CS:APP2e Web Aside OPT:SIMD: Achieving Greater Parallelism with SIMD Instructions[*]

Randal E. Bryant
David R. O'Hallaron

June 5, 2012

Notice

1 Introduction

As described in CS:APP2e Section 3.1, Intel introduced the SSE instructions in 1999, where SSE is the acronym for "Streaming SIMD Extensions," and, in turn, SIMD (pronounced "sim-dee") is the acronym for "Single-Instruction, Multiple-Data." The idea behind the SIMD execution model is that each 16-byte XMM register can hold multiple values. In our examples, we will consider the cases where they can hold either four integer or single-precision values, or two double-precision values. SSE instructions can then perform vector operations on these registers, such as adding or multiplying four or two sets of values in parallel. For example, if XMM register `%xmm0` contains four single-precision floating-point numbers, which we denote $a_0, \ldots, a_3$, and `%rcx` contains the memory address of a sequence of four single-precision floating-point numbers, which we denote $b_0, \ldots, b_3$ then the instruction

```
    mulps  (%rcx), %xmm0
```

will read the four values from memory and perform four multiplications in parallel, computing $a_i \leftarrow a_i \cdot b_i$, for $0 \leq i \leq 3$. We see that a single instruction is able to generate a computation over multiple data values, hence the term "SIMD." This multiplication is an example of what we will refer to as *vector code.* We will refer to code that operates only on one value at a time as *scalar code*.

[*]Copyright © 2010, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

GCC supports extensions to the C language that let programmers express a program in terms of vector operations that can be compiled into the SIMD instructions of SSE [1]. This coding style is preferable to writing code directly in assembly language, since GCC can also generate code for the SIMD instructions found on other processors. Writing in C also has the advantage that GCC will generate scalar code for machines that do not support vector instructions. We will describe how to write code using the GCC support for vector operations, using our combining functions as examples. The basic strategy is to define a vector data type `vec_t` that holds either four 4-byte values or two 8-byte values. If we have two such vectors `va` and `vb`, then the expression `va * vb` causes a SIMD multiplication of the vector elements.

2 Declarations

Our first step is to declare the vector data type. Since we are trying to make the same code work for base data types `int`, `float`, and `double`, we will use a combination of `typedef` declarations and constant definitions to make the code more general. As with our earlier versions, we assume that the base data type has been declared as type `data_t`.

We define `VBYTES` to be the number of bytes in a vector. For SSE, this is defined to be 16, but we would like to keep this value parameterized to easily adapt the code for other machines. We then defined `VSIZE` to be the number of elements in each vector:

```
/* Number of bytes in a vector */
#define VBYTES 16

/* Number of elements in a vector */
#define VSIZE VBYTES/sizeof(data_t)
```

We are now ready to define the vector data type. This involves a notation that is idiosyncratic to GCC:

```
/* Vector data type */
typedef data_t vec_t __attribute__ ((vector_size(VBYTES)));
```

This declaration states that data type `vec_t` is vector, where the elements are of type `data_t`, and the vector consists of `VBYTES` bytes.

We now need some means of accessing the elements of a vector. Rather than introducing additional notation, we can make use of the `union` declaration to overlay vector and array data types:

```
typedef union {
    vec_t v;
    data_t d[VSIZE];
} pack_t;
```

The following shows a simple example of computing the inner product of two SIMD vectors:

```
/* Compute inner product of SSE vector */
data_t innerv(vec_t av, vec_t bv) {
 3      pack_t xfer;
 4      int i;
 5      vec_t pv = av * bv;
 6      data_t result = 0;
 7      xfer.v = pv;
 8      for (i = 0; i < VSIZE; i++)
 9          result += xfer.d[i];
10      return result;
11  }
```

In this code, the multiplication operation on line 5 multiplies the corresponding elements of vectors `av` and `bv`. The code on lines 7–10 then accesses and sums the elements of the product vector. Using the variable `xfer`, of type `pack_t`, the code provides access to each element i of the vector with the expression `xfer.d[`i`]`.

3 Alignment Requirement

Many of the SSE instructions impose a very strict alignment requirement on memory operands. They require that any data being read from memory into an XMM register, or written from an XMM register to memory, satisfy a 16-byte alignment. An instruction that attempts to read or write unaligned data causes a segmentation fault, indicating an invalid memory reference. This alignment requirement will factor into how we write programs that make use of SSE instructions. (Starting with SSE2, there are SSE instructions that can access unaligned data, but early implementations were not very efficient, and so GCC does not currently generate code that uses them.)

4 Implementation of Combining Function

Figure 1 shows the code for a combining function that makes use of the SIMD operations. The overall idea is to set up a vector variable `accum` that accumulates either four (data types `int` and `float`) or two (data type `double`) values in parallel.

The code starts by initializing the accumulators to the identity element (lines 11–13), using the `pack_t` data type to set the individual elements of a vector.

To satisfy the alignment requirement, we may need to accumulate several vector elements using scalar operations until the remaining data vector `data` has an address that is a multiple of `VBYTES`. The code for this is shown in lines 16–19. Observe we cast pointer `data` to data type `long` so that we can test whether it is a multiple of `VBYTES`. We must also keep track of the number of remaining elements `cnt`, and consider the case where `cnt` is smaller than the number of elements in a single vector.

Lines 22–27 show the main loop for the function. We see here the use of casting to create a pointer to a vector having the same address as the pointer to the data. Dereferencing this pointer then retrieves an entire vector of data from memory, defined here by vector variable `chunk`. The statement `accum = accum OP chunk` then combines the vector values read from memory with the values in the parallel accumulators.

```
void simd_v1_combine(vec_ptr v, data_t *dest)
{
    long int i;
    pack_t xfer;
    vec_t accum;
    data_t *data = get_vec_start(v);
    int cnt = vec_length(v);
    data_t result = IDENT;

    /* Initialize accum entries to IDENT */
    for (i = 0; i < VSIZE; i++)
        xfer.d[i] = IDENT;
    accum = xfer.v;

    /* Single step until have memory alignment */
    while (((long) data) % VBYTES && cnt) {
        result = result OP *data++;
        cnt--;
    }

    /* Step through data with VSIZE-way parallelism */
    while (cnt >= VSIZE) {
        vec_t chunk = *((vec_t *) data);
        accum = accum OP chunk;
        data += VSIZE;
        cnt -= VSIZE;
    }

    /* Single-step through remaining elements */
    while (cnt) {
        result = result OP *data++;
        cnt--;
    }

    /* Combine elements of accumulator vector */
    xfer.v = accum;
    for (i = 0; i < VSIZE; i++)
        result = result OP xfer.d[i];

    /* Store result */
    *dest = result;
}
```

Figure 1: **Combining function using SIMD operations.** The vector operations cause multiple (2 or 4) values to be accumulated in parallel in variable `accum`.

In the event that the main loop terminates before all values have been accumulated, we have another loop to single step through the remaining elements (lines 30–33.)

We then reference the accumulators through a `union` and combine them to accumulate the final result (lines 36–38.)

5 Analysis

The following table shows our results for the code of Figure 1, compared to our best method using only scalar operations:

Method	`int`		`float`		`double`	
	+	*	+	*	+	*
Unroll by $\times 5$, parallelism $\times 5$	1.01	1.00	1.00	1.00	1.00	1.00
SIMD	0.50	1.62	0.75	1.00	1.62	2.97

Note that we now list the performance for single and double-precision addition separately. Although these two cases have identical performance for scalar code, they have different performance with SIMD code, since one achieves 4-way parallelism, while the other just 2-way. We see that the resulting performance is a bit mixed. For the case of integer addition and single-precision addition, we have broken the throughput barrier of 1.00 that we have seen for scalar implementations. On the other hand, the other operations did not perform as well as we achieved using scalar code.

Fortunately, we can combine our earlier techniques for further enhancing parallelism either by expanding the number of accumulators (see Problem 1) or by using reassociation (see Problem 2), yielding the following performance:

Method	`int`		`float`		`double`	
	+	*	+	*	+	*
SIMD	0.50	1.62	0.75	1.00	1.62	2.97
SIMD + 8-way parallelism	0.25	0.55	0.25	0.24	0.55	0.58
SIMD + 8-way reassociation	0.25	0.54	0.25	0.24	0.53	0.57
Throughput bound	0.25	0.50	0.25	0.25	0.50	0.50

This table also shows the throughput bounds for these computations, set by a combination of the number of elements operated on in parallel (4 or 2), and the issue time for the operations. Integer multiplication has an issue time of 2 cycles, limiting the CPE for this case to 0.50. We see that our functions nearly achieve the throughput bound.

6 Problems

Practice Problem 1:

Write vector code for the combining function that maintains four different sets of accumulators, each accumulating either two or four values, depending on the data type.

Practice Problem 2:

Write vector code for the combining function that reads four 16-byte chunks from memory on each iteration, and then uses reassociation to increase the number of combining operations that can be performed in parallel.

Practice Problem 3:

Write a SIMD version of the inner-product computation described in CS:APP2e Problem 5.15, using a single vector variable to accumulate multiple sums in parallel. You cannot assume that the argument vectors satisfy a 16-byte alignment. You can assume, however, that any degree of misalignment will be the same for both. In other words, for pointer p, the expression `((long)` p`) % 16` will yield the same value when p is `udata` as it will when p is `vdata`.

Our implementation of this function achieves a CPE of 0.75 for integer and single-precision data, and 1.50 for double-precision data.

Practice Problem 4:

Extend your code for Problem 3 to accumulate sums in two vectors. Our implementation of this function achieves a CPE of around 0.50 for integer and single-precision data, and 1.00 for double-precision data, reaching the throughput limit imposed by the load unit.

Practice Problem 5:

Write a SIMD version of the polynomial evaluation described in CS:APP2e Problem 5.5, using a single vector variable to accumulate multiple sums in parallel. Your code must work correctly regardless of the alignment of argument `a`. Our implementation of this function achieves a CPE of 2.50, twice the performance of the basic scalar implementation.

Practice Problem 6:

Use the various tricks you have learned: vector code, multiple accumulators, reassociation, and Horner's method to write the fastest polynomial evaluation function you can. Our code achieved a CPE of 1.00.

Practice Problem Solutions

Problem 1 Solution: [Pg. 5]

This code combines the style we have seen for parallel accumulation with vector code. Here is the main loop for the function:

```
/* Accumulate with 4x VSIZE parallelism */
while (cnt >= 4*VSIZE) {
    vec_t chunk0 = *((vec_t *) data);
    vec_t chunk1 = *((vec_t *) (data+VSIZE));
    vec_t chunk2 = *((vec_t *) (data+2*VSIZE));
    vec_t chunk3 = *((vec_t *) (data+3*VSIZE));
```

```
        accum0 = accum0 OP chunk0;
        accum1 = accum1 OP chunk1;
        accum2 = accum2 OP chunk2;
        accum3 = accum3 OP chunk3;
        data += 4*VSIZE;
        cnt -= 4*VSIZE;
    }
```

The following code then combines the results for all of the accumulators:

```
    /* Combine into single accumulator */
    xfer.v = (accum0 OP accum1) OP (accum2 OP accum3);

    /* Combine results from accumulators within vector */
    for (i = 0; i < VSIZE; i++)
        result = result OP xfer.d[i];
```

Problem 2 Solution: [Pg. 6]

This version only requires modifying the main loop:

```
    while (cnt >= 4*VSIZE) {
        vec_t chunk0 = *((vec_t *) data);
        vec_t chunk1 = *((vec_t *) (data+VSIZE));
        vec_t chunk2 = *((vec_t *) (data+2*VSIZE));
        vec_t chunk3 = *((vec_t *) (data+3*VSIZE));
        accum = accum OP
            ((chunk0 OP chunk1) OP (chunk2 OP chunk3));
        data += 4*VSIZE;
        cnt -= 4*VSIZE;
    }
```

As this example shows, we can enhance parallelism by reassociating the vector operations, just as we did for scalar operations.

Problem 3 Solution: [Pg. 6]

This code is a direct adaptation of the SIMD version of the combining function:

```
void inner_simd_v1(vec_ptr u, vec_ptr v, data_t *dest)
{
    long int i;
    pack_t xfer;
    vec_t accum;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    int cnt = vec_length(v);
    data_t result = 0;

    /* Initialize accum entries to 0 */
    for (i = 0; i < VSIZE; i++)
        xfer.d[i] = 0;
    accum = xfer.v;

    /* Single step until have memory alignment */
    while (((long) udata) % VBYTES && cnt) {
        result += *udata++ * *vdata++;
        cnt--;
    }

    /* Step through data with VSIZE-way parallelism */
    while (cnt >= VSIZE) {
        vec_t uchunk = *((vec_t *) udata);
        vec_t vchunk = *((vec_t *) vdata);

        accum = accum + (uchunk * vchunk);
        udata += VSIZE;
        vdata += VSIZE;
        cnt -= VSIZE;
    }

    /* Single-step through remaining elements */
    while (cnt) {
        result += *udata++ * *vdata++;
        cnt--;
    }

    /* Combine elements of accumulator vector */
    xfer.v = accum;
    for (i = 0; i < VSIZE; i++)
        result += xfer.d[i];

    /* Store result */
    *dest = result;
}
```

Problem 4 Solution: [Pg. 6]

The following shows the inner loop for this function

```
    /* Step through data with 2*VSIZE-way parallelism */
    while (cnt >= 2*VSIZE) {
        vec_t uchunk = *((vec_t *) udata);
        vec_t vchunk = *((vec_t *) vdata);
        accum0 = accum0 + (uchunk * vchunk);
        udata += VSIZE;  vdata += VSIZE;

        uchunk = *((vec_t *) udata);
        vchunk = *((vec_t *) vdata);
        accum1 = accum1 + (uchunk * vchunk);
        udata += VSIZE; vdata += VSIZE;

        cnt -= 2*VSIZE;
    }
```

Problem 5 Solution: [Pg. 6]

Our code accumulates multiple sums in parallel, using a vector `xpwrv` with elements $x^i, x^{i+1}, \ldots$ when executing the loop where pointer `a` is the address of coefficient a_i.

```
double poly_simd_v1(double a[], double x, int degree)
{
    long int i;
    pack_t xfer;
    vec_t accum;
    int cnt = degree+1;
    double result = 0;
    double xpwr = 1.0; /* Various powers of x */

    vec_t xvv;   /* Vector of x^{VSIZE} */
    vec_t xpwrv; /* Vector of increasing powers of x */

    /* Initialize accum entries to 0, and compute x^{VSIZE} */
    xpwr = 1.0;
    for (i = 0; i < VSIZE; i++) {
        xfer.d[i] = 0;
        xpwr *= x;
    }
    accum = xfer.v;

    /* Create a vector of all x^{VSIZE} */
    for (i = 0; i < VSIZE; i++)
        xfer.d[i] = xpwr;
    xvv = xfer.v;

    xpwr = 1;
    /* Single step until have memory alignment */
    while (((long) a) % VBYTES && cnt) {
        result += *a++ * xpwr;
        xpwr *= x;
        cnt--;
    }

    /* Create vector with values xpwr, xpwr*v, ... */
    for (i = 0; i < VSIZE; i++) {
        xfer.d[i] = xpwr;
        xpwr *= x;
    }
    xpwrv = xfer.v;

    /* Main loop.  Accumulate sums in parallel */
    while (cnt >= VSIZE) {
        vec_t chunk = *((vec_t *) a);
        accum += chunk * xpwrv;
        xpwrv *= xvv;
        a += VSIZE;
        cnt -= VSIZE;
    }

    /* Extract accumulated values */
    xfer.v = accum;
    xpwr = 1.0;
    for (i = 0; i < VSIZE; i++)
        result += xfer.d[i];

    /* Get highest power of x */
    xfer.v = xpwrv;
    xpwr = xfer.d[0];
    /* Single step remaining elements */
    while (cnt) {
        result += *a++ * xpwr;
        xpwr *= x;
        cnt--;
    }

    return result;
}
```

Problem 6 Solution: [Pg. 6]

Solution available on Instructor's portion of CS:APP website.

References

[1] *GCC Online Documentation*. Available at `http://gcc.gnu.org/`.

Index