263-2300-00: How To Write Fast Numerical Code
Assignment 2: 100 points
Due Date: Thu March 13 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring14/course.html
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  We set up a SVN Directory for everybody in the course. The Url of your SVN Directory is https://svn.inf.ethz.ch/svn/pueschel/students/trunk/s14-fastcode/YOUR.NETZH.LOGIN/ You should see subdirectory for each homework.
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework. Late submissions have to be emailed to fastcode@lists.inf.ethz.ch.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert them to PDF and submit to svn in the top level of the respective homework directory. Call it homework.pdf.
- (Plots)
  For plots/benchmarks, be concise, but provide necessary information (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 (lecture 5).
- (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. *Polynomial Evaluation (25 pts)* Code needed
   The code in `poly.c` contains a function for evaluating a polynomial of degree N at a point $x_0 = 0.2$. N is given as a runtime parameter from terminal.

   (a) Inspect the function `poly` and determine its op count (double additions and multiplications only).

   (b) Assign the value computed in the previous point to the macro `OPCOUNT` in `poly.c`. Compile the code disabling vectorization and determine its runtime and performance for $N = 2^i$, $i = 7, \ldots, 11$. Collect the results in a small table.

   Now implement a function `horner` which uses Horner scheme for evaluating the polynomial. The function should provide exactly the same signature of the function `poly`.

   (c) What would the op count be for the new implementation?

(d) What performance would you expect to obtain using `horner` instead of `poly`? Why? In the discussion provide all the numbers at the base of your assumption (e.g., the latency of addition and multiplication on your CPU).

(e) In `poly.c`, change the value of the macros `FUNC` and `OPCOUNT` to `horner` and the cost from **??**, respectively. Recompile your code (always disabling vectorization) and determine its runtime and performance for $N = 2^i$, $i = \{7...11\}$. Again, collect your results in a table.

(f) Compare the results in the two tables obtain from 2b and 2e: Which code exhibits higher performance? Which lower runtime? Briefly discuss.

(g) Identify key performance limitations in function `horner` and implement an optimized version of the function called `horner2`. Remember to adjust the `FUNC` macro before recompiling the code. **Hint:** A polynomial $p(x)$ of degree N can be expressed as $p(x) = \sum_{i=0}^{k} x^i p_i(x)$ with $k|N$. The latter formulation supports one specific optimization that helps attaining 80% of peak when compiling with icc or gcc on Sandy Bridge (i.e., disabling vectorization, 1.6 flops/cycle).

Report compiler, version, and flags. Submit your code to the SVN.

Solution:

(a)

$$\texttt{OPCOUNT}_{\texttt{poly}} = 1 + \sum_{i=1}^{N}\left(1 + \sum_{j=0}^{i-1} 1\right) = \sum_{i=1}^{N} i + N + 1 = \frac{(N+1)(N+2)}{2}$$

(c) Assume the following straightforward implementation of Horner scheme:

```
void horner(double * a, size_t N, double x, double * r) {
  int i,j;
  double t;
  *r = a[N];
  for(i = N-1; i >= 0; i--)
    *r = a[i] + *r*x;
}
```

For the function above the op count is

$$\texttt{OPCOUNT}_{\texttt{horner}} = \sum_{i=0}^{N-1} 2 = 2N$$

(d) Assume that multiplication and addition have a latency of 5 and 3 cycles respectively. Then due to the loop-carried dependency the expected performance is

$$P = \frac{2N}{(5+3)N} = 0.25 \text{ flops/cycle}$$

(g) The loop-carried dependency is probably the main performance limitation in the code. Taking into account that a polynomial $p(x)$ of degree N can be expressed as $p(x) = \sum_{i=0}^{k} x^i p_i(x)$ with $k|N$, we could reduce the problem of evaluating $p(x)$ to the one of evaluating k independent polynomials $p_i(x)$ using k different accumulators. For example, in our tests we use $N = 2^i$, $i = \{7...11\}$ so choosing $k = 2^i$, $i = \{1...6\}$ always divides N. The code at this link contains three possible implementation with three different values of k.

3. *Optimization Blockers (40 pts)* Code needed
Download, extract and inspect the code. Your task is to optimize the function called superslow (guess why it's called like this?) in the file **comp.c**. The function runs over an $n \times n$ matrix and performs some

computation on each element. In its current implementation, *superslow* involves several optimization blockers. Your task is to optimize the code.

Edit the Makefile if needed (architecture flags specifying your processor). Running `make` and then the generated executable verifies the code and outputs the performance (the flop count is underestimated, since the trigometric functions are ignored) of superslow. Proceed as follows

(a) Identify optimization blockers discussed in the lecture and remove them.

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

(c) In the end, the innermost loop should be free of any procedure calls and operations other than adds and mults.

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

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

(f) Submit your `comp.c` to the SVN

What speedup do you achieve?

Solution: A reasonable transformation which would yield full points can be found here: Example Solution

4. *Locality of a Convolution (20 pts)*

Consider the following C code, which computes a form of 2D convolution. The function takes as input a matrix A of size $n \times n$, and a vector H of size k^2; the result is stored in a matrix B of size $n \times n$.

```
assert(N > K && K > 2);
double A[N][N], B[N][N], H[K*K];
for (int i = 0; i < N-K; i++)
  for(int j = 0; j < N-K; j++)
    for(int k = 0; k < K*K; k++)
      B[i][j] = A[i+1][j+2] * A[i+k/K][j+k%K] * H[K*K-k-1];
for(int i = N-K; i < N; i++)
  for(int j = N-K; j < N; j++)
    for(int k = 0; k <  K*K; k++)
      B[i][j] = A[i-1][j-2] * A[i-k/K][j-k%K] * H[K*K-k-1];
```

Inspecting the data accesses, where do you see

(a) Temporal locality?

(b) Spatial locality?

Solution:

(a)
- Matrix `A` exhibits temporal locality. In consecutive iterations of the outer i and j loops, there are overlapping blocks of `A` that will be accessed in the inner-most k loop, as well as the values of `A[i+1][j+2]` and `A[i-1][j-2]`. It is debatable whether temporal locality can also be observed on the overlapping values of `A` that are accessed in the two i loops (sub-matrix `A[N-2*K][N-2*K]` to `A[N-2][N-2]`).
- Matrix `B` exhibits clear temporal locality - `B[i][j]` is overwritten in each k-loop.
- It is also debatable whether `H` exhibits temporal locality, as it depends on the value of K. All values of `H` are accessed $(N-K)^2 + K^2$ times, taking both i loops into consideration.

(b)
- Matrices `A` and `B` and array `H` exhibit spatial locality, their values are accessed in a stride of 1 in both inner-most k loops.
- It is not clear whether `A[i+i][j+2]` in the first inner-most loop and `A[i-1][j-2]` in the second inner-most loop can be considered as a spatial locality in relation to `A[i+k/K][j+k%K]` and `A[i-k/K][j-k%K]` respectively, as it depends on the value of K.