"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 (via SVN)

1. First deadline: `mv-mult.cpp`
2. Second deadline: `mandelbrot.cpp, t1.c, t2.c, t3.c, t4.c, t5.c, t6.c`

Guidelines

- There are two main parts for this lab. In the first part, you will write SIMD code by hand for loop nests in the two functions contained in `mv-mult.cpp` (for the first deadline) and `mandelbrot.cpp` (for the second deadline). Rather than writing at the asssembly level (since we haven't taught x86 assembly), you will use compiler intrinsics.
- In the second part of this lab (for the second deadline), your goal is to compile the provided 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.
- In most of the cases that we provide, you should be able to transform the code to enable compiler vectorization. In a few cases, however, you will not be able to vectorize the code. Your task is to detect whether the compiler is not vectorizing because:
  1. a data dependence that cannot easily be solved is preventing the compiler from vectorizing the code
  2. the compiler thinks that vectorization will produce slower code
  3. the compiler lacks the appropriate information (that the programmer can provide)
  4. the compiler observes data dependences that prevent it from vectorizing the code, and that the programmer can easily solve by simple transformations to the code.
- For the second part of this lab you will be using the INTEL `icc` compiler. Thus, you should use the `EWS Linux machines`.
- Grading of this lab will consider
  - if the code transformations that you applied produce a correct result, and
  - the performance of your code improved.

  Code transformations should produce the correct output independently of the input size, but performance will only be measured for the original problem size that we are giving you.
- Do not remove the `#pragma novector` that is (sometimes) in front of an outer loop that surrounds the loop that we want to vectorize.

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 type for holding 4 floats (e.g. 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 $C_i = \Sigma_{j=1}^{k} A_{i,j} * B_j$.
   We have provided code without SSE intrinsics. Correctly implementing the code with intrinsics should speed the code up by more than a factor of 2.
   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. (The code has the exact tolerance value.)
   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.
   Again, we have provided a non-SSE version of the code in the SVN; 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 following intrinsic `__m128 _mm_cmplt_ps(__m128 a, __m128 b)` might be useful to implement that comparison.
   For checking the results, the code generates fractal images from both the scalar and the vector code. The images can be compared to check correctness.
   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

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 the second part of this lab, 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

Generally, if you find yourself massively modifying the control flow or meaning of the code, you may want to rethink your method.

As noted in the guidelines, 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 at the top of the file:

```
// X,Y
```

If array `X` has a dependence relation with itself which prevents vectorization, put the following at the top of the file:

```
// X
```

Building

For this lab, we will use the Intel C compiler (`icc`). Back in Lab 2, you ran a script that loaded the `icc` compiler, so you should be able to use the `icc` compiler directly on your terminal in your EWS machine. **You must use an EWS Linux machine for this portion of the lab.**

To compile the code, use the given Makefile (replace "file" with t1/t2/others):

```
make file-vector
```

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

```
make t1-vector
```

By default the compiler is run with optimization level `-O2`, which tries to generate vector code when possible. When using the option `-vec-report2` the compiler generates a report with the loops that were and were not vectorized. For the loops that were not vectorized, the compiler will report the specific reason. You can use this compiler flag, with `-vec-report{i}`, where *i* can have take a value between *0 to 5*; the higher the number, the more information will be reported by the compiler: `-vec-report0` does not generate a report, while `-vec-report1` only reports the loops that have been vectorized.

If you want to determine whether vectorization reduced the execution time of your program, you can ask the compiler to generate scalar code by using the `-no-vec` flag:

```
make t1-scalar
```

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

Pragmas and compiler directives

Next we describe a set of `#pragmas` or compiler directives that can be used with the `icc` compiler to enable vectorization. Other compilers have similar `#pragmas`, although the pragma itself differs from compiler to compiler.

1. `#pragma vector always`. Compilers use a cost model to predict when vector code will run faster than scalar code. If the cost model predicts that the vector code will run slower than the scalar code, the compiler will not vectorize. If for a given loop the compiler reports: "loop was not vectorized: vectorization possible but seems inefficient", it is possible to use the `#pragma vector always` right before the loop to override the compier's cost model, and force it to generate vector code for the desired loop. An example is shown below:

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

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

   You should compare the execution time of the code with and without the `#pragma` and determine based on the execution times obtained whether you should use it. You may find code where the cost model of

the compiler predicts that vectorization is inefficient, but it turns out that the vector code runs faster than the scalar one. Of course, in many cases the compiler is right and the vector code is slower or as fast as the scalar code. So, use this `#pragma` only when vectorization reduces the execution time of the program.

2. `#pragma novector` is used to request the compiler to not vectorize. When the scalar code of a loop runs faster than the vector code, the programmer can use this `#pragma` in front of the loop so that the compiler skips the vectorization for this loop, while vectorizing the rest of the loops in the file.

3. `restrict` keyword. When the compiler reports that a loop was not vectorized due to data dependences it is possible that the compiler cannot analyze the data dependences. This occurs when the function containing the loop that was not vectorized uses pointers for which the compiler cannot determine if they alias or not, that is, if the same memory location can be accessed with two or more different pointers.

   When the programmer knows that the pointers do not alias, it is possible to use the `restrict` qualifier. A pointer declaration using this qualifier establishes an association between the pointer and the object pointed by the pointer, making that pointer and expressions based on that pointer, the only ways to directly or indirectly access the value of that object. Be careful when using this keyword, and only use them when you know that the pointers do not alias. Otherwise, you may alter the semantics of the program.

   The function declaration below uses the `restrict` keyword.

```
void f1(float** restrict c, float** b, float** a) {
    for (int j = 1; j < n; j++) {
        c[j] = b[j] + a[j];
    }
}
```

   Notice that when programmer does not use the `restrict` keyword, the compiler can still vectorize the code. But, since the compiler needs to guarantee that it will generate correct code, it needs to add some checks that execute at runtime to determine which code version (scalar or vector) to execute. For the code above, the compiler needs to guarantee that the array `c` (the one being written) does not overlap with `b` or `a`. There is no need to check array `a` overlaps with `b`, as they are only being read. To check if array `a` overlaps with `c`, since arrays are laid out in consecutive memory locations, the code generated by the compiler checks if the last element of `c` is in a memory position smaller than the first element of `a` or viceversa, that is, if the last element of `a` is in a memory position smaller than the first element of `c`. If any of these two comparisons is true, arrays `c` and `a` do not overlap; otherwise, arrays overlap. The same comparison is done between `b` and `c`. An example of the generated code is shown below:

```
if (((c + n * sizeof(float)) < a) || ((a + n*sizeof(float)) > c)) &&
   (((c + n * sizeof(float)) < b) || ((b + n*sizeof(float)) > c))
{
    // pointers do not alias: execute vector code
}
else
{
    // pointers alias: execute scalar code
}
```

   As the code above shows, the number of checks increases with the number of pointers involved in the loop (in the worst case it needs to generate n^2 number of checks). Thus, when the number of pointers is too large, the compiler may decide that the benefit of vectorization may be not be enough to compensate the overhead added by the checks and decide to not generate vector code.

4. `#pragma ivdep`. Even when the pointers have been qualified with the `restrict` keyword, compilers may still report that vectorization failed due to data dependences. Usually what happens is that the compiler fails to determine if it is safe to vectorize, as the compiler's data dependence analysis cannot always produce precise information. In this scenario, when the programmer knows that a given code does not have data dependences that prevent vectorization, the programmer can insert `#pragma ivdep`

in front of the loop. Again, care must be taken when using this `#pragma` as the compiler will generate incorrect results if there were indeed data dependences that made vectorization not legal.

5. `#pragma auto_inline(on|off)`. Most of the functions that we provide use this `pragma` to indicate the compiler that the function should not be inlined. For an explanation of function inlining see:
   `http://en.wikipedia.org/wiki/Inline_function`
   Sometimes, however, whether vectorization is profitable or not (obtains speedups) depends on the parameter values passed to the function. Thus, the compiler needs to specialize the function for the specific parameters in the function call. This can be done through versioning and inlining. To force the compiler to inline a function, use the `#pragma auto_inline on`.

Apart from using `pragmas` and compiler directives, in many cases programmers need to transform their code to enable compiler vectorization. Vector loads of current processors are designed to load 128-bit 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 force vectorization using the `#pragma vector always`, but 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.

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. Both the original and the transformed code are shown below. Although the original code can be vectorized by the compiler, the transformed code runs significantly faster.

Original code

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

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

Transformed code

```
int ptx[LEN], int pty[LEN],
int ptz[LEN];

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

Re-rolling

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).

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. This counter counts the processor cycles.
   `http://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 `#pragma novector` 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. 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);
}
```