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

Assignment 3: 100 points
Due Date: Mon, April 11th, 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.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. (25 pts) Cache Simulator
   For this exercise, we consider the trace driven cache simulator Dinero IV, and the skeleton code available here. To run the simulator, we need to compile Dinero IV first. Windows users can only use Cygwin to run the code, since the software cache simulator is based on GNU dependencies and build tools.

   To compile the cache simulator run:

   ```
   cd d4
   ./configure
   make
   # ... d4/dineroIV is built
   ```

   To run the cache simulator, execute:

   ```
   cd ..
   make            # make the executable
   make trace      # create a memory trace
   make simulation # simulate a memory trace
   ```

   Let's assume one cache of size 32KB, 8-way associative, with 64B of block size, and LRU replacement. Given a a baseline matrix transposition algorithm, do the following:

   (a) Modify the loop structure of the `transpose` function in `main.c` to reduce the number of cache misses for the given size 4096×4096. For this use optimization ideas learned in class and consider variants. Do not use unrolling or scalar replacement. Do not modify `trace.c` nor `mat.c` files, and use `mat_get` and `mat_set` to read / write to the matrices (those two routines will generate the memory trace used for Dinero IV).

   (b) Test your algorithm for square matrices of size $n \times n$, where $n = 2^k$ and $k \in \{8, 9, \ldots, 12\}$, and report the results.

2. Cache mechanics (25 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`. Assume that the cache starts empty and

that local variables and computations take place completely within the registers and do not spill onto the stack.

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

(a) Consider the supposedly cache-unfriendly matrix computation below. Assume that the `mat` matrix starts at address 0 (*min* computes minimum of two floats).

```
void hostile_to_cache(float ** mat, int v, int n) {
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            mat[i][v] = min(mat[i][v], mat[i][k] + mat[k][v]);
}
```

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

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

(b) Next consider the supposedly cache-friendly implementation of the same computation. Assume that `tmp` array is of size n and starts just after `mat`.

```
void cache_friendly(float ** mat, float * tmp, int v, int n) {
    for (int i = 0; i < n; i++){
        tmp[i] = mat[i][v];
    }
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            tmp[i] = min(tmp[i], mat[i][k] + mat[k][v]);
    for (int i = 0; i < n; i++){
        mat[i][v] = tmp[i];
    }
}
```

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

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

(c) Does the cache-friendly code result in fewer cache misses?

3. (20 pts) Cache Mechanics
In this problem, you will compare the performance of direct mapped and 4-way associative caches for the initialization of 2-dimensional arrays of data structures. Both caches have a size of 1024 bytes. The direct mapped cache has 64-byte blocks while the 4-way associative cache has 32-byte blocks.

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

You are given the following definition of a coffee particle:

```
typedef struct{
    int color[3];
     float position[3];
     int aroma;
     int taste;
} particle_t;
particle_t coffee[16][16];
register int i, j, k;
```

Also assume that

- `sizeof(int) = 4` and `sizeof(float) = 4`
- `coffee` begins at memory address 0
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i`, `j`, `k` are stored in registers and any access to these variables does not cause a cache miss.

(a)
```
for (i = 0; i < 16; i ++) {
    for (j = 0; j < 16; j ++) {
        for(k = 0; k < 3; k ++) {
            coffee[i][j].color[k] = 0;
        }
        for(k = 0; k < 3; k ++) {
            coffee[i][j].position[k] = 0.0;
        }
        coffee[i][j].aroma = 0;
        coffee[i][j].taste = 0;
    }
}
```

i. What fraction of the writes in the above code will result in a miss in the direct mapped cache?
ii. What fraction of the writes will result in a miss in the 4-way associative cache?

(b)
```
for (i = 0; i < 16; i ++) {
    for (j = 0; j < 16; j ++) {
        for (k = 0; k < 3; k ++) {
            coffee[j][i].color[k] = 0;
            coffee[j][i].position[k] = 0.0;
        }
        coffee[j][i].aroma = 0;
        coffee[j][i].taste = 0;
    }
}
```

i. What fraction of the writes in the above code will result in a miss in the direct mapped cache?
ii. What fraction of the writes will result in a miss in the 4-way associative cache?

(c)
```
for (i = 0; i < 16; i ++) {
    for (j = 0; j < 16; j ++) {
        for (k = 0; k < 3; k ++) {
            coffee[i][j].color[k] = 0;
            coffee[i][j].position[k] = 0.0;
        }
        coffee[i][j].aroma = 0;
        coffee[i][j].taste = 0;
    }
}
```

i. What fraction of the writes in the above code will result in a miss in the direct mapped cache?
ii. What fraction of the writes will result in a miss in the 4-way associative cache?

4. (25 pts) Roofline
Consider a processor with the following hardware parameters:

- Can issue two scalar floating point additions and one scalar floating point multiplication per cycle.
- Memory bandwidth is 26 GB/s.
- One cache with 64-byte cache block size.
- CPU frequency is 3.25 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.

(b) Consider the execution of the following three kernels on the platform above (vector x has size N and vector y has size $N+1$). Assume a cold cache scenario and a vector size N such that both vectors fit whithin the cache once they are loaded from memory. Both vectors are cache-aligned (first element goes into first cache block).

```
void kernel1 (double *x, double *y, size_t N) {
    int i;
    for (i = 0; i < N; ++i)
        x[i] += 1.7 * y[i] + y[i+1];
}
```

```
void kernel2 (double *x, double *y, size_t N) {
    int i;
    for (i = 0; i < N; ++i)
        x[i] += 1.7 + y[i] + y[i+1];
}
```

```
void kernel3 (double *x, double *y, size_t N) {
    int i;
    for (i = 0; i < N; ++i)
        x[i] *= 1.7 * y[i] * y[i+1];
}
```

For **each** of the kernels,

i. Derive the operational intensity (counting reads and writes) and locate it in your roofline plot.

ii. Take the instruction mix into account to derive a (now kernel-specific) tighter bound for peak performance. Include these in the roofline plot and label by kernel number.

iii. Is it is possible to reach the peak performance from ii. ? If not, include the tighter performance bound in your roofline plot and label it.

(c) For each of the three kernels, consider the following modification in the memory access pattern (strided access). We only show `kernel1`, but assume the same modification in `kernel2` and `kernel3`. The vectors have an according larger size.

```
void kernel1 (double *x, double *y, size_t N) {
    int i;
    for (i = 0; i < N; ++i)
        x[i] += 1.7 * y[8*i] + y[8*i+2];
}
```

Answer all the questions in part (b) for the modified kernels using a new roofline plot for readability.