CS61C Summer 2015 Lab 9 - SIMD Intrinsics and Unrolling

Setup

Copy the directory `~cs61c/labs/09` to an appropriate location in your home directory.

```
cp -r ~cs61c/labs/09  [appropriate directory]
```

Note that all code using SSE instructions is only guaranteed to work on the hive machines.

Many newer processors support SSE intrinsics, so it is certainly possible that your machine will be sufficient, but you may not see accurate speedups. Ideally, you should ssh into one of the hive machines to run this lab.

Exercises

Exercise 1: Familiarize Yourself

Given the large number of available SIMD intrinsics we want you to learn how to find the ones that you'll need in your application.

Intel hosts a variety of tools related to intrinsics, which you can find here (but these are not necessary for this lab).

The one that we're particularly interested in is the Intel Intrinsics Guide. Open this page and once there, click the checkboxes for everything that begins with "SSE" (SSE all the way to SSE4.2).

Do your best to interpret the new syntax and terminology (refer to the lecture slides for how to decode the instruction abbreviations). Find the 128-bit intrinsics for the following SIMD operations (one for each):

- Four floating point divisions in single precision (i.e. `float`)
- Sixteen max operations over signed 8-bit integers (i.e. `char`)
- Arithmetic shift right of eight signed 16-bit integers (i.e. `short`)

Checkoff

- Record these intrinsics in a text file to show your TA.

Exercise 2: Reading SIMD Code

In this exercise you will consider the vectorization of 2-by-2 matrix multiplication in double precision (the code we looked at in the SIMD lecture):

$$\begin{pmatrix} C[0] & C[2] \\ C[1] & C[3] \end{pmatrix} = \begin{pmatrix} C[0] & C[2] \\ C[1] & C[3] \end{pmatrix} + \begin{pmatrix} A[0] & A[2] \\ A[1] & A[3] \end{pmatrix} \begin{pmatrix} B[0] & B[2] \\ B[1] & B[3] \end{pmatrix}$$

This accounts to the following arithmetic operations:

```
C[0] += A[0]*B[0] + A[2]*B[1];
C[1] += A[1]*B[0] + A[3]*B[1];
C[2] += A[0]*B[2] + A[2]*B[3];
C[3] += A[1]*B[2] + A[3]*B[3];
```

You are given the code `sseTest.c` that implements these operations in a SIMD manner. The following intrinsics are used:

`__m128d _mm_loadu_pd( double *p )`	returns vector (p[0], p[1])
`__m128d _mm_load1_pd( double *p )`	returns vector (p[0], p[0])
`__m128d _mm_add_pd( __m128d a, __m128d b )`	returns vector (a_0+b_0, a_1+b_1)
`__m128d _mm_mul_pd( __m128d a, __m128d b )`	returns vector (a_0b_0, a_1b_1)
`void _mm_storeu_pd( double *p, __m128d a )`	stores p[0]=a_0, p[1]=a_1

Compile sseTest.c into x86 assembly by running:

```
make sseTest.s
```

Now, observe the general structure of the code in `sseTest.s`. See if you can find the for-loop in `sseTest.s` (hint: it's a trick question, see exercise 4) and see if you can identify which instructions are performing SIMD operations. Be prepared to describe to your TA what is happening in-general, but you do not need to spend too much time on this section (recall that we are not interested in x86 assembly in this class). We **don't** expect you to tell us exactly what the code is doing line by line. If you're stuck with decipering the x86 assembly, refer to the original `sseTest.c` file that contains the matching C code.

Checkoff

- Explain to your TA what the x86 code is generally doing and note any interesting features you identified.

Exercise 3: Writing SIMD Code

For Exercise 3, you will vectorize/SIMDize the following code to achieve approximately 4x speedup over the naive implementation shown here:

```
static int sum_naive(int n, int *a)
{
   int sum = 0;
   for (int i = 0; i < n; i++)
   {
      sum += a[i];
   }
   return sum;
}
```

You might find the following intrinsics useful:

`__m128i _mm_setzero_si128( )`	returns 128-bit zero vector
`__m128i _mm_loadu_si128( __m128i *p )`	returns 128-bit vector stored at pointer p
`__m128i _mm_add_epi32( __m128i a, __m128i b )`	returns vector (a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)
`void _mm_storeu_si128( __m128i *p, __m128i a )`	stores 128-bit vector a at pointer p

Start with `sum.c`. Use SSE instrinsics to implement the `sum_vectorized()` function.

To compile your code, run the following command:

```
make sum
```

Checkoff

- Show your TA your working code and performance improvement.

Exercise 4: Loop Unrolling

Happily, you can obtain even more performance improvement! Carefully unroll the SIMD vector sum code that you created in the previous exercise. This should get you about a factor of 2 further increase in performance. As an example of loop unrolling, consider the supplied function `sum_unrolled()`:

```
static int sum_unrolled(int n, int *a)
{
   int sum = 0;

   // unrolled loop
   for (int i = 0; i < n / 4 * 4; i += 4)
   {
      sum += a[i+0];
      sum += a[i+1];
      sum += a[i+2];
      sum += a[i+3];
   }

   // tail case
   for (int i = n / 4 * 4; i < n; i++)
   {
      sum += a[i];
   }

   return sum;
}
```

Also, feel free to check out Wikipedia's article on loop unrolling for more information.

Within `sum.c`, **copy your `sum_vectorized()` code into `sum_vectorized_unrolled()` and unroll it four times.**

To compile your code, run the following command:

```
make sum
```

Checkoff:

- Show your TA the unrolled implementation and performance improvement.
- Explain to your TA why the version that has not been manually optimized outperforms some of the manually optimized versions (what does the -O3 flag do?).