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

Assignment 5: 100 points
Due Date: Thu April 3, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring14/course.html
Questions: fastcode@lists.inf.ethz.ch

General 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 sub-directory 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.

Submission instructions for this homework (read carefully):

Follow the instructions below to create, name, and submit your source code. For all questions where you are asked to provide C code, we provide a corresponding C template file that you need to use to answer the question. In particular,

- do NOT change the signature of the functions
- do NOT change the type of global arguments
- comply with the given environment variables, do not add others
- do not cross-reference functions among your submitted files. Each .c file should be completely independent should be compilable on its own, (when compiled with our own main.c that you don't have access to).
- clean up your code as much as possible, do not leave in debug statements
- we provide a helpful sample main.c to show you how we would like the code to be structured. You are free to use it or not use it. It doesn't come with a timer, as you need to implement one. (possibly using the one from Assignment 2). Do not submit this sample file.

Verifying your code: All code that you produce as a part of this assignment (and future assignments too!) needs to be verified for computational correctness. For MMM, the easiest way is to compare to the standard triple loop for a few randomly selected input matrices. We will independently verify your code for correctness. Incorrect programs will not receive any credit.

Submitting your code: Your C files corresponding to the questions should be named code0.c, code1.c, code2.c, and code3.c. In all, you will submit the 4 code<n>.c files, and the finalcode.c file. Do NOT change file names! Do not zip or otherwise archive the files.

Exercises:

1. *(Mini-MMM 65 pts)* Code needed
   The goal of this exercise is to implement a high performance mini-MMM (similar to how ATLAS optimizes it) to multiply two square $N_B \times N_B$ matrices (N_B is a parameter), which is then used within MMM in problem 2.

   (a) (By definition) Implement the code that implements MMM directly based on its definition (triple loop implementation), using the ijk loop order. Call this **code0**. We view this code as implementing a mini-MMM.

   (b) (Register blocking) Block into micro-MMMs with $M_U = N_U = 2,\ K_U = 1$. The inner triple loop must have the kij order, as explained in class. Manually unroll the innermost i- and j-loop and perform scalar replacement on this unrolled code and write SSA code. Assume that 2 divides N_B. Call this **code1**.

   (c) (Unrolling) Unroll the innermost k-loop by a factor of 2 ($K_U = 2$, which doubles the loop body) and again do scalar replacement (SSA code). Note that the $M_U \times N_U$ block of the resulting matrix is loaded only once outside the innermost k-loop (as explained in class). Assume that 2 divides N_B. This part gives you **code2**.

   (d) (Alternative micro-MMM, following the x86 extended model from class) Now block **code0** for mini-MMM into micro-MMMs with $M_U = 1,\ N_U = 8,\ K_U = 2$. Again, unroll the innermost k,i,j loops and do scalar replacement with SSA. Assume that 8 divides N_B. This part gives you **code3**.

   (e) (Best block size N_B) Determine the L1 data cache size C_1 in doubles and its block size B_1, also in doubles. Use the model (inequality) from class (section 2e in the MMM optimization notes) to determine the best (largest) block size N_B for each: **code1**, **code2**, **code3**. Run these three for this block size and report the performance obtained (three numbers) in flops/cycle. Which one is best?

   (f) (Blocking for L2 cache) Now go through the same steps as in the previous part, but this time considering your L2 cache. Measure and report the three performance numbers.

   Your best mini-MMM is the code plus block size that achieved the highest performance among the six in parts 1e and 1f

2. *MMM (15 pts)* Implement an MMM for multiplying two square $n \times n$ matrices assuming N_B divides n, blocked into $N_B \times N_B$ blocks using your best mini-MMM code from exercise 1. This is your **finalcode**. Create a performance plot comparing this code and **code0** (by definition) above for sizes roughly in the range $n = 100, \ldots, 1500$ in steps of roughly 100 (the exact numbers will depend on the N_B you found since you want multiples of N_B). The x-axis shows n; the y-axis performance in flops/cycle. Briefly discuss the plot.

3. *Roofline (15 pts)* Assume the following hardware parameters for an Intel Haswell CPU:

   - Can issue one scalar add and two scalar multiplications per cycle.
   - CPU frequency is 3.2 GHz.
   - Last level cache (LLC) size is 6 MB and cache block size is 64 bytes.
   - Maximal bandwidth is 25.6 Gbyte/sec.

   Draw a roofline plot for double precision floating point operations on the given hardware. The units for x-axis and y-axis are flops/byte and flops/cycle, respectively. Specifically, the plot should contain 2 lines:

   (a) Upper bound based on peak performance π.

   (b) Upper bound based on the maximal memory bandwidth β.

Provide enough detail (labels etc.) so we can check correctness.

Now consider running the following code for the computation of $C = AB + C$ on the platform above (all matrices have size $N \times N$):

```
void mmm(double * A, double * B, double * C, size_t N) {
  int i,j,k;
  for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
      for(k = 0; k < N; k++)
        C[i][j] += A[i][k]*B[k][j];
}
```

(c) Is it possible to reach peak performance? If not, include a tighter performance bound specific for `mmm` in your roofline plot.

Finally consider the following code:

```
double fjump(double * x, size_t N) {
  int i, j;
  double res = 1.;
  for(i = 0; i < 16; i++)
    for(j = i; j < N; j += 16)
      res *= x[j]+x[j+1];
  }
  return res;
}
```

(d) Can the execution of `fjump(x, 3*1024*1024)` reach peak performance? If not determine a tighter performance bound.

Solution:

(a) The CPU taken into account can issue two mults and an add per cycle, so $\pi = 3$ flops/cycle.

(b) The CPU can transfer data at a maximal bandwidth $B = 25.6$ GB/s. Using the CPU frequency ($f = 3.2$ GHz) we can compute $\beta = \frac{B}{f} = 8$ bytes/cycle.

(c) The answer is no. The CPU peak performance can be approached only when a computation presents a 2:1 balance of multiplications and additions. The cost for `mmm` can be expressed as $C(N) = (\text{Mult}(N), \text{Add}(N)) = (N^3, N^3)$. The bound on the runtime is given by $r = r_{\text{mixed}} + r_{\text{adds}}$, where

$$r_{\text{mixed}} = \left(N^3 + \frac{N^3}{2}\right) \cdot \frac{1}{3} = \frac{N^3}{2} \text{ cycles}$$

are spent computing using the full execution unit (i.e., two multiplications + one addition issued at every cycle), and

$$r_{\text{adds}} = \frac{N^3}{2} \text{ cycles}$$

are spent computing only additions. The bound $r = N^3$ cycles leads to the following bound on performance for `mmm`:

$$\pi_{\text{mmm}} = \frac{2N^3}{N^3} = 2 \text{ flops/cycles .}$$

(d) At every i-iteration 12 MB (i.e., twice the LLC size) are transferred to the CPU. This implies that every access to x[j] is a miss and that for every j-iteration a whole cache block must be transferred to the CPU. The operational intensity of `frump` is thus

$$I_{\text{fjump}} = \frac{2}{64} = \frac{1}{32} \text{ flop/byte} .$$

Finally, we can determine a tighter bound for the function's performance:

$$\pi_{\text{fjump}} = \beta I_{\text{fjump}} = \frac{1}{4} \text{ flop/cycle} .$$

The answers above are summarized in the roofline plot sketched in Fig 1.

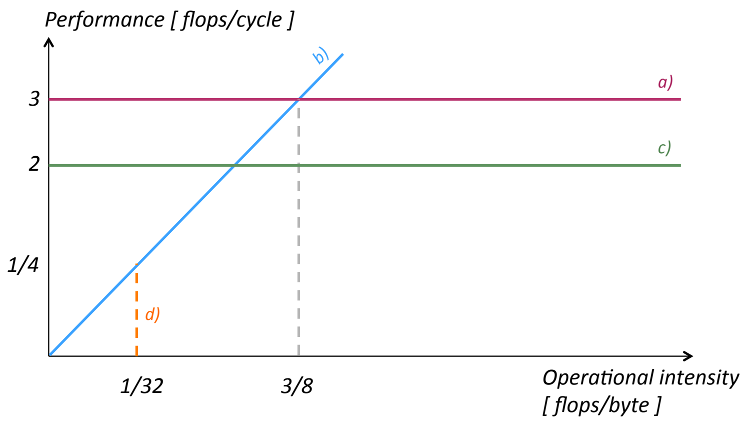

Figure 1: Sketch of the roofline plot for Ex. 3.