263-2300-00: How To Write Fast Numerical Code

Assignment 2: 100 points
Due Date: Fr, March 18th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring16/course.html
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=2125. Before submission, you must enroll in the Moodle course.
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework, meaning submit latest 48 hours after the due time. For example, submitting 1 hour late costs 1 late day. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's or Gagandeep's office. Late homeworks have to be submitted electronically by email to the fastcode mailing list.
- (Plots)
  For plots/benchmarks, **provide (concise) necessary information for the experimental setup (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions**. Follow (at least to a reasonable extent) the small guide to making plots from the lecture.
- (Code)
  When compiling the final code, ensure that you use optimization flags. **Disable SSE/AVX for this exercise when compiling**. Under Visual Studio you will find it under Config / Code Generator / Enable Enhanced Instructions (should be off). With gcc their are several flags: use -mno-abm (check the flag), -fno-tree-vectorize should also do the job.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Short project info (10 pts)* Go to the list of mile stones for the projects. If you have not done that yet, please register your project there. Read through the different points and fill in the first two with the following about your project (be brief):

   Point 1) An exact (as much as possible) but also short, problem specification.

   For example for MMM, it could be like this:

   Our goal is to implement matrix-matrix multiplication specified as follows:

   Input: Two real matrices A, B of compatible size, $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$. We may impose divisibility conditions on n, k, m depending on the actual implementation.
   Output: The matrix product $C = AB \in \mathbb{R}^{n\times m}$.

   Give the name of the algorithm you plan to consider for the problem and a precise reference (e.g., a link to a publication plus the page number) that explains it.

   Point 2) A very short explanation of what kind of code already exists and in which language it is written.

2. *Maximal performance program (25 pts)* Determine the maximal floating point peak performance on your machine (excluding vector instructions). Write a program that achieves a floating point performance as high as possible. The program does not need to compute anything of importance. Here are some guide lines:

(a) For the operations count, count only floating point instructions (additions and multiplications).

(b) Do not use vector instructions.

(c) There should be no obvious way of simplifying the computation (otherwise the compiler may do it). You may want to check the assembly code to check whether all ops are still there.

(d) Make sure you use the result you compute afterwards in the code (but outside the timing bracket), otherwise the compiler may optimize the computation away.

(e) Very briefly explain the design decisions behind your program and report the percentage of peak performance you achieve. Be ambitious (can you get above 90%?, 95%?).

Solution:

The code provided here achieves maximal performance of 1.98 f/c (99%) on a 2.60GHz Intel Core i7-3720QM CPU with 32KB L1 cache, 256KB L2 and 6MB L3 cache.

3. *Optimization Blockers (40 pts)* Code needed

In this exercise, we consider computing the discrete convolution of kernel h (length 30) with vector x to produce vector y (both of length n). Mathematically, it can be written as:

$$y_m = \sum_{i=0}^{30} h_i f(m-i), \quad 0 \leq m < n$$

where $f(m-i) = 0$ if $(m-i) < 0$, otherwise $f(m-i) = x_{m-i}$. We provide the function *slow_filter* in file `comp.c`, that performs this computation very slowly. Your task is to optimize this function (without using vector instructions).

Run `make` to compile the code. For Windows users, we recommend using Cygwin as a developing environment. Edit the Makefile if needed (architecture flags specifying your processor). The generated executable verifies the code and outputs the performance in flops/cycle. Proceed as follows:

(a) Perform scalar replacement and loop unrolling as discussed in the lecture to increase the performance. Identify any other optimization strategies that may be applicable.

(b) For every optimization you perform, create a new function in `comp.c` that has the same signature as *slow_filter* and register it to the timing framework through the *register_function* function in `comp.c`. Let it run and, if it verifies, determine the performance.

(c) When done, rerun all code versions also with optimization flags turned off ($-O0$ in the Makefile).

(d) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of *slow_filter*. Briefly discuss the table.

(e) Submit your `comp.c` to Moodle.

What speedup do you achieve?

Solution:

We performed the following optimizations on *slow_filter*:

- Scalar replacement of y in inner loop
- Unrolling the inner loop to remove unnecessary branching due to *and* clause
- Unrolling the inner loop by factor of 4
- Complete unrolling of inner loop
- Complete scalar replacement of h in inner loop

The optimized code was tested on a 2.60GHz Intel Core i7-3720QM CPU with 32KB L1 cache, 256KB L2 and 6MB L3 cache. The results can be viewed here.

4. *Locality (20 pts)*

Consider the following C code, where the integer array i contains the n values between 0 and $n-1$ in random order.

```
int i[n]; // Assume randomly initialized
double x[n], y[n], h[K];

for (int j = n-1;  j >= 0; j-=K)
  for(int k = 0; k < K; k++)
    y[ j-k ] = x[ i[j-k] ] * h[k];
```

Considering accesses to arrays i, x, y, and h where do you see

(a) Temporal locality?

(b) Spatial locality?

Solution:

(a) Temporal locality: Accesses to array h. At every iteration of variable j every element of h is accessed once. All items of array i, x, and y are touched only once.

(b) Spatial locality: All accesses but those to array x exhibit spatial locality (sequential accesses). Random accesses to x do not favor spatial locality.