DaCS is: "[DaCS for Cell/B.E. Programmer's Guide and API Reference](#)" and "[DaCS for Hybrid-x86 Programmer's Guide and API Reference](#)" (IBM Semiconductor solutions library, October 2007).
- Read "[Maximizing the power of the Cell Broadband Engine® processor: 25 tips to optimal application performance](#)" (developerWorks, June 2006) to learn some [SPE programming tips](#).
- Get information from Jonathan Bartlett's series on "Programming high performance applications on the Cell/B.E. processor" (developerWorks, January 2007 to present) provides an [intro to Linux® on the PS3](#), [programming the PS3's SPE](#), an [intro to the SPU](#), [SPU performance programming](#), [C/C++ SPU programming](#), and [managing smart buffer DMA transfers](#).
- To learn more on Cell/B.E. programming, try the developerWorks series:
 - "[Programming high-performance applications on the Cell/B.E. processor](#)"
 - "[PS3 fab-to-lab](#)"
 - "[The little broadband engine that could](#)"
- Speaking of Cell/B.E. SDK documentation, there's a new blog series that abstracts important topic sections of some of the major SDK documentation to give you a quick-read on the topic (in case you don't need a fuller explanation) -- they're called [Infobombs](#), and some topics already covered include
 - [Getting a successful FC7 install on a PS3](#).
 - [The basic structure of an ALF application](#).
 - [Double buffering on ALF as an optimization](#).
 - [ALF and DaCS for x86 hybrids](#).

- [Configuring and using the Basic Linear Algebra Subprograms.](#)
- [Glossaries on ALF and DaCS error codes, trace events, and ALF attributes.](#)
- Refer to the [Cell Broadband Engine documentation](#) section of the IBM Semiconductor Solutions Technical Library for a wealth of downloadable manuals, specifications, and more.
- Sign up for the [developerWorks newsletter](#) and get the latest developer news and Cell/B.E. happenings delivered to your inbox each week. Check *Power Architecture*® when you sign up to receive Cell/B.E. news in your newsletter.
- The [Cell Broadband Engine/Power Architecture notebook](#) is a blog-based resource that hosts [news](#), as well as two instructional features -- the "[Forum watch](#)" of interesting questions and hot topics from the forum and the "[Infobomb](#)" series (short, precise, task-specific, quick-read knowledge *bombs* gleaned from Cell/B.E. documentation).

Get products and technologies

- Get Cell: [Contact IBM about custom Cell-based or custom-processor based solutions.](#)
- Get your copy of the [IBM SDK for Multicore Acceleration 3.0](#) or browse through the [library of Cell/B.E. documentation.](#)
- Find all Cell/B.E.-related articles, discussion forums, downloads, and more at the IBM developerWorks [Cell Broadband Engine resource center](#): your definitive resource for all things Cell/B.E.
- Contact IBM about [custom Cell/B.E.-based or custom-processor based solutions.](#)

Discuss

- Get fast answers from IBM experts and real-world practitioners in developerWorks [Cell Broadband Engine discussion forum](#)
- Check out the [Cell Broadband Engine Architecture forum](#) to get your technical questions about the processor answered.
- Check out the [Cell Broadband Engine Architecture forum](#) to get your technical questions about the processor answered. Juicy problems and answers from the forums are rounded up periodically and highlighted in the "[Forum watch](#)" [blog series](#).
- Go to the [Power Architecture blog](#) for news, downloads, instructional resources, and event notifications for Cell/B.E. and other Power Architecture-related technologies. You can find the popular "Forum watch" blog series (Q&A

roundup), the "FixIt" technology updates, and the Infobomb quick-read technology introductions.

About the author

Peter Seebach

Peter Seebach has been writing about Cell Broadband Engine development for a few years, and programming in general for many more. He sort of wishes modern vector processors would come with a built-in couch.

Trademarks

IBM and developerWorks are trademarks of IBM Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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

Other company, product, or service names may be trademarks or service marks of others.