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

Assignment 4: 120 points
Due Date: Th, April 13th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring17/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=3122. 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, Gagandeep's or Georg'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.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. Cache mechanics for scalar code (40 pts)
   Consider a direct mapped cache of size 16KB with a block size of 16 bytes. The cache is write-back and write-allocate. Remember that `sizeof(float) == 4` and `sizeof(size_t) = 8`. Assume that the cache starts empty and that access to local variables i.e., `i`, `j` and `n` do not cause cache misses.

   (a) The code below computes the sum of each row in matrix `mat` into vector `v`. Assume that `mat` and `v` are stored consecutively in memory and that `mat` starts at address 0.

   ```c
   void sum_row(float * v, float ** mat, const size_t n) {
       for (size_t i = 0; i < n; i++)
           for (size_t j = 0; j < n; j++)
               v[i] = v[i] + mat[i][j];
   }
   ```

   i. What is the cache miss rate if $n = 64$?
   ii. What is the cache miss rate if $n = 128$?

   (b) Next consider the code below which computes the sum of each column in matrix `mat` into vector `v` with the same assumptions as for (a).

   ```c
   void sum_column(float * v, float ** mat, const size_t n) {
       for (size_t i = 0; i < n; i++)
           for (size_t j = 0; j < n; j++)
               v[i] = v[i] + mat[j][i];
   }
   ```

   i. What is the cache miss rate if $n = 64$?
   ii. What is the cache miss rate if $n = 128$?

   Show in each case some detail so we see how you got the result.

2. (25 pts) Cache Mechanics for vector code
The AVX vectorized code below computes the outer product of vectors `u` and `v` into matrix `w`. Assume the same cache and setup as for previous problem. Assume that `w`, `u` and `v` are allocated consecutively (in the order `w`, `u`, `v`) and that `w` starts at address 0.

```
void outer_product(float * w, float *u, float * v, const size_t n) {
    for (size_t i = 0; i < n; i++){
        __m256 op1 = _mm256_set1_ps(u[i]);
        for (size_t j = 0; j < n; j += 8){
             __m256 op2 = _mm256_loadu_ps(v + j);
             __m256 res = _mm256_mul_ps(op1, op2);
             _mm256_storeu_ps(w + n * i + j, res);
        }
    }
}
```

(a) What is the cache miss rate if $n = 64$?

(b) What is the cache miss rate if $n = 128$?

(c) Does it make a difference in both cases if `w`, `u` and `v` are allocated consecutively but in another order? If yes, then report with an example.

Show in each case some detail so we see how you got the result.

3. (50 pts) Roofline
We consider a Intel Core i7 4700K processor, as such it has the following specifications

- The processor is a Haswell micro architecture
- Memory bandwidth is 13 GB/s.
- Cache with 64-byte cache line size.
- CPU frequency is 3.5 GHz.

(a) 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. Draw one roof each for the following scenarios

- using FMAs
- using vectorized FMA's

(b) Consider the execution of the following three kernels on the platform above (vector x and y have size N, matrices A, B and C have size $N \times N$). Assume a cold cache scenario and sizes for N such that the whole working set of each kernel fits into cache once its loaded. Both vectors and the matrices are cache-aligned (first element goes into first cache block). The kernels are executed separately (different processes and no interaction between them). Assume that variables `i`, `j`, `k` and `N` are stored in registers, and ignore the integer operations. Assume that no optimizations that change operational intensity are performed (the code stays as is) and assume that only `kernel1` and `kernel2` can be vectorized, but not `kernel3`:

```
// you can assume vectorization
void kernel1(double *x, double *y, const int N) {
 for (int i = 0; i < N; ++i)
  x[i] = 1.1 * y[i] + 3.1;
}
```

```
// A,B and C are N*N matrices
// you can assume vectorization
void kernel2(double * A, double * B, double * C, const int N) {
 for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
   for (int k = 0; k < N; k++)
    C[i * N + j] += A[i * N + k] * B[j * N + k];
}
```

```c
// A,B and C are N*N matrices, t is a N sized vector
// assume the compiler can not vectorize this!
void kernel3(double * A, double * B, double * C, double * t, const int N) {
 for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
   for (int k = 0; k < N; k++) {
    t[N - k - 1] += A[i * N + k] * B[j * N + k];
     C[i * N + j] += t[(k + 1) % N] * 1.1;
 }
}
```

For **each** of the kernels,

i. Derive the operational intensity (counting reads and writes - keep in mind that we are in a write-allocate scenario) and locate it in your roofline plot.

ii. For each of the kernels derive the maximal theoretical achievable performance on this platform given their operational intensity. (Assume that `kernel3` is not vectorized!)

(c) For each of the three kernels, assuming that we change the data type from double to float

- what are the new operational intensities?
- what are the new theoretical bounds on the performance? (Hint: Think about the change for vectorized code)