263-2300-00: How To Write Fast Numerical Code
Assignment 3: 100 points
Due Date: Th, March 19th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/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=1317. Before submission, you must enroll in the Moodle course. Enrollment key is "263-2300".
- (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. 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 Daniele's office. Late homeworks have to be submitted electronically.
- (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 (soon in lecture).
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Cache mechanics (12 pts)* Consider an 8-way, 32KB cache with a cache block size of 64 bytes. Assume 64 GB of byte-addressable RAM, on a 64-bit machine.

   (a) Determine the number of cache sets.

   (b) Determine the length of the cache tag for each cache line. How many of those bits will be effectively used on this particular machine ?

   (c) For a given `char * p` with address `0xDE147BA` (in hexadecimal format) calculate the cache tag value, set index, and block offset. Express the obtained values in hexadecimal format.

   Solution:

   (a) cache size $= S \cdot E \cdot B$, thus $S = \frac{32 \cdot 2^{10}}{8 \cdot 64} = 64$ sets.

   (b) cache tag bits $= log_2(\frac{RAM_MAX}{S \cdot B}) = log_2(\frac{2^{64}}{64 \cdot 64}) = 52$ bits.
   effective tag bits $= log_2(\frac{RAM}{S \cdot B}) = log_2(\frac{2^{36}}{64 \cdot 64}) = 24$ bits.

   (c) i. Cache tag: $\lfloor \frac{addr}{S \cdot B} \rfloor =$ `0xDE14`
   ii. Set index: $\lfloor \frac{addr \bmod S \cdot B}{B} \rfloor =$ `0x1E`.
   iii. Block offset $(addr \bmod S \cdot B) \bmod B =$ `0x3A`.

2. *Cache mechanics (12 pts)* Consider a direct mapped cache of size 16KB with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate. Remember that `sizeof(int) == 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.

   Now consider the following two implementations of a horizontal flip and copy of the matrix. Assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

(a)
```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS]) {
  int i, j;
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
      dest[i][COLS - 1 - j] = src[i][j];
}
```

i. What is the cache miss rate if ROWS = 64 and COLS = 64?

ii. What is the cache miss rate if ROWS = 96 and COLS = 64?

(b)
```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS]) {
  int i, j;
  for (j = 0; j < COLS; j++)
    for (i = 0; i < ROWS; i++)
      dest[i][COLS - 1 - j] = src[i][j];
}
```

i. What is the cache miss rate if ROWS = 64 and COLS = 64?

ii. What is the cache miss rate if ROWS = 96 and COLS = 64?

Solution:

(a) i. ROWS = 64, COLS = 64: 25%

ii. ROWS = 96, COLS = 64: 25%

(b) i. ROWS = 64, COLS = 64: 25%

ii. ROWS = 96, COLS = 64: 75%

3. *Cache mechanics (20 pts)* 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. You are given the following definitions:

```
typedef struct{
  float irr[3];
  short theta;
  short phi;
} photon_t;
photon_t surface[16][16];
register int i, j, k;
```

Also assume that

- `sizeof(short) = 2` and `sizeof(float) = 4`
- `surface` 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 ++) {
      surface[i][j].irr[k] = 0.;
    }
    surface[i][j].theta = 0;
    surface[i][j].phi = 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 < 16; k ++) {
      surface[j][i].irr[k] = 0;
    }
    surface[j][i].theta = 0;
    surface[j][i].phi = 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?

(a) i. Miss rate for writes to surface: 5%

ii. Miss rate for writes to surface: 10%

(b) i. Miss rate for writes to surface: 20%

ii. Miss rate for writes to surface: 20%

4. *Roofline (25 pts)* Assume the following hardware parameters for an Intel CPU:

- Can issue one scalar add and two scalar multiplications per cycle.
- CPU frequency is 3.5 GHz.
- Last level cache (LLC) size is 8 MB and cache block size is 64 bytes.
- Memory bandwidth is 28 Gbyte/sec.

Draw a roofline plot for single 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 on the platform above (all the matrices have size $N \times N$):

```
void compute1(float *A, float *B, float *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] = 0.4*C[i][j] + 0.6*A[i][k]*B[j][k];
}
```

(c) Is it possible to reach peak performance for `compute1`? If not, include a tighter performance bound specific for `compute1` in your roofline plot.

Again consider running the following code on the platform above (all the matrices have size $N \times N$):

```
void compute2(float *A, float *B, float *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] = C[i][j] + 0.6*A[i][k]*B[j][k];
}
```

(d) Is it possible to reach peak performance for `compute2`? If not, include a tighter performance bound specific for `compute2` in your roofline plot.

Finally consider the following code (all the matrices have size $N \times N$):

```
void compute3(float *A, float *B, float *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] = C[i][j] + 0.6*A[k][i]*B[k][j];
}
```

(e) Can the execution of `compute3(A,B,C,65536)` reach peak performance? If not, include a tighter performance bound specific for this execution of `compute3` in your roofline plot.

Solution:

(a) The given Intel CPU can perform two mults and 1 add per cycle, hence peak performance is $\pi = 3$ flops/cycle

(b) The given Intel CPU can transfer 28 Gbyte/sec and has frequency of 3.5 GHz, thus peak bandwidth is $\beta = 28 \times 10^9 / 3.5 \times 10^9 = 8$ bytes/cycle

(c) The function `compute1` loads $3N \times N$ matrices. The access pattern for the function allows for spatial and temporal locality so that once the matrices are loaded into cache for $3N^2 \leq 8MB$ there would be no cache misses. In order to achieve peak performance, the code should have the perfect mix of adds and mults (1:2). The cost for the function `compute1` in terms of adds and mults is $(N^3, 3N^3)$. The processor can perform N^3 adds and $2N^3$ mults in N^3 cycles. Thus,

$$r_{mixed} = N^3 \; cycles$$

The remaining N^3 mults can be performed in $N^3/2$ cycles. Thus,

$$r_{mults} = N^3/2 \; cycles$$

The lower bound for the computation is

$$r = N^3 + N^3/2 = 3N^3/2 \; cycles$$

Thus, the theoretical peak performance for function `compute1` is,

$$\pi_{compute1} = 4N^3/(3N^3/2) = 8/3 \text{ flops/cycle}$$

Hence it is not possible to achieve peak performance for function `compute1` theoretically.

(d) The function `compute2` has the same access pattern as `compute1`. The computational cost for the the function `compute2` in terms of adds and mults is $(N^3, 2N^3)$. Thus it has perfect balance of adds and mults (1:2). The processor can perform N^3 adds and $2N^3$ mults in N^3 cycles. Thus the lower bound for the computation is

$$r = N^3 \; cycles$$

Thus, the theoretical peak performance for function `compute2` is,

$$\pi_{compute1} = 3N^3/N^3 = 3 \text{ flops/cycle}$$

Hence it is possible to achieve peak performance for function `compute2` theoretically.

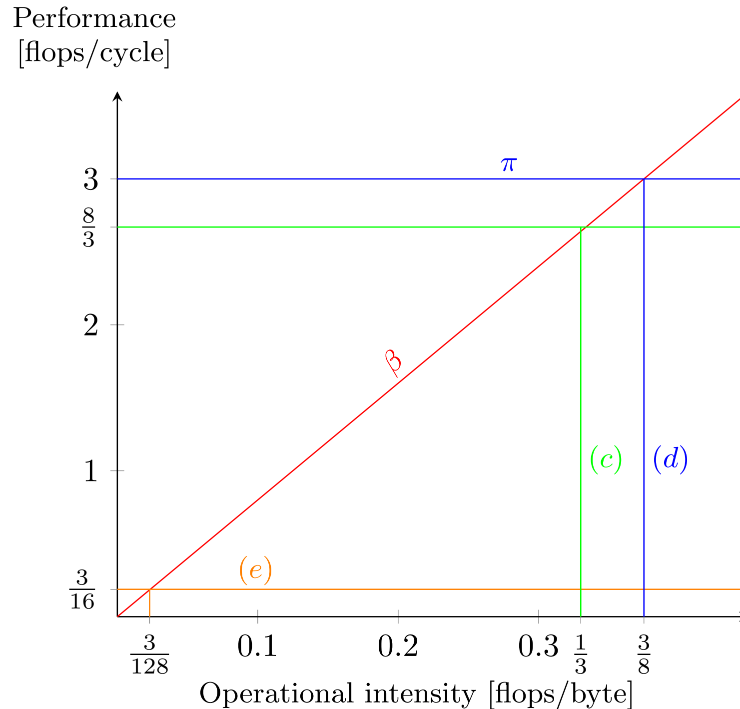

Figure 1: Roofline plot for Ex. 4

(e) The function `compute3` has the same number of adds and mults as `compute2`. However, it has a different excess pattern. In the triple loop, the elements of matrices A and B are accessed columnwise. The cache size is 8 MB and the size of each cache block is 64 bytes. Thus, the number of entries in cache are,

$$2^{23}/2^6 = 2^{17}$$

For $N = 65536$, there are 2^{16} rows (columns) in A and B each. So after one iteration of innermost k loop, the cache becomes full with rows of A and B. Assuming cache follows least recently used policy (LRU) for cache line replacement, the subsequenct iterations result in cache miss for each access to matrices A and B. Thus the computation become memory bound. The operation intensity $I_{compute3}$ can be calculated as,

$$I_{compute3} = 3N^3/(64N^3 + 64N^3 + N^2) = 3N/(128N + 1) \approx 3/128 \text{ flops/byte}$$

Thus the peak performance for the function `compute3` is bounded by,

$$\pi_{compute3} = \beta I_{compute3} = 3/16 \text{ flops/cycle}$$

Hence it is not possible to achieve peak performance for function `compute3` theoretically.

All the answers above are summarized in the roofline plots of Figure 1.

5. *Roofline and MMM (26 pts)*
We consider a processor with the following parameters:

- π: Peak performance in flops/cycle.
- β: Peak bandwidth in bytes/cycle.

- γ: Cache size in bytes (there is only one cache).

(a) Assume a function that is run for a given input size N. Show that if the dot of this function in the roofline plot is on the ridge point (the point where the performance bound and the bandwidth bound intersect) then $T_{\text{comp}} = T_{\text{mem}}$. Here, T_{comp} is the time required to execute the floating point ops, and T_{mem} is the time to transfer the needed data from and to memory. Under which condition is the reverse also true?

(b) Assume an implementation of matrix-matrix multiplication (MMM) with cost $W(N) = 2N^3$ flops for square matrices of size $N \times N$. Also assume that this implementation incurs the minimal number of cache misses possible and the dot in the roofline plot for all sizes lies on the ridge point. Further, assume now the use of another CPU with the same parameters except for peak performance, which is $\pi' = \alpha\pi$ flops/cycle, with $\alpha > 1$. On this CPU the MMM function will be memory bound. By which factor should the cache size be increased to bring the dot back to the ridge point? Explain.

Solution:

(a) Having the dot on the ridge implies that

$$I(N) = \frac{W(N)}{Q(N)} = \frac{\pi}{\beta} \ ,$$

where $W(N)$ is the program runtime in flops and $Q(N)$ is the amount of off-chip traffic in bytes. From this equation we can determine that

$$\frac{W(N)}{\pi} = T_{\text{comp}} = \frac{Q(N)}{\beta} = T_{\text{mem}} \text{ cycles} \ .$$

The reverse is also true under the condition that there is perfect overlap between T_{comp} and T_{mem}:

$$T = T_{\text{comp}} = T_{\text{mem}} \Rightarrow \pi = \frac{W(N)}{T} \ , \beta = \frac{Q(N)}{T} \ ,$$

where T is the total execution time of the function.

(b) It was mentioned in class (see notes from Lecture 6) that the optimal operational intensity for MMM is $I(N) = \Theta(\sqrt{\gamma})$. Assuming that the dot lies on the ridge implies that

$$I(N) = \Theta(\sqrt{\gamma}) = \frac{\pi}{\beta} \ .$$

On the second CPU with performance $\pi' = \alpha\pi$, maintaining the dot on the ridge would require the presence of a cache with a size γ' a factor α^2 larger than γ:

$$\frac{\pi'}{\beta} = \frac{\alpha\pi}{\beta} = \alpha\Theta(\sqrt{\gamma}) = \Theta(\sqrt{\alpha^2\gamma}) = \Theta(\sqrt{\gamma'}) \ .$$