18-645/SP07: How to Write Fast Code

Assignment 6

Due Date: 1(a) on Th Mar 6 6:00pm

Due Date: rest on Fr Mar 21 6:00pm

http://www.ece.cmu.edu/~pueschel/teaching/18-645-CMU-spring08/course.html

For this assignment and the followings, unless we explicitly state it, please do not return any raw arrays of numbers. We will always ask you to draw some graph using the numbers. You should always comment graphs briefly.

You can assume arrays length are always divisible by 4.

Homework submission. *Submit the homework as pdf. Name your file '18645-assign6-userid.pdf' where 'userid' is your andrew user id. The .pdf file must include all plots and figures. Do not put the .pdf file in a zip or tar archive - attach it separately. Send it to <schellap+18645-assign6@andrew.cmu.edu>. In addition to the electronic copy, you must also submit a print-out of your pdf to the TAs at PH-B10 or to Carol Patterson at PH-B15.*

Code submission. *The code you write this week is to be put inside the PDF file. Please only include the relevant "compute" part of the program: we are not interested in initialization or timing, so please do* ***not*** *include these or other extraneous lines.*

Timing. *For timing your code, use the rdtsc timer and always time several successive iterations, i.e., do a warm cache measurement.*

1. *(15 pts)* **Getting icc to work**

   (a) **Running Intel C compiler (icc)** The Intel compiler can only be run on x86 machines. The following page lists the x86 cluster machines:
   http://wiki.ece.cmu.edu/index.php/Clusters
   Note that the ECE "color cluster" does **not** include x86 machines. **icc** will work on any of the machines in the HH1107 Undergraduate Cluster (these have Intel Xeon processors). To determine the OS and the architecture of any Unix machine you are on, type in a terminal:

   ```
   uname -m
   ```

   The output should be "i686" or similar. If you see a "sun4u" or similar, you are on a Sun machine.
   To access the intel compiler on any of the x86 ECE cluster machines, use the following command at a Unix prompt:

   ```
   source /afs/ece.cmu.edu/class/ece645/install-root/bin/iccvars.sh
   ```

   (If you use csh, use the .csh file instead of the .sh file). Now, you should be able to run icc like so:

   ```
   icc -o helloWorld.out helloWorld.c
   ./helloWorld.out
   ```

   To make sure your compile environment works properly, you must compile the file that we provide – `scalar-pointwise-multiplication.c` – using the flag "`-DN=128`", and **send a mail to the submission email address confirming that you have this part working, by 6pm, Thursday, March 6.**

   (b) Time the compute function in `scalar-pointwise-multiplication.c` for N successively equal to all four-powers between 4 and 4^{10}. Do not add any optimizations flags to the compiler.
   Time it again with the optimization flag "`-msse2`". The compiler should now output a message saying that loops are vectorized.
   Use the two series of numbers to make a comparative plot (Mflop/s as a function of N). State the vector single-core peak performance of your machine *for this vector multiplication operation*, and briefly explain the performance behavior.

2. *(10 pts)* **Alignment**

   SSE vector operations manipulate chunks of 128 bit = 16-byte of memory at a time. Therefore, SSE operations need to be *aligned* to 16-byte memory boundaries to be efficient. In fact memory operations come in two flavors, aligned and unaligned. This is not only a performance issue: feeding an unaligned memory address to an aligned load would raise an exception that would kill your program. Therefore, in the next questions, you should always replace all your `malloc` by `_mm_malloc` to avoid this problem.

   But what does 16-byte alignment mean? Write a short piece of code (inside your submitted document file) that gives the closest aligned memory location for a given memory pointer `void *p` that is bigger or equal to `p`.

3. *(20 pts)* **Auto-vectorization** The easiest way to vectorize is to rely on the compiler's auto-vectorization capabilities. Some of these capabilities rely on heuristics and it is possible to get different behaviors by providing "hints".

   (a) As we've seen in part 1, icc is able to automatically vectorize simple code. You can get additional information on the automatic vectorizer by adding the flag "`-vec-report3`." When compilers have trouble optimizing, they can do *versioning* which means keeping multiple versions of the same-code inside the object file and firing them under different conditions. Using this knowledge, explain the difference between the vector code you generated earlier and the vector code you obtain by adding the compiler flag "`-fno-alias`."

   (b) Change the body of the `compute` function in `scalar-pointwise-multiplication.c` to :

   ```
   for (i=0;i<N/2;i++)
       Y[i]=X1[i]*X1[2*i];
   ```

   Try compiling this loop with the auto-vectorizer and look at the report. Now add the following pragma above the for loop: "`#pragma vector always`" and compile again. Plot the performance of the two codes for all N successively equal to all four-powers between 4 and 4^{10}. Explain briefly.

4. *(40 pts)* **Intrinsics, basics**

   (a) Compile the `vector-pointwise-multiplication.c` file (using the "`-msse2`" flag). Pointwise multiplication is now implemented using intrinsics which are a special set of functions that are recognized by the compiler. Note that it requires a different header and note the use of the `__m128` type which corresponds to 4-way float. Good documentation on intrinsics can be found on the Intel website ftp://download.intel.com/support/performancetools/c/linux/v9/intref_cls.pdf or Microsoft website http://msdn2.microsoft.com/en-us/library/kcwz153a.aspx.

   Time this version and create a new graph that compares it to the lines from exercise 1. Briefly discuss.

   (b) Using `_mm_add_ps`, implement a new compute function with parameters `float *y`, `float *x1`,`float *x2` and `float *x3` that computes $y = x1 * x2 + x3$.

   (c) Using `_mm_set1_ps`, implement a new compute function with parameters `float *y`, `float *x` and *a constant* `float a` that computes $y = a * x$.

   (d) Using `_mm_set_ps`, implement a new vectorized compute function with parameters `float *y`, `float *x` that is equivalent to the following scalar code:

   ```
   for (i=0;i<N;i++)
      Y[i]=i*X[i];
   ```

   It can be implemented naively using multiple calls to `_mm_set_ps` but this can be easily optimized so that only one call to `_mm_set_ps` is used. Implement both.

   Add a compiler optimized version of the naive code (try coming up with the best pragmas/flags you can find[1]), create a performance graph with all three lines similar to before and discuss briefly.

5. *(40 pts)* **Intrinsics, medium**

[1] Don't go to multi-core options

(a) Implement a pointwise multiplication of a complex vector of length n by a second complex vector of the same length.

The first variant implements the operation using the *split complex* data format, where a complex vector is stored using two arrays (one for all real parts, and one for all imaginary parts).

Your code should be functionally equivalent to the following:

```
// vector a[N] is scaled by vector b[N], split complex format
void compute(float *ar, float *ai, float *br, float *bi)
{
    for (int i=0; i<N; i++) {
        _Complex float ac = (ar[i]+__I__*ai[i]) * (br[i]+__I__*bi[i]);
        ar[i] = creal(ac);
        ai[i] = cimag(ac);
    }
}
```

(b) The second variant implements the operation using the *interleaved complex* data format, where a complex vector is stored using one array in which real and imaginary parts are interleaved, i.e., alternate. To do this you will need shuffles and/or unpacks. Your code should be functionally equivalent to

```
// vector a[N] is scaled by vector b[N], interleaved complex format
void compute(_Complex float *a, _Complex float *b)
{
    for (int i=0; i<N; i++)
        a[i] *= b[i];
}
```

(c) Implement the pointwise multiplication of a real square $N \times N$ matrix by its transpose. You may want to use the macro `_MM_TRANSPOSE4_PS`.

Your code should be functionally equivalent to the following:

```
// square matrix A[N][N] pointwise multiplied with its transpose
void compute(float *A, float *B)
{
    for (int j=0; j<N; j++)
        for (int i=0; i<N; i++)
            B[i*N+j] = A[i*N+j] * A[i+j*N];
}
```

6. *(20 pts)* **Extra credit** Sometimes, the function that would help you is simply not present on the hardware you target and you have to build ways around that. For instance, the pointwise minimum of two 4-way floats does exist but the pointwise minimum of two 4-way 32 bits integer will only come in with SSE4. Can you come up with an efficient replacement for it? You way want to use comparisons and binary operations to do that.

   Your function should have the following signature

   ```
   __m128i _mm_min_epi32( __m128i a,__m128i b );
   ```

   and perform the following:

   ```
   r0 := (a0 < b0) ? a0 : b0
   r1 := (a1 < b1) ? a1 : b1
   r2 := (a2 < b2) ? a2 : b2
   r3 := (a3 < b3) ? a3 : b3
   ```

7. *(5 pts)* **Do not forget this one!** How much time did you spend on the assignment?