"Want to make your computer go really fast? Throw it out a window" – Anonymous

Learning Objectives

Loop vectorization (using SIMD instructions and compiler-based vectorization)

1. Understand how SIMD instructions work by writing SIMD code using compiler instrinsics
2. Analysis of data dependences
3. Use of a compiler to enable vectorization of code (using pragmas and compiler directives)

Work that needs to be handed in

1. First deadline: PrairieLearn Assignment Lab13a
2. Second deadline: PrairieLearn Assignment Lab13b

Guidelines

- There are two types of assessments in this lab. 1) Writing SIMD loop nests by hand. 2) Using pragmas, compiler directives or simple program transformations to enable compiler to vectorize loops.
- You will write SIMD code by hand for loop nests for two functions. The first one, Matrix Multiplication is due by first deadline (part of Lab13a). The second, Mandelbrot, is due by second deadline (part of Lab13b).
- Rather than writing at the assembly level (since we haven't taught x86 assembly), you will use compiler intrinsics. (More on that later)
- Lab13b has 6 questions where you are required to use compiler directives and program transformations to enable compiler to vectorize code. Some of the loops may not be vectorizable. In those case, you have to identify the dependencies that prevent vectorization.
- Lab files are also available through Github. You will need to load `llvm/cs225` module to run it properly on EWS. Detailed instructions are provided later.

Problem Description: Manual vectorization w/ compiler intrinsics

The Intel **Streaming SIMD Extensions (SSE)** comprise a set of extensions to the Intel x86 architecture that are designed to greatly enhance the performance of advanced media and communication applications.

In class, you saw actual Intel SSE *assembly instructions* - however, these are not easy to program with (in general, assembly is not the language of choice for larger programs). Fortunately, some compilers will have built-in intrinsics (which appear as function calls) that provide a one-to-one mapping to SSE assembly.

Example: Inner Product

Recall that the *inner product* of two vectors $\mathbf{x} = (x_1, x_2, ..., x_k)$ and $\mathbf{y} = (y_1, y_2, ..., y_k)$ is defined as follows: $\mathbf{x} \bullet \mathbf{y} = x_1y_1 + x_2y_2 + ... + x_ky_k$. Normally, we could compute the inner product as follows:

```
float x[k];  float y[k];         // operand vectors of length k
float inner_product = 0.0;       // accumulator

for (int i = 0; i < k; i++) {
    inner_product += x[i] * y[i];
}
```

To take advantage of SSE operations, we can rewrite this code using the SSE intrinsics:

```
#include <xmmintrin.h>
// All SSE instructions and __m128 data type are defined in xmmintrin.h file


float x[k];  float y[k];         // operand vectors of length k
float inner_product = 0.0, temp[4];
__m128 acc, X, Y;                // 4x32-bit float registers

acc = _mm_set1_ps(0.0); // set all four words in acc to 0.0
int i = 0;
for (; i < (k - 3); i += 4) {
    X = _mm_loadu_ps(&x[i]); // load groups of four floats
    Y = _mm_loadu_ps(&y[i]);
    acc = _mm_add_ps(acc, _mm_mul_ps(X, Y));
}

_mm_storeu_ps(temp, acc); // add the accumulated values
inner_product = temp[0] + temp[1] + temp[2] + temp[3];

for (; i < k; i++) {             // add up the remaining floats
    inner_product += x[i] * y[i];
}
```

where `__m128` is a 128-bit type for holding 4 floats (e.g. 32-bit single precision floating point numbers).

Problems

1. **Matrix-Vector Multiplication**
   Write a function **mv_multiply** that multiplies a matrix and a vector. Recall that if A is a $k \times k$ matrix, B is a k-vector, and $A * B = C$, then C is a k-vector where $C_i = \Sigma_{j=1}^{k} A_{i,j} * B_j$. Note that:
   - We have provided code without SSE intrinsics. Correctly implementing the code with intrinsics should speed the code up by around a factor of 4 (as our vectors hold 4 floats).
   - You must check whether your optimized code is giving the same result as the unoptimized version. However, since you are dealing with floating point numbers, the two results need not be exactly same. You should check if the two values are within a tolerable gap. See the main function for more info.
   - For this problem, `SIZE` may **not be a multiple of 4**, so you should write your code accordingly. We will be testing this case, so you should too.
2. **Cubic Mandelbrot Set**
   Write a function that determines whether a series of points in a complex plane are inside the Cubic Mandelbrot set. Let $f_c(z) = z^3 + c$. Let $f_c^n(z)$ be the results of composing $f_c(z)$ with itself n times. (So $f_c^n(z) = f_c^{n-1}(f_c(z))$ and $f_c^1(z) = f_c(z)$.) Then, a point (x, y) is considered to be in the Cubic Mandelbrot set, if for a complex number $c = x + yi$, $f_c^n(0)$ does not diverge to infinity as n approaches infinity. Note that:
   - Again, we have provided a non-SSE version of the code; once vectorized the code should run roughly twice as fast.
   - The code makes a simplifying assumption that if $|f_c^{200}(0)| < 2$, then it does not diverge to infinity. The intrinsic `__m128 _mm_cmplt_ps(__m128 a, __m128 b)` might be useful to implement that comparison, combined with some casting.
   - For checking the results, the code generates fractal images from both the scalar and the vector code (`mandelbrot-scalar.bmp` and `mandelbrot-vector.bmp`). The images can be compared to check correctness using the `diff` command.
   - For this problem, you can assume that `SIZE` is a multiple of 4. This way you do not have to worry about remaining iterations.

Useful Intrinsics

```
__m128 _mm_loadu_ps(float *)
__m128 _mm_storeu_ps(float *, __m128)
__m128 _mm_add_ps(__m128, __m128) // parallel arithmetic ops
__m128 _mm_sub_ps(__m128, __m128)
__m128 _mm_mul_ps(__m128, __m128)
__m128 _mm_cmplt_ps(__m128 a, __m128 b)
```

Notes for running on EWS

1. Compile your code with the provided Makefile.
2. For further details about SSE intrinsics, visit the Intel Intrinsics guide:
   `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

Problem Description: Compiler-based Vectorization

In these problems, you are given C code containing simple loops that has been designed to challenge the vectorization capabilities of the compiler. You need to compile the code, read the compiler generated reports, and transform the code appropriately to enable compiler vectorization. The transformation will require the use of pragmas, compiler directives, or simple program transformations. Cache conscious programming approaches like fission, fusion, etc. are acceptable program transformations. Keep in mind that these transformations are not vectorization – rather, they transform the code such that it becomes possible for the compiler to vectorize the code.

For the purposes of this lab, you should limit yourself to the following modifications:

- Loop fusion, fission, and interchange
- Loop unrolling and re-rolling
- Pragmas and directives to give the compiler more information (see below for more detailed explanations)
- Statement reordering

Do not perform any of the following modifications:

- Adding temporaries (i.e. do not allocate additional memory)
- Loop tiling or prefetching, which are optimizations that focus only on caching, not vectorization
- Including extra headers (you don't need them).

Some of the problems are not vectorizable because of dependences In such cases, you **should not** modify the code. Rather, write a comment at the top of the file noting which arrays are involved in the dependence. For example, if arrays `X` and `Y` have a dependence relation which prevents vectorization, put the following on **line 1** of the file, before any includes.

```
// X,Y
```

If array `X` has a dependence relation with itself which prevents vectorization, put the following on **line 1** of the file, before any includes.

```
// X
```

Note that every file we have given needs to be modified in some way. You should either be modifying the code to make the compiler vectorize it, or placing a comment at the top explaining the dependency. Any runtime difference in the given code (before you change anything) is due to measurement error.

Building on EWS

For this lab, we will use the Clang compiler and its auto-vectorizer. For feature consistency, we will require a Clang 3.9.1. The installation we've used up until now has been 3.5, so on EWS you'll need to run the following command to obtain Clang 3.9.1:

```
module load llvm/cs225
```

If you forget to load this module (or are using your own machine without a new enough version), our Makefile will notify you.

To compile the code, use the given Makefile (replace "file" with t1/t2/t3/t4/t5/t6):

```
make file-vector
```

An example command to compile the code in t1.c would be:

```
make t1-vector
```

By default, the compiler is run with optimization level `-O2`, at which Clang will try to vectorize loops when possible. Additionally, we have enabled flags which will print remarks about the compiler's compilation process: which loops were and were not vectorized, the size of the vectors used, as well as hints.

If you want to determine whether vectorization reduced the execution time of your program, you can ask the compile to generate scalar code by using the `-fno-vectorize` flag:

```
make t1-scalar
```

For convenience, we've included a make target which will build and run both for comparison:

```
make t1
```

You can compare the execution time of the scalar and vector executables to determine if vectorization helped to speed up your program and what speedup/slowdown was obtained. When the compiler fails to vectorize the loop, you should obtain similar execution times for both executables, as `file-vector.c` contains scalar code.

Pragmas

Now, we describe a set of `#pragma` statements and compiler directives that can be used with the Clang compiler to enable and control vectorization. Other compilers have similar `#pragma` statements, although the pragma itself differs from compiler to compiler.

The most important pragma is `#pragma clang loop`. This pragma has many options:

- `vectorize(disable)`, `vectorize(enable)`, and `vectorize(assume_safety)` tell Clang to not vectorize, to try to vectorize, and to vectorize without checking dependencies, respectively. You will not need to disable vectorization, however, you may find it useful to instruct Clang to try to vectorize a loop, even if its cost model believes it isn't beneficial. Additionally, if Clang believes there are dependencies which prevent vectorization, but you know that the code is vectorizable, you can tell it to assume the code to be safe and vectorize anyway.
- `vectorize_width(...)` tells Clang which vector size to use when vectorizing a loop. By default, it will pick its own vector size, but you may choose a different vector size altogether. Using this option implies vectorization is enabled, so there is no need to use `vectorize(enable)`. For example, for a vector size of 8, specify `vectorize_width(8)`.
- `interleave(disable)` and `interleave(enable)` control Clang's ability to interleave loop iterations; that is, combine multiple iterations into a single iteration of vectorized code.
- `interleave_count(...)` controls the number of iterations that Clang will attempt to interleave. This syntax is similar to `vectorize_width`, accepting some number.
- `distribute(disable)` and `distribute(enable)` control Clang's ability to split loops apart (distribute work) in order to enable vectorization. This is essentially loop fission, but done by the compiler. Enabling this feature may allow Clang to vectorize certain loops, however you may need to do the fission by hand if Clang is unable.

Here is an example of a pragma that enables a few of these options:

```
#pragma clang loop vectorize_width(4) interleave_count(2) distribute(enable)
for ( ... ) {
    ...
}
```

For more information about these options, see: `https://llvm.org/docs/Vectorizers.html`

Restricting pointers

When optimizing code, a compiler may believe code is vectorizable, but choose not to do so because the vectorized code assumes that the data it's operating on does not overlap in memory (aliasing), which the compiler may not be able to prove for all uses.

When the programmer knows that the pointers do not alias, it is possible to use the `restrict` keyword, which instructs the compiler to assume that the memory does not alias, and optimize using that assumption. For example, if we wanted to tell the compile that the `c` array does not alias any other data, we can write:

```
void f1(float *restrict c, float **b, float **a) {
    for (int i = 0; i < n; i++ {
        for (int j = 0; j < n; j++) {
            c[i] += b[i][j] + a[i][j];
        }
    }
}
```

Alternatively, the compiler can add runtime checks to ensure that pointers do not alias using pointer arithmetic and run vectorized code when possible. An analog in C would look like:

```c
if ( ... ) {
    // pointers do not alias: execute vector code
} else {
    // pointers alias: execute scalar code
}
```

However, this adds extra cost, especially in the case of multi-dimensional arrays using pointers (as in the `f1` function) where the compiler may need to do a check at every iteration.

Although this keyword is a part of C, a compiler is free to ignore it and make its own assumptions (e.g. most compilers now ignore the `register` keyword), so it may or may not be beneficial depending on your choice of compiler.

For this lab, none of our data will alias (by design), so we have added the `restrict` keyword for you where needed, and you should not need to add or remove it anywhere in the code.

Other transformations

Apart from using pragmas and compiler directives, in many cases you will need to transform your code to enable compiler vectorization. Vector loads of current processors are designed to load 128 bits of consecutive data. Thus, when the code has non-unit stride accesses, that is, consecutive iterations access elements that are not consecutive in memory locations, the compiler may decide to not vectorize. Even if you ask Clang to vectorize using `vectorize(enable)`, you might find that you obtain performance slowdown rather than speed-up. In some cases, if you have two nested loops, interchanging the loops can result in unit-stride accesses. If the compiler does not apply the interchange you can apply it manually, but you need to verify that the transformation is legal, that is, the transformed code does the same the original code was doing.

There are also transformations that programmers apply to reduce execution time of scalar code that can sometimes *prevent* compiler vectorization. One of these examples is loop unrolling. In most cases, when the loop is unrolled, the compiler needs to re-roll the code to be able to vectorize the code efficiently. If the compiler does not re-roll the code, the programmer can do it (re-rolling a loop basically means to write it in its regular non-unrolled form).

Another situation where you might want to transform your code to obtain unit stride accesses is when accessing an array of structs. In this case, if the code is accessing the same field of different structs in the array, you will get non-unit strides. The programmer can transform the code by using an array for each field. Although the original code can be vectorized by the compiler, the transformed code runs significantly faster.

```c
// Original code
typedef struct{int x, y, z;} point;
point pt[LEN];

for (int i = 0; i < LEN; i++) {
    pt[i].y *= scale;
}
```

```c
// Transformed code
int ptx[LEN], int pty[LEN],
int ptz[LEN];

for (int i = 0; i < LEN; i++) {
    pty[i] *= scale;
}
```

Timing

Finally, notice that we need to measure the execution time of the loops in order to determine whether vectorization was efficient. To reliably measure this time we have instrumented the code in several ways:

1. Before starting the execution of the code we measure the time stamp counter of the processor using the `rdtsc` instruction (accessible through the `__rdtsc()` intrinsic). This counter counts the processor cycles. (See: `https://en.wikipedia.org/wiki/Time_Stamp_Counter`)
   Similarly, we read the time stamp counter at the end. The difference between the two readings measures the execution time of the loop.

2. We add an outermost loop

   ```
   for (int nl = 0; nl < 1000000; nl++) { }
   ```

   to increase the amount of time the loop executes. We do this because the loops we are executing are too small and run very fast and so the measuring technique that we use may not have enough precision. Notice that the compiler is smart and will notice that this loop is useless and will remove it or interchange it with the inner loop. In order to prevent the compiler from doing that, we add an instruction after the inner loop which modifies one element of an arrays inside the inner loop (in some cases we have used a `vectorize(disable)` that you should not remove).

3. To avoid the compiler removing the whole function (the dead code elimination pass could realize that the result of this code is never used) we also sum all the elements of one of the arrays computed by the loop that we are measuring, and we print the result of this sum (which can also be used to verify that any optimizations were correct). Thus, if the loop that we want to measure is:

   ```
   void t1(float *A, float *B) {
       for (int i = 0; i < 1024; i+=2) {
           A[i+1] = A[i] + B[i];
       }
   }
   ```

   We need to write the following code:

   ```
   void t1(float *A, float * B) {
       unsigned long long start_c, end_c, diff_c;
       start_c = __rdtsc();

       for (int nl = 0; nl < 1000000; nl++) {
           for (int i = 0; i < 1024; i+=2) {
               A[i+1] = A[i] + B[i];
           }
           B[0]++;
       }

       end_c = __rdtsc();
       diff_c = end_c - start_c;
       float giga_cycle = diff_c / 1000000000.0;
       float ret = 0;
       for (int i = 0; i < 1024; i++) {
           ret += A[i];
       }
       printf("It took %f giga cycles and the result is: %f", giga_cycle, ret);
   }
   ```