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 $k \in \{8, 9, \dots, 12\}$, and report the results.

   Solution: The initial run of the code with DineroIV shows us that we have a 0.125 miss ratio for reads, when reading matrix `m1`, but also miss ratio of 1 when writing to `m2`. Therefore, we need to block the matrix transposition, such that we improve the miss ratio for write accesses. The simplest form of blocking would be modifying the 2-fold loop code into a 4-fold loop code:

```
int b = 8;
for (int i = 0; i < m; i += b)
    for (int j = 0; j < n; j += b)
        for (int k = i; k < i + b; ++k)
            for (int l = j; l < j + b; ++l)
                mat_set(m2, l, k, mat_get(m1, k, l));
```

The code above allows us to change the size of the block and test different versions. Figure 1 clearly shows that block of size 8 is the best strategy for the given cache parameters.

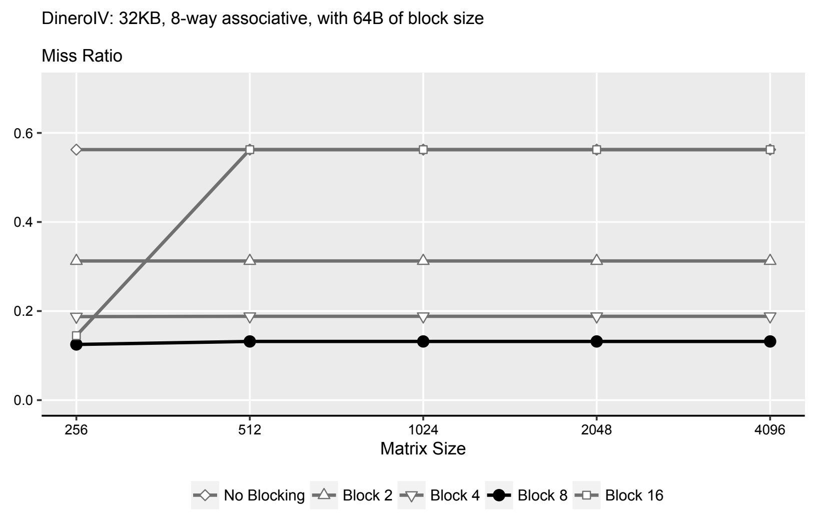


Figure 1: Miss ratio for both read and write accesses for different block sizes

The blocking will significantly reduce the miss ratio of the write accesses as shown in Table 1. Ideally, with the prefect block size, we would expect that read miss ratio will be identical to the write miss ratio, but this is not the case, due to conflict misses. The number of conflict misses will be OS dependent, due to different implementation of the `malloc` function, used internally by `mat_alloc`.

Size	No blocking	Block 2	Block 4	Block 8	Block 16
256	0.125 / 1.0	0.125 / 0.5	0.1250 / 0.25	0.125 / 0.1250	0.125 / 0.1646
512	0.125 / 1.0	0.125 / 0.5	0.1265 / 0.25	0.125 / 0.1387	0.125 / 1.0000
1024	0.125 / 1.0	0.125 / 0.5	0.1265 / 0.25	0.125 / 0.1387	0.125 / 1.0000
2048	0.125 / 1.0	0.125 / 0.5	0.1265 / 0.25	0.125 / 0.1387	0.125 / 1.0000
4096	0.125 / 1.0	0.125 / 0.5	0.1265 / 0.25	0.125 / 0.1387	0.125 / 1.0000

Table 1: Miss ratio for reads / writes for different block sizes (Mac OS X Yosemite 10.10.5)

While the analysis so far will get you the full points, for the enthusiasts, we have to note that the miss ratio can be decreased even further. One way to do this is by using a temporary storage, where we first copy the block that we read, and then we write it to the output matrix. An implementation of this variant is available here. Results are depicted on Table 2.

Size	256	512	1024	2048	4096
R/W Miss Ratio	0.0625 / 0.0626	0.0626 / 0.0626	0.0626 / 0.0626	0.0626 / 0.0626	0.0625 / 0.0626

Table 2: Blocking with size 8 and temporary storage for transposition (Mac OS X Yosemite 10.10.5)

Note that this implementation will only improve the miss ratio, while decrease performance, since it doubles the number of reads and writes. A more realistic scenario would be blocking for registers, loop unrolling etc, which we omit for simplicity.

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?

Solution:

(a) *mat* is accessed $4n^2$ times.

i. *mat* fits in cache, only compulsory misses, thus miss rate $= \frac{1024}{4*64*64} = 6.25\%$.

ii. *mat* does not fit in cache, the entries are accessed column-wise as *mat*[i][v] and *mat*[i][k] inside the inner i-loop. If *mat*[i][v] and *mat*[i][k] lie on the same cache line then there is only one cache miss (which happens for 4 iterations of outer k-loop and can be ignored), otherwise there are two cache miss for each column-wise access. The column-wise access *mat*[k][v] also results in relatively smaller number of cache misses so it can be ignored. Thus the cache miss rate $\approx \frac{2*128*128}{4*128*128} = 50\%$.

(b) *mat* and *tmp* are both accessed $2n^2 + 2n$ times.

i. For the first loop, there are 80 compulsory misses. For the second loop, there are 960 compulsory misses for loading *tmp*, plus there are conflict misses whenever *tmp*[i] maps to the same cache line as *mat*[i][k] (ignoring single conflict with *mat*[k][v]). The number of times *tmp* conflicts with *mat*[i][k] is 64. Each conflicts result in two misses, thus total number of cache misses in double loop are 1088. The cache misses for the third loop are small and can be ignored. Thus cache miss rate $\approx \frac{1168}{4*64*64+4*64} = 7.1\%$.

ii. For the second loop, each access to *mat*[i][k] results in a cache miss. There are also cache misses due to entries in *tmp* and *mat* mapping to the same cache line, which being small can be neglected. The cache misses for the first and third loop can also be neglected. Thus, cache miss rate $\approx \frac{128*128}{4*128*128+4*128} \approx 25\%$.

(c) When the matrix *tmp* does not fit into cache, the cache-friendly reduces miss rate.

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?

Solution:

(a) There are 8 writes for each iteration of j-loop. Each cache line in direct mapped can hold two *particle_t* objects whereas an associative cache line can hold one such object.

  i. There are cache misses for every alternate *particle_t* object. Thus, fraction of write that are misses = 0.0625.

  ii. There are cache misses for writing every *particle_t* object. Thus, fraction of write that are misses = 0.125.

(b) i. The *coffee* array is accessed column-wise which results in cache miss every time an array entry is written. Thus, fraction of write that are misses = 0.125.

  ii. There are cache misses for writing every *particle_t* object. Thus, fraction of write that are misses = 0.125.

(c) Same as part (a).

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.

Solution

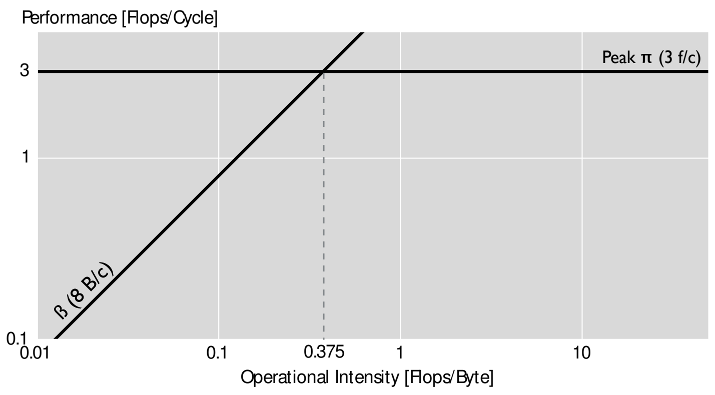


Figure 2: Roofline plot for part 4a

(a) The given CPU performs two additions and one multiplication per cycle, hence peak performance is $\pi = 3$ flops/cycle. With respect to memory bandwidth, it can transfer 26 Gbytes/sec and has a frequency of 3.25 GHz, thus peak bandwidth is $\beta = 26 \times 10^9/(3.25 \times 10^9) = 8$ bytes/cycle. The ridge point of the roofline plot is at 0.375 flops/byte. See Figure 2.

(b) `kernel1`

i The function loads vector `x[N]` and `y[N+1]` only once. The access patterns of `x` and `y` exhibit spatial locality since all elements are loaded in one cache line and are always used. Assuming that the vectors are cache-block aligned, approximately[1] $2N$ doubles are loaded from memory, while N doubles are written to memory. A total of $3N$ flops are performed. Therefore, the operational intensity is:

$$I \approx 3N/(16+8)N = 0.125 \ flops/byte.$$

ii `kernel1` has perfect balance of additions and multiplications for the given platform, 2:1 (adds:mults). Thus, the peak performance of the platform is a tight bound for the peak performance of the kernel.

[1]this is approximate because depending on the relation between N and the cache block size, and due to the fact that y has an extra element, one or two of the cache blocks loaded won't be fully utilized. Both solutions, approximate and exact under certain assumptions, will be considered correct

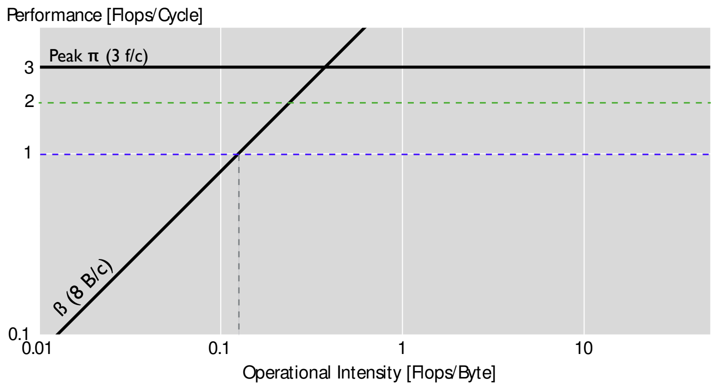


Figure 3: Roofline plot for part 4b

iii The kernel, however, does not reach peak performance because it is memory bound.

`kernel2`

i The operational intensity is the same as `kernel1` because the only difference is the instruction mix, the instruction count remains the same.

ii The maximum performance is 2 flops/cycle, due to the instruction mix (additions only).

iii This kernel does not reach the peak performance either.

`kernel3`:

i Again, the operational intensity is the same as for the previous kernels.

ii The maximum performance is 1 flop/cycle (multiplications only)

iii This kernel reaches the tight peak performance bound of 1 flop/cycle.

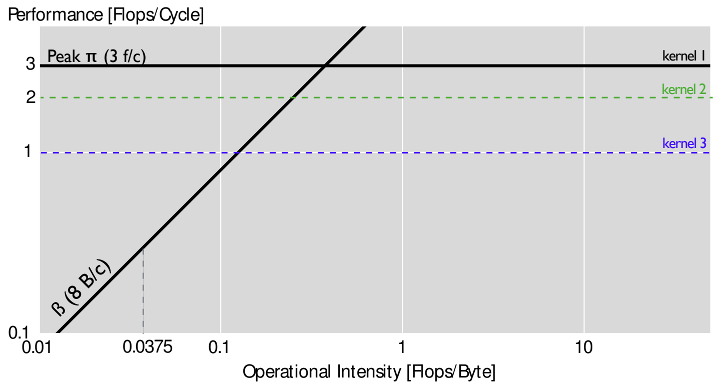
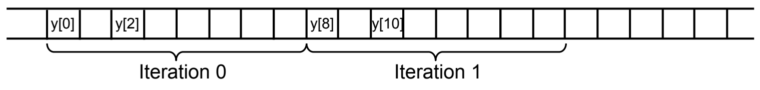


Figure 4: Roofline plot for part 4c

(c) In this case, y is accessed as a stride. As a result, only two elements are used for every cache block loaded from y. Thus, in order to use $N + 1$ elements, approximately $8N$ elements have to be loaded from memory. This results in a lower operational intensity for all the three kernels:

$$I \approx 3N/(16 + 8 \times 8)N = 0.0375 \ flops/byte.$$



As depicted in Figure 4, the hardened performance bounds are equivalent to part 4b. None of the kernels, however, will reach its corresponding peak performance bound.