

An introduction to compiling for the Cell Broadband Engine architecture, Part 3: Make the most of SIMD

When all you have is a vector unit, everything looks like a parallel operation

Skill Level: Intermediate

[Power Architecture editors \(dwpower@us.ibm.com\)](mailto:dwpower@us.ibm.com)
developerWorks
IBM

07 Feb 2006

Third in the "[An introduction to compiling for the Cell Broadband Engine™ architecture](#)" series, this tutorial discusses the compiler issues in optimizing code to run efficiently on SIMD-capable processors. In particular, it shows how to optimize code that must run both on the VMX SIMD engine of the PowerPC® core of the Cell Broadband Engine (Cell BE) processor, and also on the SIMD-only Synergistic Processor Elements (SPEs).

Section 1. Before you start

Prerequisites

This tutorial presupposes familiarity with the material in the [first tutorial in the series](#), specifically computer architecture in general and the Cell BE architecture in particular, and basic familiarity with what compilers do. Familiarity with the second tutorial, "[Optimizing for the SPE](#)," is also helpful.

Section 2. Automatic simdization

Overview

This tutorial, the third in [the series](#), covers automatic conversion of code to take advantage of the Single Instruction Multiple Data (SIMD) units. (This is by contrast with the optimizations to perform scalar code on SIMD units, covered in the [previous tutorial](#).)

SIMD code offers some challenges not found in traditional vector code generation. This tutorial describes these issues and discusses some ways to overcome them. The term "simdization" is a special case of the more general term "vectorization." This tutorial covers the hardware constraints of the SIMD hardware in the Cell BE processor, highlights the simdization techniques used, and discusses some of the ways in which complexity is managed.

SIMD computation

SIMD instructions process multiple data per operation; in the trivial case, for instance, one instruction would load four items from an array into a single vector register, or add all of the components of two vector registers together. Two sorts of SIMD processing are available on the Cell Broadband Engine processor. The Power Processor Element (PPE) has a VMX unit (also known as AltiVec; see [Resources](#) for more information about VMX). The Synergistic Processor Elements (SPEs) run a special SIMD-only instruction set designed for computational density. In both cases, vectors are a fixed 128-bit length; this is most often four 32-bit objects, but they can also be treated as eight 16-bit objects or sixteen 8-bit objects.

Successful simdization

A simdizer must do some things to successfully convert loops to work effectively on an SIMD processor (or the SIMD unit of a general-purpose processor): Identify and extract parallelism in the code and satisfy constraints. For the Cell BE processor, the simdizer should support both the VMX instructions and the SPEs.

Extracting parallelism

You can identify three key kinds of parallelism in code: loop-level parallelism,

basic-block parallelism, and entire short loops. Short loops can be entirely eliminated, while larger loops must be unrolled partially, while keeping the loop structure intact. In some cases, an SIMD-friendly group of operations will occur in code without any kind of loop structure; these can also often be grouped together.

Satisfying constraints

SIMD processors tend to have a number of constraints that must be satisfied. For instance, both the VMX and SPE implementations support loads only on 16-byte boundaries. To obtain a range of elements not aligned on such a boundary, the compiler must load two ranges and combine them using a permutation operator. Data size conversion is another common problem; adding `short` and `long` values together requires that the `short` values be unpacked into multiple registers. Again, this is best handled by the compiler. Finally, the hardware only supports loading of contiguous regions; algorithms which require non stride-one access to memory (for instance, access to every other entry in an array) require the compiler to juggle bits around in the background.

Constraint: memory alignment

The SIMD hardware in the Cell BE processor loads and stores only at 16-byte boundaries. Misaligned access doesn't create traps or latency; it just never happens. Addresses are always truncated silently to 16-byte boundaries, effectively rounding down. Thus, if the arrays `b` and `c` are aligned at 16-byte boundaries, an attempt to load `b[1]` will actually load `b[0]` through `b[3]`. If you are trying to add elements that don't have the same relative alignment (say, `b[1]` to `c[2]`), simply performing the load operations and an add operation will produce garbage, as `b[1]` is added to `c[1]` instead of `c[2]`.

Constraint: stride-one memory access

The hardware allows only loading and storing of 16-byte contiguous chunks. If an algorithm requires access to, for instance, every other member of an array, then larger pieces of the array must be loaded and packed together. Storing is just as elaborate, requiring read/modify/write cycles; a chunk of memory is loaded, the objects to be stored are unpacked into the parts of it that need to be stored, and then the memory is written back out.

Constraint: data size in SIMD registers

A vector of `int` objects holds four objects; a vector of `short` objects holds eight. You cannot simply add the vectors together. Rather, you must either pack two `int`

vectors into a `short` vector, or unpack two `short` vectors into an `int` vector. For instance, the following code requires an unpacking operation:

```
int a[n];
short b[n];
for (i = 0; i < n; ++i)
    a[i] = a[i] + b[i];
```

This code would be implemented by loading eight members of `b` at a time, and loading two sets of members of `a`; one at `a[i]`, one at `a[i+4]`. The members of `b` are then unpacked into two vectors, which can be added to the two vectors of members of `a`.

Figure 1. One register of 16-bit values holds as many objects as two registers of 32-bit values

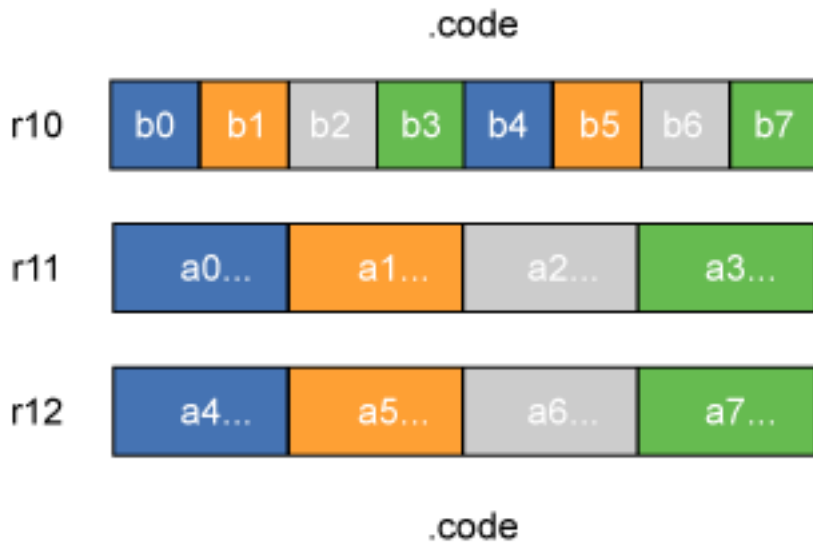
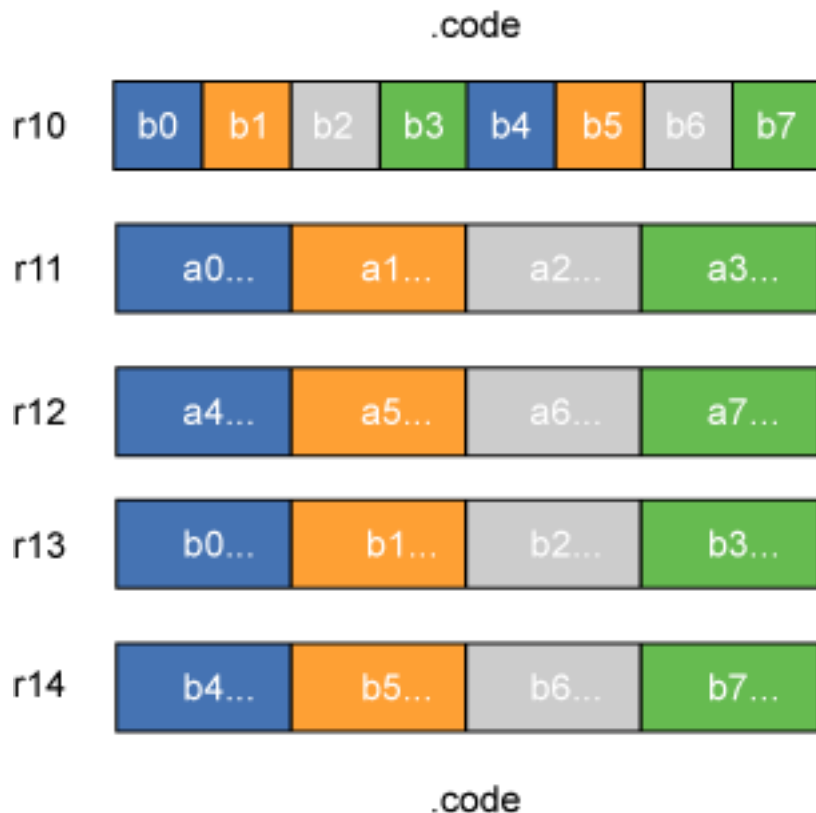


Figure 2. The 16-bit values expanded to match the 32-bit values



SIMD viewed as short vectors

Traditional vectorization looks for parallelism across loop iterations. A Cray-like vector unit might process up to 64 elements per vector. The much shorter vectors of an SIMD processor require narrower parallelism; typical SIMD units support two, four, eight, or 16-element vectors (usually 16 bytes). It is easier to find sufficient parallelism to make efficient use of this, and it may be found either in loops or in basic blocks.

Multimedia fits in short vectors

Short vectors are particularly well suited to typical multimedia data, such as colors (represented as R, G, and B fields, sometimes with an Alpha channel) or vertices (where the fields are X, Y, and Z). For instance, the following ray-tracing code (see [Resources](#) for more examples) is an excellent match for an SIMD processor:

```
for (i = 0; i < n; ++i) {
    q = quads[i];
    vertex[i].x = W0*q.p[0].x + W1*q.p[1].x + W2*q.p[2].x + W3*q.p[3].x;
    vertex[i].y = W0*q.p[0].y + W1*q.p[1].y + W2*q.p[2].y + W3*q.p[3].y;
    vertex[i].z = W0*q.p[0].z + W1*q.p[1].z + W2*q.p[2].z + W3*q.p[3].z;
}
```

```
} vertex[i].w = W0*q.p[0].w + W1*q.p[1].w + W2*q.p[2].w + W3*q.p[3].w;
```

Section 3. Simdization technique highlights

Alignment constraints overview

This section reviews ways in which the compiler can deal with alignment constraints in SIMD operations, starting from an example of unaligned code, and showing how the compiler can abstract and deal with the alignment properties of the data and the alignment requirements of the hardware.

Correct SIMD execution of misaligned computation

Our discussion of misaligned computation works from the following code fragment:

```
for (i = 0; i < 100; ++i)
    a[i+3] = b[i+1] + c[i+2];
```

For the purposes of this discussion, it is assumed that `a[0]`, `b[0]`, and `c[0]` are all aligned at 16-byte boundaries. Thus, it is assumed that the accesses to `a`, `b`, and `c` are each misaligned, and worse, misaligned not only with respect to memory, but with respect to each other.

Streamshifting data

At this point, we introduce the concept of streamshifting, which is to say, shifting all of the values in a stream, in a way analogous to shifts within a single register. Accessing all of the members of `b` sequentially using SIMD registers would bundle `b[0]` through `b[3]`, `b[4]` through `b[7]`, and so on; the goal is to bundle `b[1]` through `b[4]`, `b[4]` through `b[8]`, and so on.

In general, shifting a stream of values in this direction is called *left shifting*, and shifting them the other way (for instance, storing the four values in an SIMD register to `a[1]` through `a[4]`) is called *right shifting*.

Shifting to correct alignment.

For the example program, one solution would be to use a streamshift left of four bytes on `b`, and left of eight bytes on `c`, so that the first SIMD registers would contain `b[1]` through `b[4]` and `c[2]` through `c[5]`; adding these registers would produce the values for `a[3]` through `a[6]`, which could then be stored into the `a` array by streamshifting right by 12 bytes.

How could the compiler do it?

To use this design, the compiler must first build a data reorganization graph, indicating the shifts that are required. Then "shift streams" can be placed appropriately in the data flow using optimizing policies, and SIMD code can be used to perform the actual operations.

For instance, with the example program, one option might be to use a right shift of eight bytes on the `b` stream, and of four bytes on the `c` stream; this would leave both streams with an offset of twelve bytes, precisely correct for calculating values in `a`.

Abstracting alignment properties

The compiler concerns itself with two types of streams. The first is a stream of data in memory; the second is a series of values loaded into registers, which might be treated as a stream. In either case, the offset of the first value uniquely describes the alignment of the stream. For a register stream, an offset of four indicates an integer value in the second location within a register, and those values following it. For a memory stream, an offset of four indicates an address that is four bytes past a 16-byte boundary.

Abstracting aligning of data

The next stage is to abstract the process by which alignment is changed, by defining a new stream operation, `vshiftstream(x, y)`, which represents shifting a stream with offset `x` to offset `y`. For instance, given a stream with offset four (`b[i+1]` in the example), `vshiftstream(4, 12)` represents shifting it eight bytes to the right, from offset four to offset 12 (suitable for storing into `a[i+3]`). In general, a shift from a lower offset to a higher offset is a *right shift*, and a shift from a higher offset to a lower offset is a *left shift*. If the data to be processed are at a 16-byte boundary, the offset is zero.

Abstracting alignment constraints: data reorganization graph

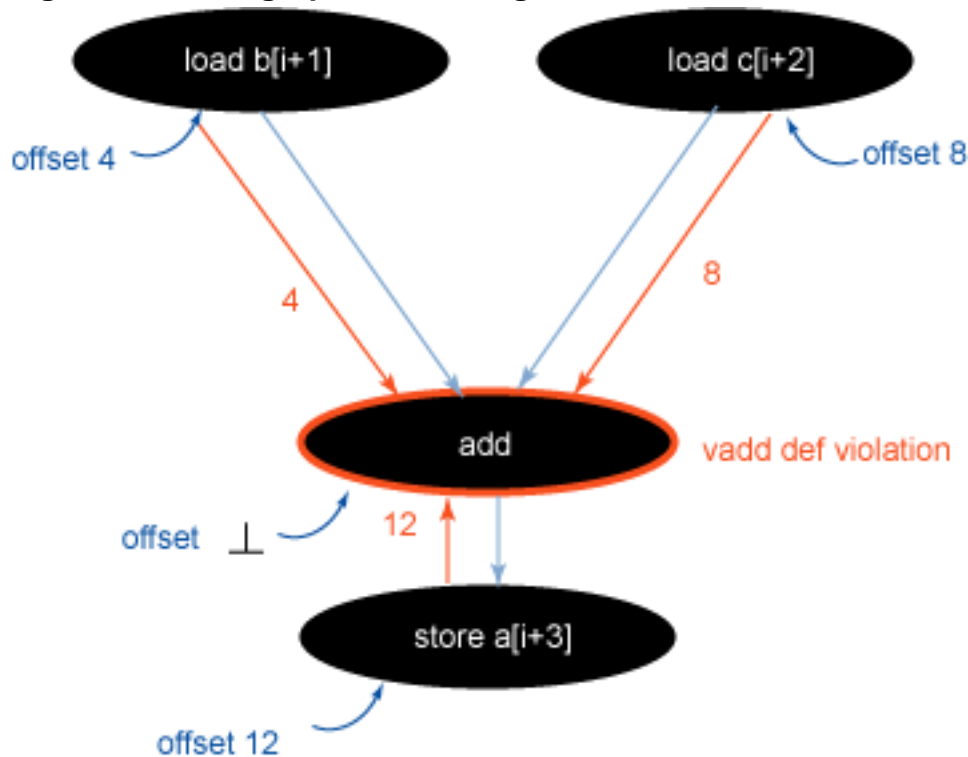
To use these abstractions, you must define the offsets of data and the alignment constraints of operations. For instance, the load of `b[i+1]` has offset four, and the load of `c[i+2]` has offset eight. The addition operator has the constraint that offsets must be identical.

A graph in which all of the constraints are met is called a *valid graph*. A graph in which a constraint is not met is considered an *invalid graph*.

An invalid graph

As an example of an invalid graph, consider the following simple graph:

Figure 3. A data graph with an alignment mismatch



This graph has two objects, one with offset four, one with offset eight, being fed into a `vadd` operation which requires that all inputs have the same offsets. This violates the definition of the `vadd` operation.

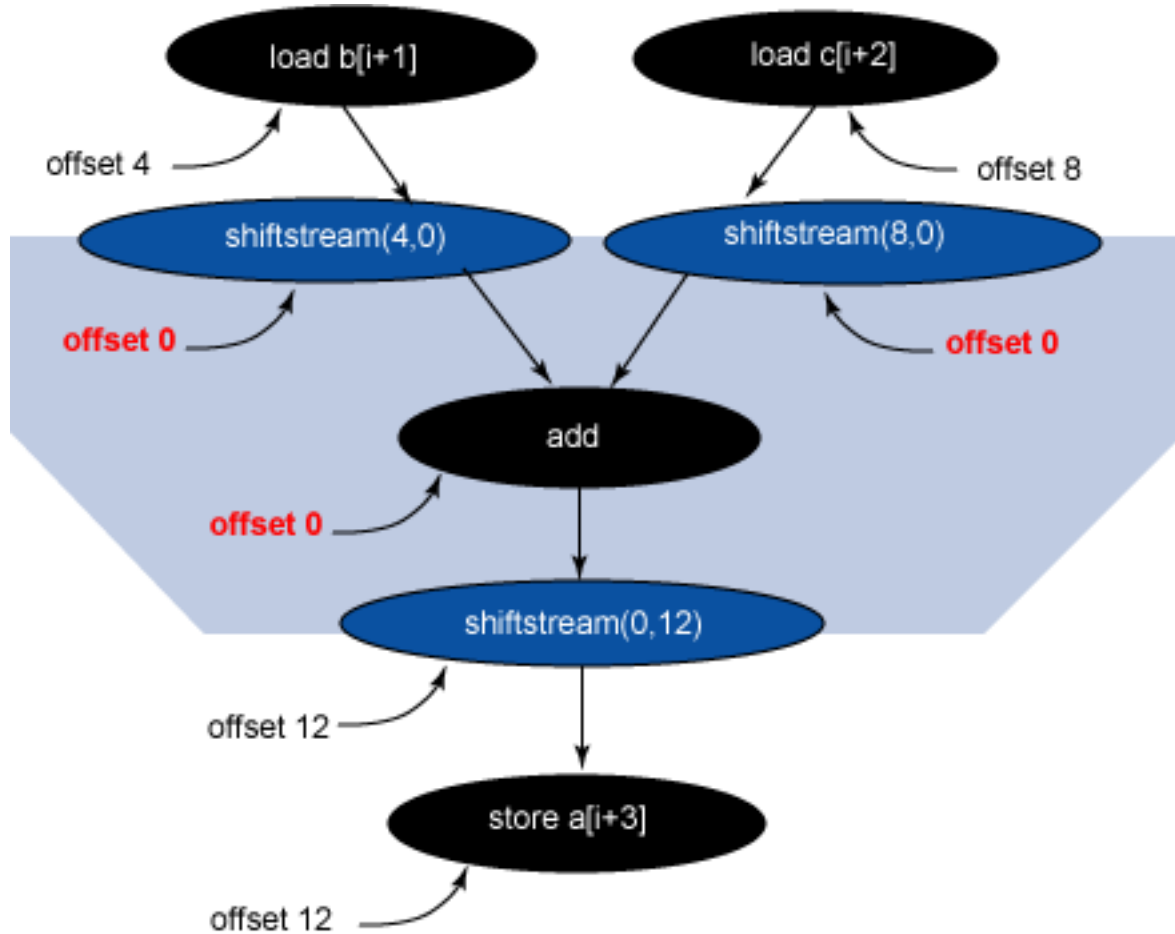
Making a graph valid

To resolve this, the compiler can insert `vshiftstream()` operations at suitable points. For instance, inserting a `vshiftstream(4,12)` operation in the `b[i+1]` stream, and a `vshiftstream(8,12)` operation in the `c[i+2]` stream, makes the `vadd` operation valid, and furthermore makes the following `vstore` operation valid.

Shift stream placement: Zero policy

The simplest policy to implement is to shift all streams to offset zero on load, and from offset zero on store. This is not optimized at all, but is particularly useful for runtime alignment, when the exact alignment changes needed are not known at compile time. With this policy, the `b[i+1]` stream and `c[i+2]` streams are each shifted to an offset of zero and added with an offset of zero; the results of the add are then shifted to an offset of 12 for storing in `a[i+3]`.

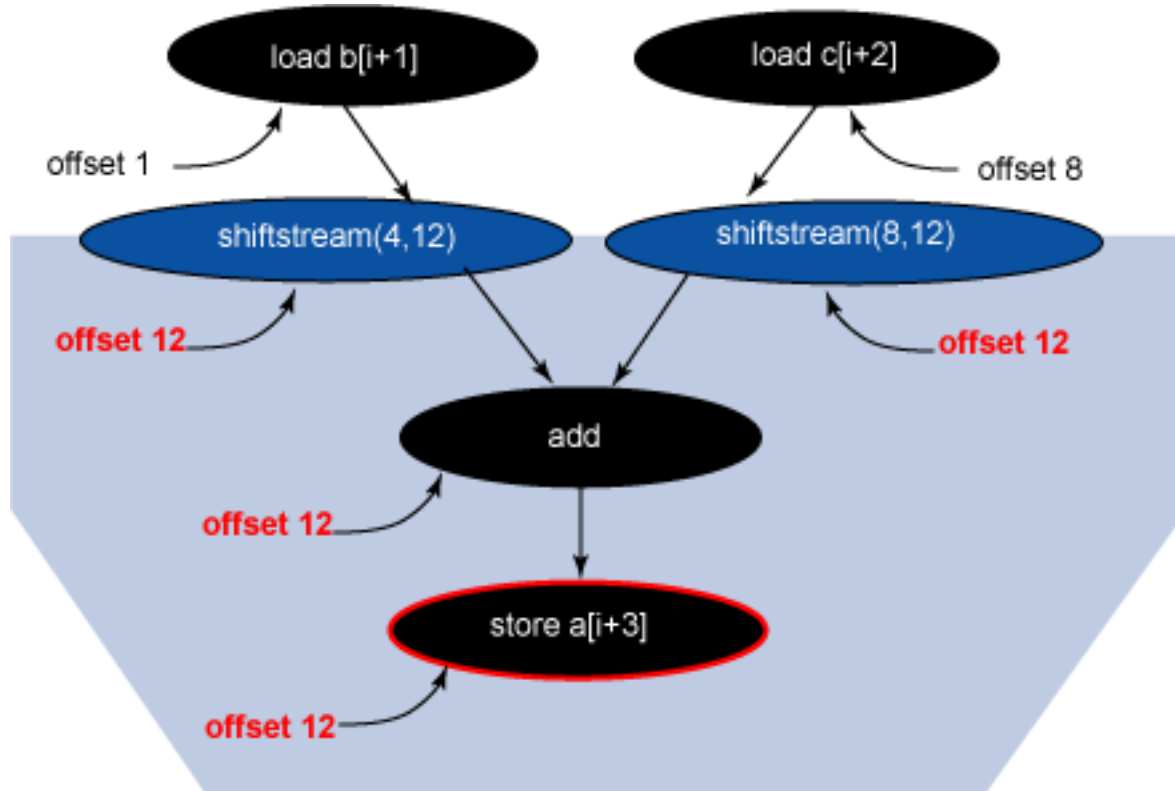
Figure 4. Shift stream placement: zero policy



Shift stream placement: Eager policy

Another policy which requires a bit of lookahead is an eager policy, which shifts to the eventual store offset as quickly as possible. For instance, since $a[i+3]$ has offset 12, the $b[i+1]$ and $c[i+2]$ streams are both shifted to offset 12. In the sample case, this policy requires only two `vshiftstream` operations instead of three.

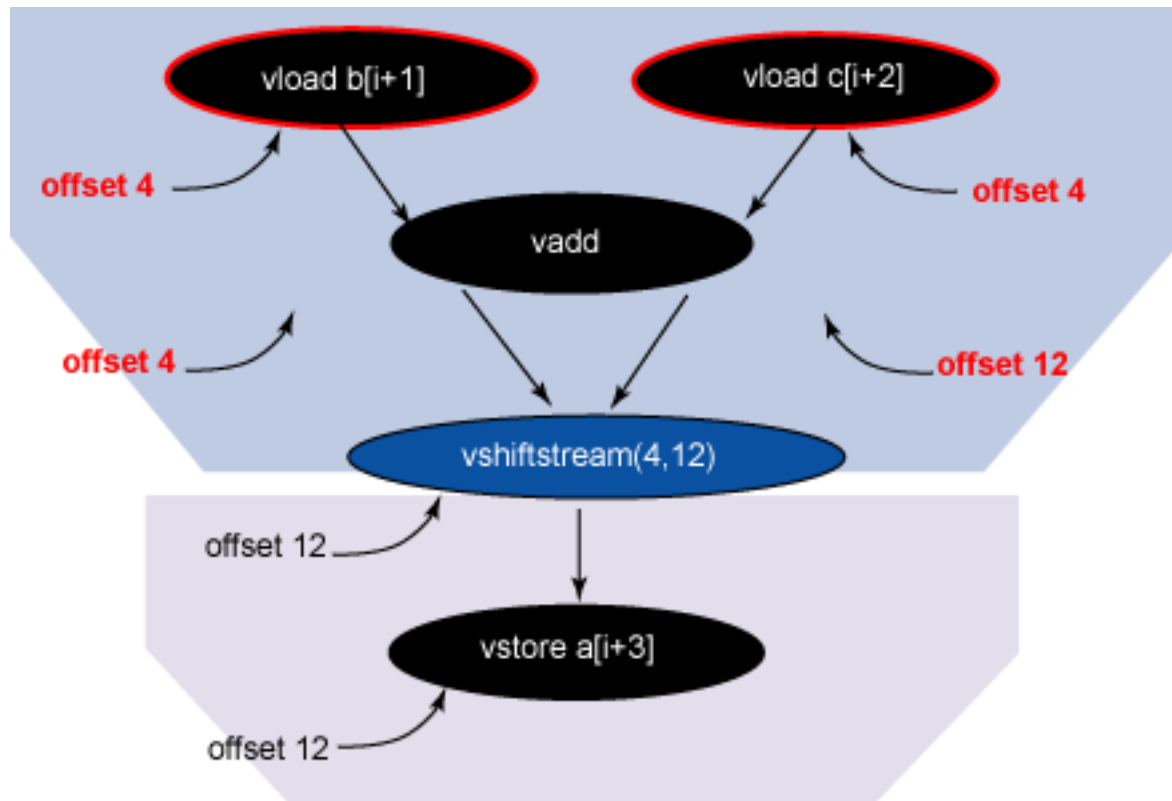
Figure 5. Shift stream placement: eager policy



Shift stream placement: Lazy policy

This policy is similar to the eager policy, but instead of shifting as soon as possible, it delays shifts until they are necessary. For instance, if the loop used $b[i+1]$ and $c[i+1]$, they would be used and added with their existing offset of four, and would only be shifted to the offset of 12 right before the store into $a[i+3]$. This policy can save shifts in many cases. The compiler uses this policy, because it produces the best results on most code.

Figure 6. Shift stream placement: lazy policy



Section 4. SIMD code generation

Overview

SIMD code is generated from a valid data reorganization graph. Stream shifts are managed using SHUFB instructions to merge two vector registers into one vector.

A loop is given a prologue and epilogue which get the data aligned, and then a comparatively simple and efficient simdized loop body; the steady state for the majority of the work should be this inner loop body, with the prologue and epilogue handling the setup and dealing with whatever's left at the end. Either prologue or epilogue may be very short, depending on store alignment and tripcount.

Code generation for stream shift

The stream shift code is used when you need a stream of vectors starting at an unaligned address. This works by loading the first and second vectors in the original

data and shuffling them together to create the first shifted vector. After this, each new vector requires only one load and one shuffle, because the previous load operation provided the first of the two vectors to combine.

Code generation for partial store

Partial stores require a load-modify-save operation. However, this is necessary only at the beginning and end of a stream. For instance, in the example loop, the initial store of `a[3]` (`a[i+3]` when `i` is zero) requires a load of `a[0]` through `a[3]`, a shuffle operation to put the result into the last slot of the register, and a store. After that, however, every time suitably aligned chunks of `b` and `c` are loaded, the results can simply be stored; for instance, `b[2]` through `b[5]` are added to `c[3]` through `c[6]`, and stored as `a[4]` through `a[7]` -- an aligned store which requires no special treatment.

Code generation for loops with multiple statements

Consider the case of the following loop:

```
for (i = 0; i < n; ++i) {  
    a[i] = ...;  
    b[i+1] = ...;  
    c[i+3] = ...;  
}
```

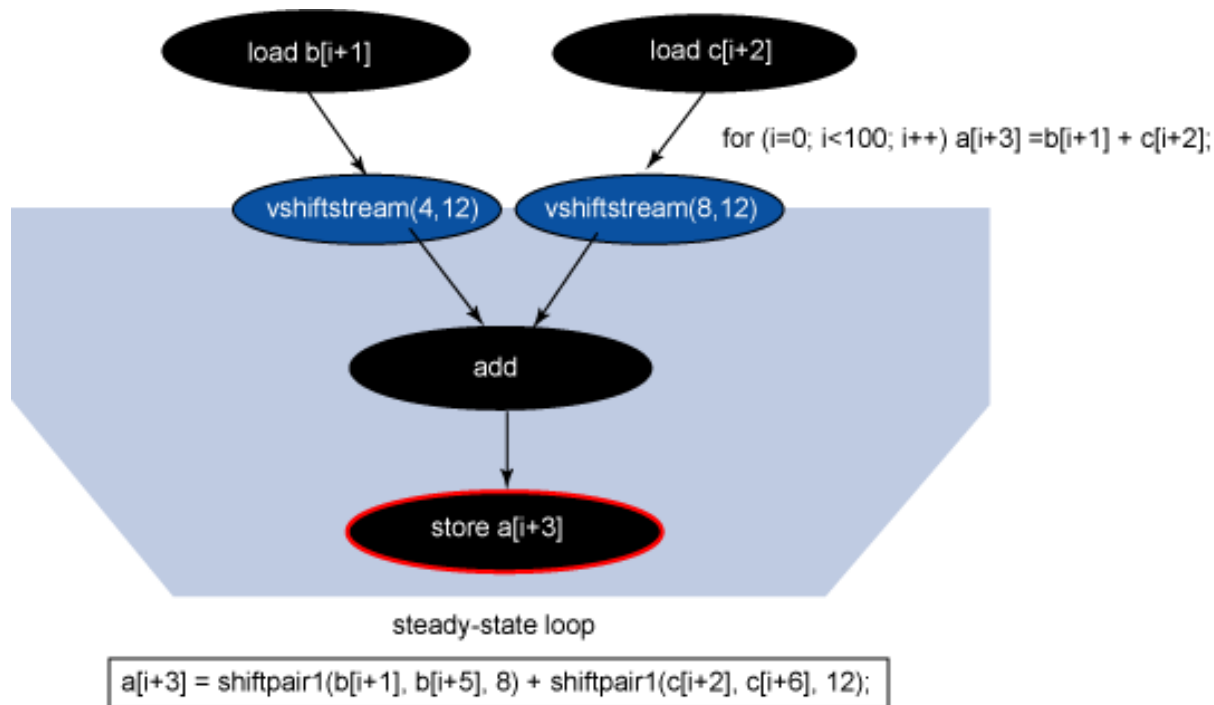
The compiler generates a prologue to handle the first four members of `a`, the first three members of `b`, and the first member of `c`. After this, all three can be handled in aligned chunks, processing `array[i+4]`; when the loop is done, the remaining member of `b`, and the remaining three members of `c`, can be handled separately by the epilogue.

The steady state of the example

Referring back to our example once more, the steady-state loop looks something like this:

```
a[i+3] = shiftpairl(b[i+1], b[i+5], 8) + shiftpairl(c[i+2], c[i+6], 12);
```

Figure 7. Shiftstreams feeding into an add operation.



Pseudocode after simdization

This pseudocode shows the simdization, without any software pipelining on adjacent loads. All operations are vectors; loads and stores are normalized to a truncated address. The `splice`, `shiftpairl`, and `shiftpairr` operations are pseudo operations for data reorganization, which are mapped later to `spu_shuffle` or `spu_sel` instructions.

```

i = 0;
/* prologue */
a[i] = splice(a[i],
             shiftpairl(b[i-4], b[i], 8) + shiftpairl(c[i-4], c[i], 12),
             12)
/* inner loop */
do {
    a[i+4] = shiftpairl(b[i], b[i+4], 8) + shiftpairl(c[i], c[i+4], 12);
    i = i + 4;
} while (i < 95);
/* epilogue */
a[i+4] = splice(a[i],
             shiftpairl(b[i], b[i+4], 8) + shiftpairl(c[i], c[i+4], 12),
             12 + (i * -4 + 384));
    
```

Intrinsic codes after simdization

This is the same example after software pipelining, loop normalization, and address truncation. The `spu_shuffle`, `spu_sel`, `spu_add`, and `spu_mask` operations are

SPU intrinsics; `0x0809010b, . . .` is a vector literal.

```

a[0] = spu_sel(a[0],
spu_add(spu_shuffle(b[-4], b[0],
                 <0x08090a0b,0x0c0d0e0f,0x10111213,0x14151617>),
        spu_shuffle(c[-4],c[0],
                 <0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>),
        spu_maskb(15));
oldSPCopy0 = c[0];
oldSPCopy1 = b[0];
i = 0;
do {
    tc = c[i*4+4];
    tb = b[i*4+4];
    a[i*4+4] = spu_add(spu_shuffle(oldSPCopy1, tb,
                                <0x08090a0b,0x0c0d0e0f,0x10111213,0x14151617>),
                    spu_shuffle(oldSPCopy0, tc,
                                <0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>));

    oldSPCopy0 = tc;
    oldSPCopy1 = tb;
    i = i + 1;
} while (i < 24);
a[100] = spu_sel(spu_add(spu_shuffle(b[96],b[100],
                                <0x08090a0b,0x0c0d0e0f,0x10111213,0x14151617>),
                    spu_shuffle(c[96], c[100],
                                <0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>)),
                a[100], spu_maskb(15));

```

Compiler designers' notes

A few points of interest came up during the development of this aspect of the Cell BE compiler.

There is prior art for using loop peeling to handle misalignment; this simpler technique is only effective if all streams in the computation are relatively aligned. It is equivalent to a data reorganization graph with no stream shift, but with handling of store misalignment.

Store misalignments have implicit skewing effects. The compiler must ensure that the skewing is valid, meaning, if there are multiple possible store misalignments, with different amounts of skew, there might be a need for multiple epilogues.

Runtime properties make things harder for the compiler. Anything the compiler does not know must be handled at runtime. With a runtime stream shift, it's necessary to decide whether to shift left or right, and it's not always easy to tell. Runtime store alignment creates skew determined at runtime, which creates issues for dependence checking and SIMD epilogues. A large number of epilogues might need to be created to handle the possible outcomes, even if only some will ever occur in practice.

In some cases, the compiler can generate two versions of a loop: one for cases where everything is aligned, which can then run at full speed without overhead, and

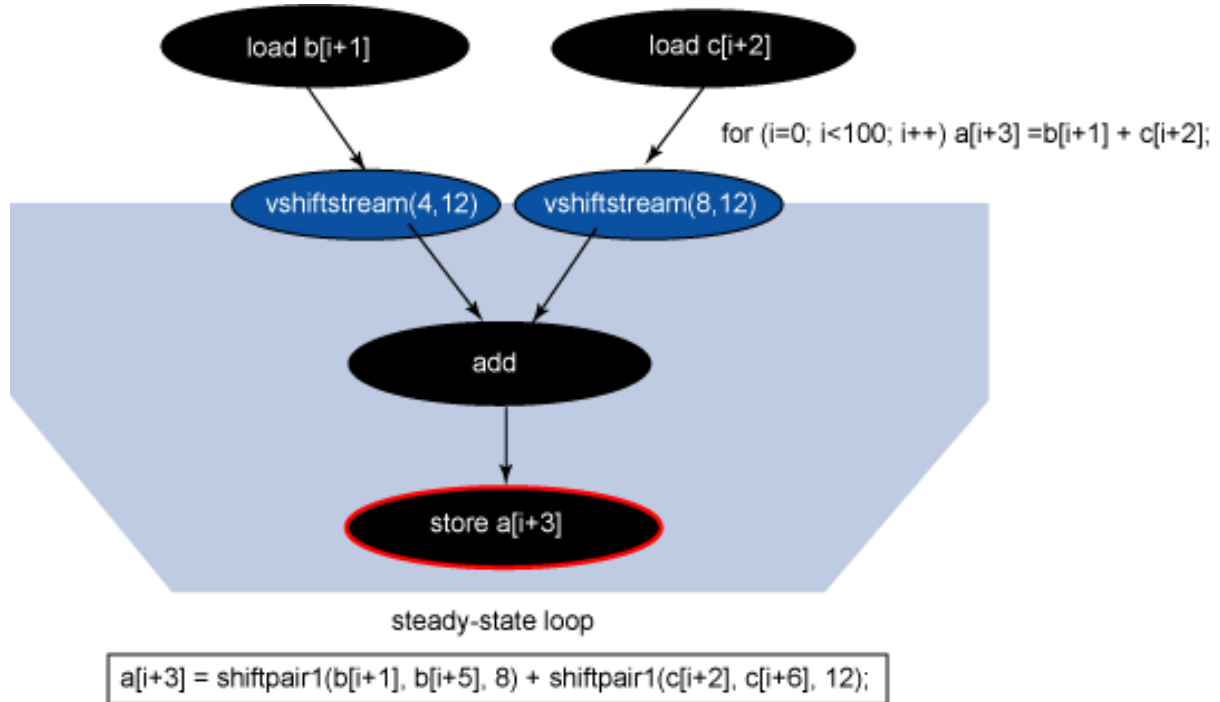
another for cases where alignment fix-up is needed.

Section 5. Managing complexity

How to integrate all of these requirements

The Cell BE compiler deals with a great amount of work: several kinds of parallelism, multiple constraints on generated code, and multiple targets to address. This section looks at ways in which the IBM compiler manages the complexity of compiling for the Cell BE.

Figure 8. Supporting multiple constraints



Multiple sources of SIMD parallelism: Loop level

The first kind of SIMD parallelism is loop-level parallelism, where SIMD is used to execute a single statement across consecutive iterations of a loop. Compiler researchers have been successful at efficiently handling misaligned data, pattern recognition (such as reduction and linear recursion), leveraging loop transformations (in most compilers), amortizing overhead (versioning and alignment handling), and employing cost models.

Multiple sources of SIMD parallelism: Basic-block level

Basic-block level parallelism applies SIMD across multiple isomorphic operations. It is successful at handling unrolled loops (whether unrolled manually or by the compiler), extracting SIMD parallelism within structs (for example, calculations applying to a structure with x, y, and z members), and extracting SIMD parallelism within single statements with complicated but repetitive calculations.

An example of basic-block level simdization

The following example code shows the kind of code which basic-block level parallelism works on:

```
int subdivide() {
    QUAD q;
    while (cnt-- > 0) {
        for (i = 0; i < n; ++i) {
            q = quads[i];
            vertex[i].x = W0*q.p[0].x + W1*q.p[1].x + W2*q.p[2].x + W3*q.p[3].x;
            vertex[i].y = W0*q.p[0].y + W1*q.p[1].y + W2*q.p[2].y + W3*q.p[3].y;
            vertex[i].z = W0*q.p[0].z + W1*q.p[1].z + W2*q.p[2].z + W3*q.p[3].z;
            vertex[i].w = W0*q.p[0].w + W1*q.p[1].w + W2*q.p[2].w + W3*q.p[3].w;
        }
    }
}
```

In this sample, the extraction of the contents of the q.p array members, and the multiplication by constants, provide examples of operations on adjacent fields, which can be handled more efficiently with SIMD instructions.

Multiple sources of SIMD parallelism: Short-loop level

In some cases, if a loop is small enough, SIMD can be applied across the entire loop, effectively collapsing it entirely. This can allow for attempts to do the same thing at the next level. As an example, consider this code:

```
for (k = 0; k < 248; ++k)
    for (i = 0; i < 8; ++i)
        res[k] += in[k+i] * coef[k+i];
```

If the inner loop is reduced to a small number of SIMD instructions, the outer loop will be easier to optimize.

Section 6. How to support the cross product of all of these?

Overview

All of the components contributing to the complications of this compiler have mutual interactions, making the whole more complicated than the sum of its parts. This section reviews some of the approaches taken to supporting, not just each of these features, but all of them.

Key abstraction: Virtual SIMD vector

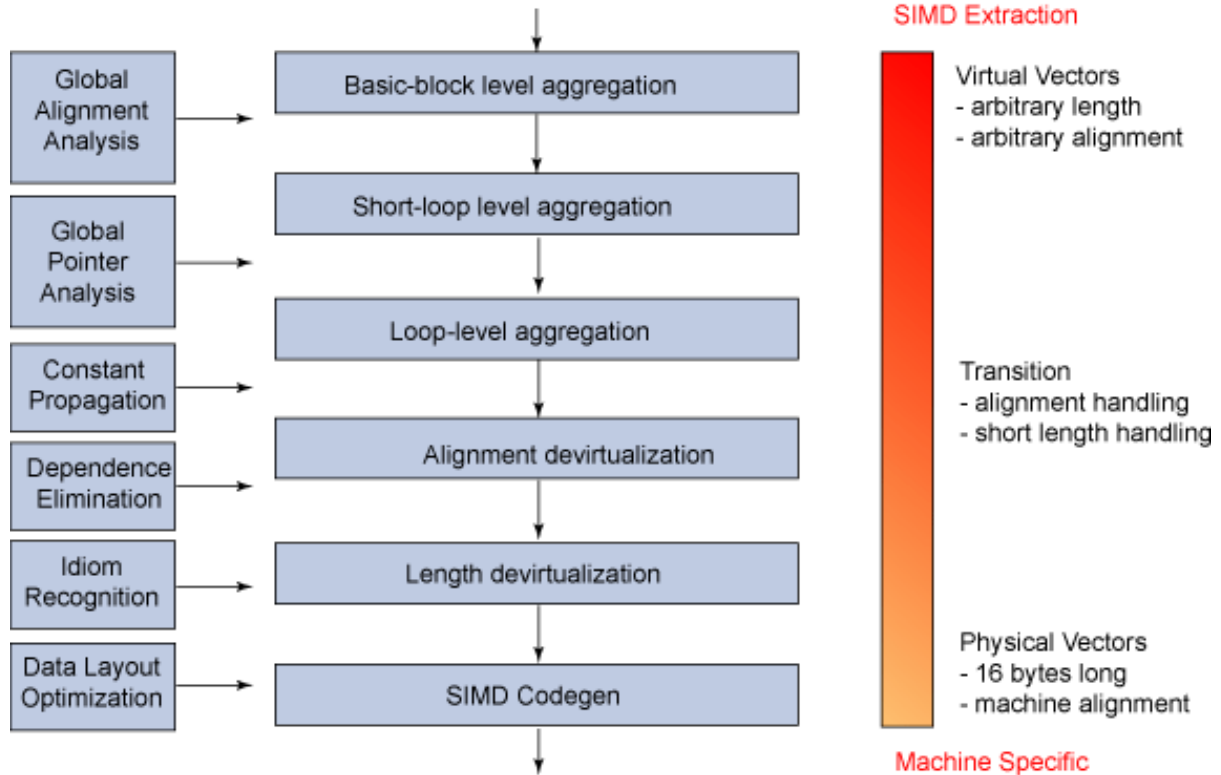
A key abstraction used in the compiler is a virtual SIMD vector. This has arbitrary length and no alignment constraints. The SIMD vector is used to extract SIMD parallelism, using a virtual vector as a representation, and abstracting away all of the hardware complexity.

The virtual SIMD vector can then be progressively "de-virtualized," until each vector in the loop satisfies all of the hardware constraints. In extreme cases, if this imposes too much overhead, vectors can be reverted back to scalars.

Integrated simdization framework

The integrated framework is modular and integrated, and hides the underlying hardware complexity.

Figure 9. The progression from virtual vectors to physical vectors



The compiler starts by aggregating operations into virtual SIMD vectors: first basic-block aggregation, then short-loop aggregation, and finally generic loop aggregation. Once this is done, the virtual vectors are subjected first to alignment devirtualization (handling the alignment issues of the hardware) and then to length devirtualization (handling the finite size of the vector hardware operations); the results can be passed on for actual SIMD code generation.

Section 7. Example 1: Basic-block and loop-level aggregation

Example code

This example uses the following simple loop to illustrate how the integrated simdization framework applies in practice:

```
for (i = 0; i < 256; ++i) {
    a[i].x = ...;
    a[i].y = ...;
}
```

```

    a[i].z = ...;
    b[i+1] = ...;
}

```

Value streams

The following shows the value streams used by the first four iterations of the sample loop:

```

a0.x a0.y a0.z a1.x a1.y a1.z a2.x a2.y a2.z a3.x a3.y a3.z
  b1          b2          b3          b4

```

Figure 10. The original value streams



Basic-block level aggregation

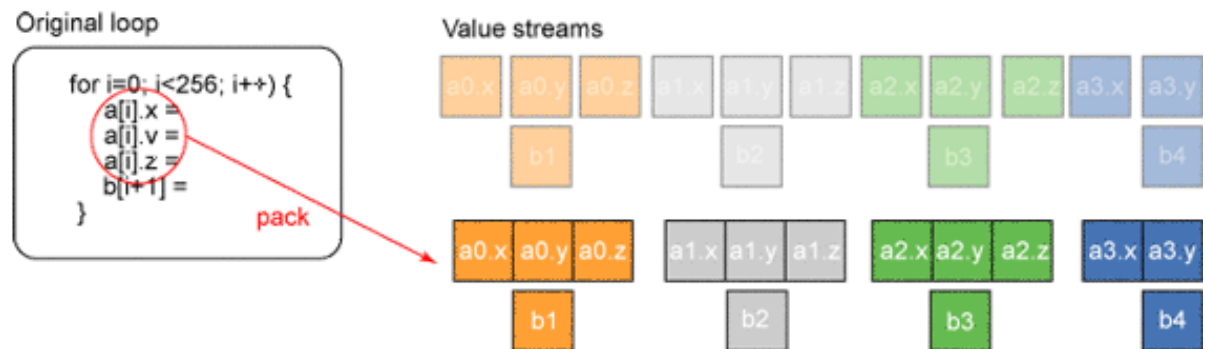
The first pass of aggregating values is to combine the references to x, y, and z together. They are packed regardless of alignment, simply because the statements are isomorphic. Each object is being packed with other objects near it.

```

a0.x,a0.y,a0.z a1.x,a1.y,a1.z a2.x,a2.y,a2.z a3.x,a3.y,a3.z
  b1          b2          b3          b4

```

Figure 11. Basic-block level aggregation



The loop now looks, conceptually, like this:

```
for (i = 0; i < 256; ++i) {
    (a[i].x,y,z) = ...;
    b[i+1] = ...;
}
```

Loop-level aggregation

Now, loop-level aggregation lets us combine like objects into streams of arbitrary length, still ignoring alignment requirements:

```
a0.x,a0.y,a0.z,a1.x,a1.y,a1.z,a2.x,a2.y,a2.z,a3.x,a3.y,a3.z
b1,b2,b3,b4
```

Figure 12. Loop-level aggregation



This packs each element with itself over other iterations, rather than with other things within a single iteration. At this point, final vector lengths must be multiples of 16 bytes. Scalars and vectors are treated alike during this phase, and packing ignores alignment.

Conceptually, the code is now:

```
for (i = 0; i < 256; i += 4) {
    (a[i].x, ..., a[i+3].z) = ...;
    (b[i+1], ..., b[i+4]) = ...;
}
```

Alignment devirtualization

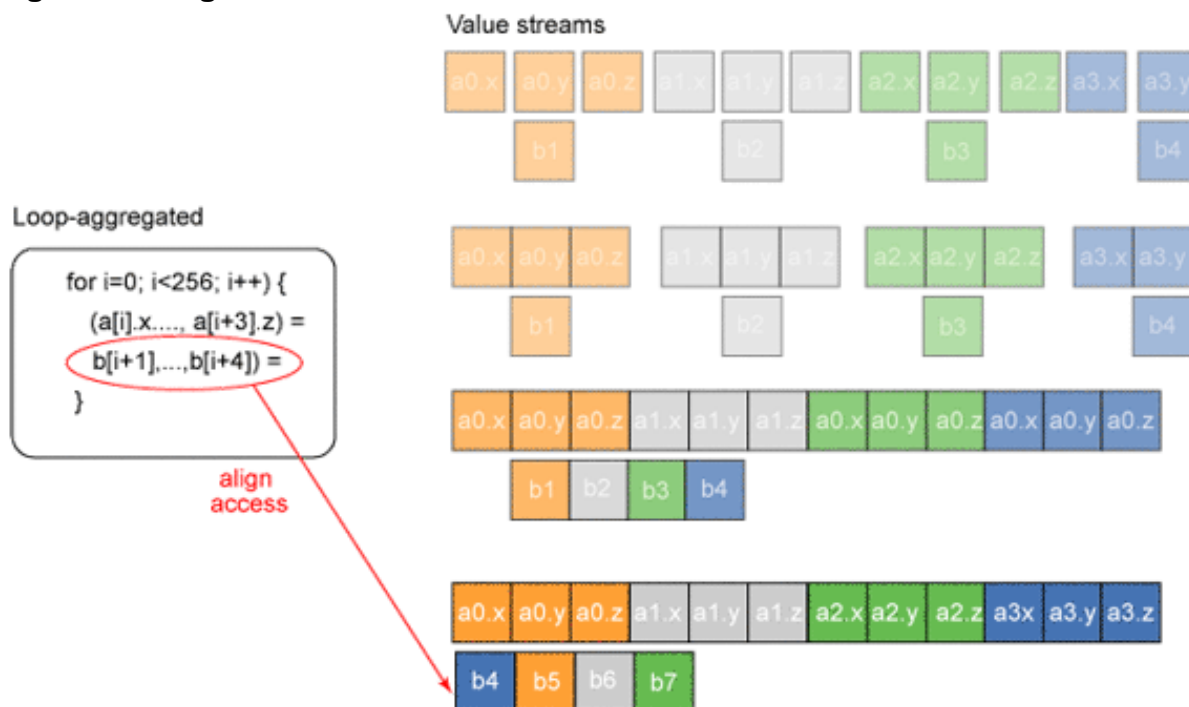
At this point, the misaligned stream is shifted; computations are skewed so that the loop computes $b[i+4]$ through $b[i+7]$.

The code has been reorganized a bit further:

```

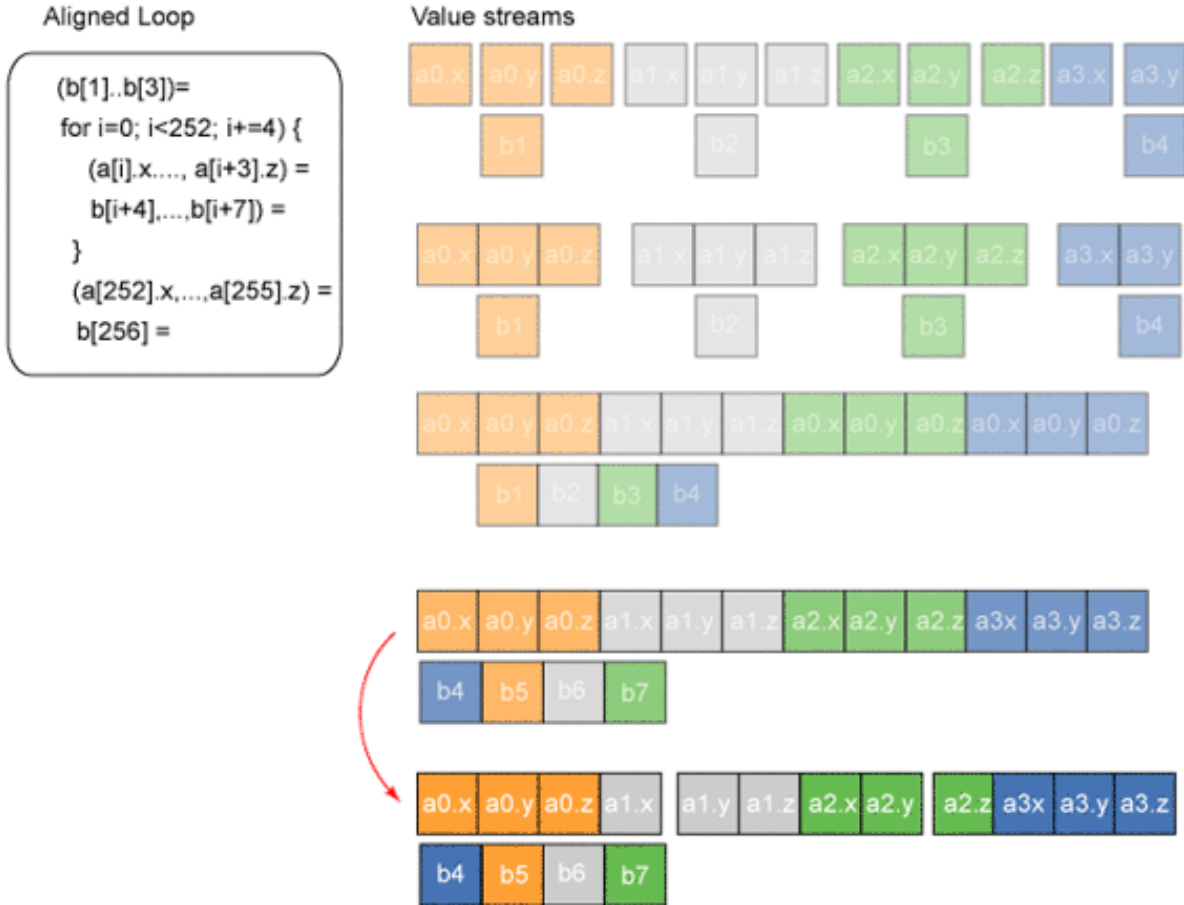
(b[1], ..., b[3]) = ...;
for (i = 0; i < 252; i += 4) {
    (a[i].x, ..., a[i+3].z) = ...;
    (b[i+1], ..., b[i+4]) = ...;
}
(a[252].x, ..., a[255].z) = ...;
b[256] = ...;
    
```

Figure 13. Alignment devirtualization



At this point, all that remains is to break the streams into 16-byte chunks, which can be processed accordingly.

Figure 14. Length devirtualization



Section 8. Example 2: Data-size conversion and misalignment

An example with mismatched data sizes

This section reviews another example, based on the following code:

```
for (i = 0; i < 256; ++i) {
    a[i] += (int) b[i+1];
}
```

In this example, the array `b` is assumed to hold `short` values (2 bytes), which must be converted to `int` values to be added into the members of `a`.

Phase 1: Loop-level aggregation

In the first phase of optimization, the statement is packed with itself across consecutive iterations of the loop. Virtual vectors have a uniform number of elements, even though a vector of eight `int` objects would be 32 bytes, and a vector of eight `short` objects would be only 16.

Phase 2: Alignment devirtualization

In the second phase, the `b[i+1]` references are aligned. Because the vectors are long, this is efficient and simple. So, a `vstreamshift` operation is put into the data path for the objects coming out of the array `b`.

Phase 3: Length devirtualization

This is the phase where the mismatched lengths come into play. Eight `short` objects fit into a single register, but only four `int` objects. Thus, the code to load integers from `a` must be duplicated, loading from `a[i]` and from `a[i+4]`. The single register taken from `b` is unpacked into two registers, each holding four `int` objects; then these are added to the two vectors taken from `a`, and stored.

Section 9. Notes and conclusions

Compiler designers' notes

As with the section on techniques, the compiler designers wanted to call attention to some specific points:

Managing complexity includes both interactions among different phases in simdization and interactions between the simdization code and other compiler optimizations.

The compiler uses an internal representation of vectors. The virtual-length vectors capture the effects of different aggregation phases. Using generic operations in the internal representation helps in supporting multiple platforms.

Auxiliary analysis and transformations are also important. These include alignment

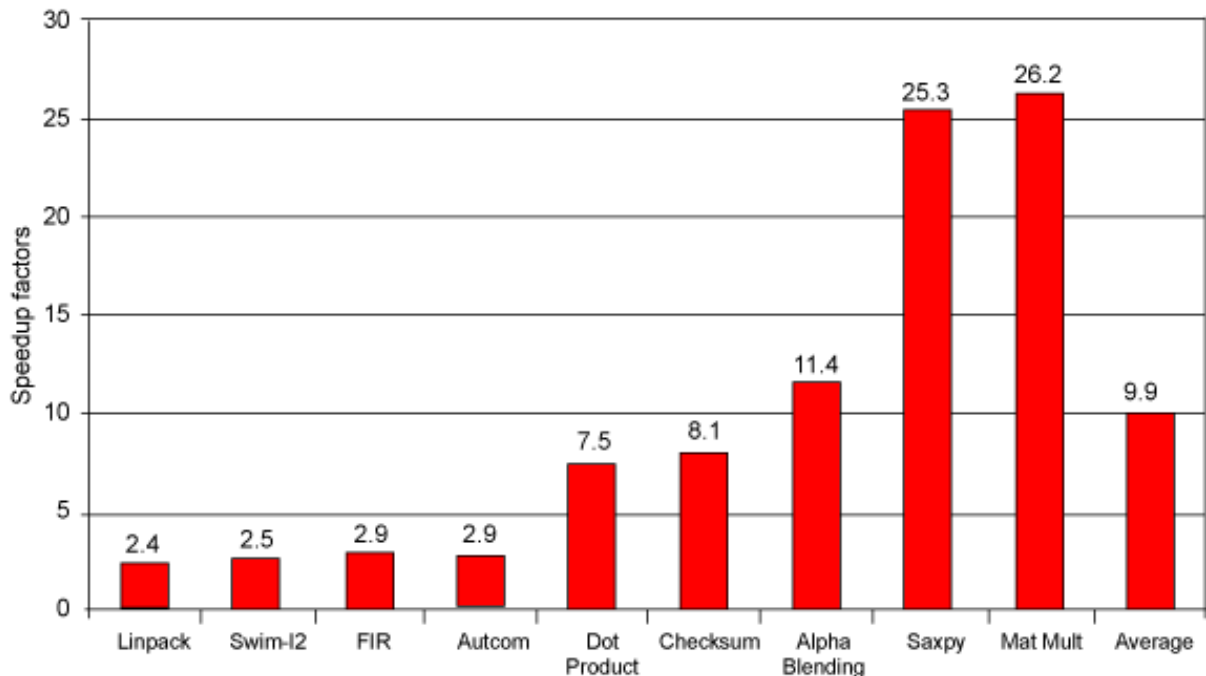
analysis, pointer analysis, and dependence analysis. Redundant conversion elimination has a significant effect. Finally, data layout optimizations can fix alignment problems and provide for stride-one access to data, reducing the overhead imposed by hardware constraints.

On a processor such as the PowerPC®, or the PPE of the Cell BE, it's important to make sure that simdized code is actually more efficient than the code it replaces; the setup costs of a simdized loop might be higher than its benefit. On the SIMD-only SPEs, however, it doesn't matter nearly as much, because scalar code has very high overhead. Thus, while it's important to track the overhead and complexity of the simdized code in general, the SPEs will nearly always end up using it even when it's somewhat expensive.

One hidden cost of simdization is that the resulting code might be much harder to optimize efficiently, decreasing performance.

Results

Figure 15. Speedups obtained through automatic simdization



This chart shows the speedups of optimized code with automatic simdization against scalar code, running on a single SPE; for instance, if unoptimized code ran in four seconds, and optimized code ran in one second, the speedup would be four.

Conclusion

The IBM compiler has a workable integrated and modular approach to simdization. It extracts SIMD parallelism at multiple levels, efficiently handles constraints such as alignment and data conversion, and can target multiple ISAs, such as VMX and the SPU. The next tutorial in this series looks at partitioning and parallelization techniques to allow code to run on multiple processor elements at once, bringing us from optimizations of code for a single element into optimizations for the processor as a whole.

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 3 is based on the section "Automatic Simdization."

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](#).
- For examples of ray-tracing code, see the Apple doc "[Data Handling and Data Formats](#)" and Ryan Michael Schmidt's (one of the "other" RMS's) "[Transforming Normals: A Tutorial](#)."
- "[IBM's Blue Gene/L Compiler Implementation](#)" by Mark Mendell and Roch Archambault offers a more general overview of the BlueGene/L compiler.
- "[Efficient SIMD Code Generation for Runtime Alignment and Length Conversion](#)" by Peng Wu, Alexandre E. Eichenberger, and Amy Wang presents two major enhancements to the state of the art in both performance and coverage.

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.

Trademarks

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