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.

Solution:

(a) `mat` and `v` are accessed row-wise n^2 and $2n^2$ times respectively. For every iteration, we assume that `v` is accessed before `mat`.

i. For $n = 64$, $sizeof(\texttt{mat}) == cache\_size$. First 64 elements of `mat` and `v` map to the same cache lines. There are conflict misses when code accesses blocks of `v` and `mat` that map to the same cache line. This happens for $(i = 0, 0 \leq j \leq 3)$ resulting in 4 conflict misses for `v` and 3 for `mat`. The remaining misses are compulsory misses for `mat` and `v`. Thus, cache miss rate $= \frac{(7+1024+16)\times 100}{3*64*64} \approx 8.5\%$.

ii. For $n = 128$, $sizeof(\texttt{mat}) == 4 * cache\_size$. There will be conflict misses between `v` and `mat` when $(i = 0, 0 \leq j \leq 3)$, $(i = 32, 32 \leq j \leq 35)$, $(i = 64, 64 \leq j \leq 67)$ and $(i = 96, 96 \leq j \leq 99)$ resulting in 28 misses. The remaining misses are compulsory misses for `mat` and `v`. Thus cache miss rate $= \frac{(28+4096+32)\times 100}{3*128*128} \approx 8.5\%$.

(b) The code accesses `mat` column-wise now.

i. `mat` fits in the cache. There will be conflict misses between `v` and `mat` whenever $j = 0$. Each conflicting cache line has 3 misses for `mat` and 4 for `v`. There are 16 conflicting cache lines and thus, the total number of conflict misses are 112. The remaining misses are compulsory misses for `mat` and `v`. Thus cache miss rate $= \frac{(112+1024+16)\times 100}{3*64*64} \approx 9.4\%$.

ii. `mat` does not fit in cache. As a result, every access to `mat` results in cache miss. There is 1 conflict miss for `v` whenever $j \in \{0, 32, 64, 96\}$. The remaining misses are compulsory misses for `v`. Thus cache miss rate $= \frac{(512+128*128+32)\times 100}{3*128*128} \approx 34.4\%$.

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.

```c
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.

Solution:
The code accesses `u`, `v` and `w` n, n^2 and n^2 times repectively. The AVX load instruction fetches 8 floats into the cache in 2 cache lines. We assume that both cache lines are loaded together so that one AVX access results in one compulsory miss.

(a) `w` fits in cache whereas `u` and `v` conflict with `w` on 16 cache lines each. There is 1 conflict miss for `u` due to conflict with `w` for the second iteration of the i-loop. There are 8 conflict misses for `v` due to conflict with `w` for $i = 2$. The remaining misses are compulsory misses for `u`, `v` and `w`. Thus cache miss rate $= \frac{(9+512+8+16)\times 100}{2*64*64+64} \approx 6.6\%$.

(b) `w` does not fit in cache whereas `u` and `v` conflict with `w` on 32 cache lines each. There are 4 conflict misses for `u` due to conflict with `w` for $i \in \{1, 33, 65, 97\}$. There are 64 conflict misses for `v` due to conflict with `w` for $i \in \{2, 34, 66, 98\}$. The remaining misses are compulsory misses for `u`, `v` and `w`. Thus cache miss rate $= \frac{(68+2048+16+32)\times 100}{2*128*128+128} \approx 6.6\%$.

(c) Yes, for $N = 64$ the conflict misses for `v` can be reduced to zero by allocating `u`, `v` and `w` in the order `v`,`w`,`u`.

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`:

```c
// 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;
}
```

```c
// 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!)

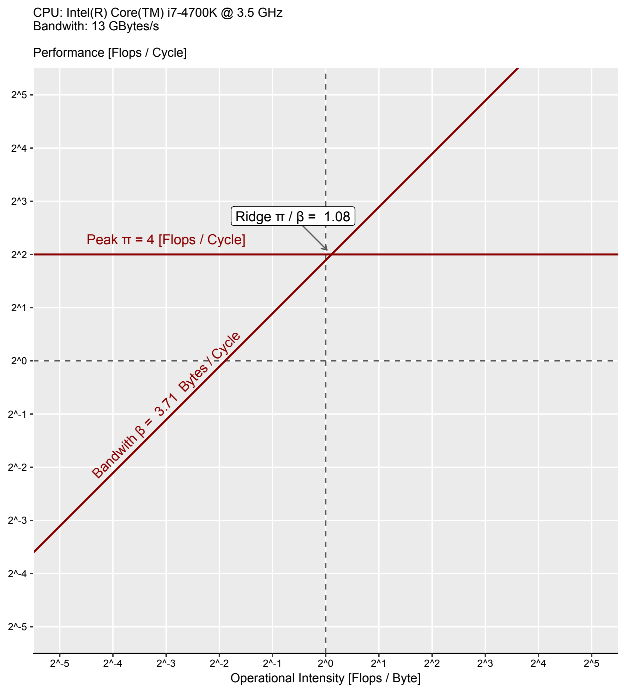
(c) For each of the three kernels, assuming that we change the data type from double to float

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

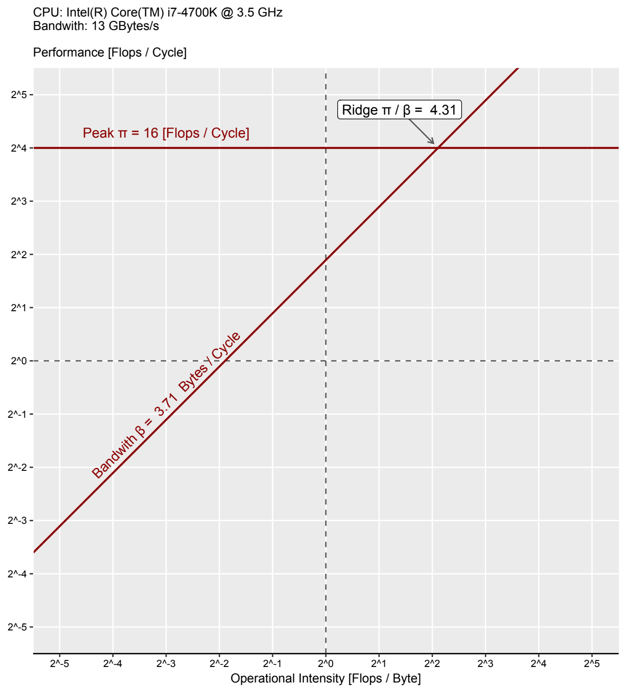
Solution

(a) Haswell has two FMA units, each of them capable of processing 4 doubles. As FMA is counted as two flops, we can do 16 flops at the same time when using the vectorized FMAs, and 4 flops when we use the scalar FMAs. The bandwidth is 13 GB/s and the frequency of the CPU is 3.5GHz, therefore the bandwidth expressed in bytes per cycle is $13/3.5 = 3.71$ bytes/cycle. Figure 1 shows the two roofline plots.

- Figure 1(a): In the scalar FMA, the ridge is defined at $4/3.71 = 1.08$ flops/byte.
- Figure 1(b): In the vectororized FMA, the ridge is defined at $16/3.71 = 4.31$ flops/byte.



(a) Scalar FMA



(b) Vectorized FMA

Figure 1: Roofline plot for scalar and vectorized FMA

(b) Note that the whole working set of each kernel fits the cache. Therefore we would experience only mandatory cache misses, and the memory will be loaded into the cache only once. Since we are in a write-allocate scenario, upon each write-miss, memory location is loaded in the cache, and then write is performed. As a result, the operational intensities are given as:

$$\texttt{kernel1}: O(N) = \frac{W(N)}{Q(N)} = \frac{2 \cdot N}{8 \cdot (2 \cdot N_{reads} + N_{writes})} = \frac{1}{12}$$
$$\texttt{kernel2}: O(N) = \frac{W(N)}{Q(N)} = \frac{2 \cdot N^3}{8 \cdot (N^2 \cdot 3_{reads} + N^2_{writes})} = \frac{N}{16}$$
$$\texttt{kernel3}: O(N) = \frac{W(N)}{Q(N)} = \frac{4 \cdot N^3}{8 \cdot (\langle N^2 \cdot 3 + N \rangle_{reads} + \langle N^2 + N \rangle_{writes})} = \frac{N^2}{8N+4}$$

Figure 2 shows location of the 3 kernels on the roofline plot with the calculated operational intensities for different values of N, assuming maximal theoretical performance.

`kernel1`: Has a constant operational intensity. As a result, this kernel becomes memory bound with respect to the given CPU bandwidth. This means that for any value of N, the maximal theoretical performance of this kernel will not exceed 0.31 flops/cycle, regardless whether we deal with vectorized or scalar code, as shown on Figure 2(a) and 2(b).

`kernel2`: The operational intensity of this kernel is dependent on N. This means that for lower values of N, the kernel is memory bound. However for big enough N, this kernel is able to reach the maximal theoretical performance of the machine. This means that in the scalar version it will reach 4 flops/cycle and in the vectorized version 16 f/c, as shown on Figure 2(a) and 2(b).

`kernel3`: Similar to `kernel2`, this kernel is memory bound for small values of N, and compute bound on larger values of N as shown on Figure 2(c). Theoretically this kernel could potentially reach the maximal theoretical bound of 4 f/c. However, due to dependencies, this bound is unrealistic.

(c) Assuming that the double precision floating point data structure is changed with a single precision floating point structure, the new dependencies are defined as follows:

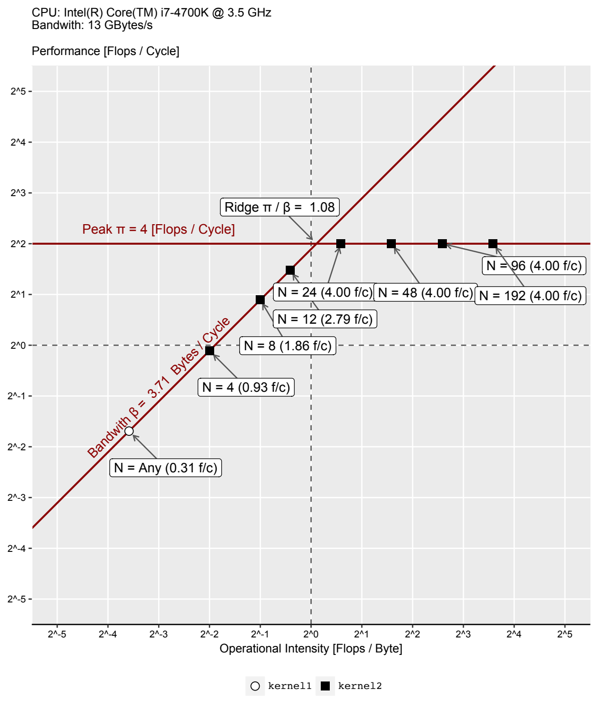
`kernel1`: $O(N) = \frac{W(N)}{Q(N)} = \frac{2 \cdot N}{4 \cdot (2 \cdot N_{reads} + N_{writes})} = \frac{1}{6}$
`kernel2`: $O(N) = \frac{W(N)}{Q(N)} = \frac{2 \cdot N^3}{4 \cdot (N^2 \cdot 3_{reads} + N^2_{writes})} = \frac{N}{8}$
`kernel3`: $O(N) = \frac{W(N)}{Q(N)} = \frac{4 \cdot N^3}{4 \cdot (\langle N^2 \cdot 3 + N \rangle_{reads} + \langle N^2 + N \rangle_{writes})} = \frac{N^2}{4N+2}$

The new theoretical bounds are such that in the scalar scenario, we can still execute 4 flops / cycles using the two FMA units on the Haswell CPU. However, things change when we use vectorized code. Namely, we can now process 8 doubles at a time, making the maximal theoretical performance of the CPU to be 32 flops / cycle. The new theoretical bounds on performance are defined as:

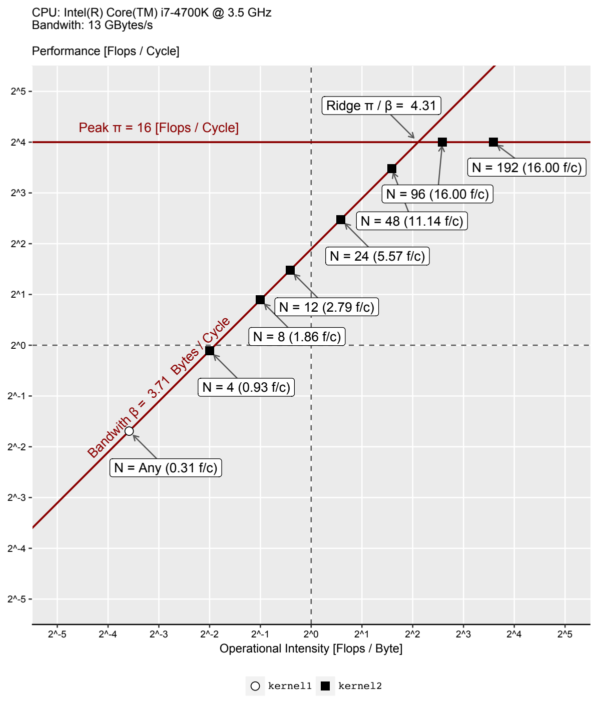
`kernel1`: This kernel still has constant operational intensity and the maximal theoretical performance of this kernel will not exceed 0.62 flops/cycle, regardless whether we deal with vectorized or scalar code.

`kernel2`: Similarly to the previous case, this kernel will be memory bound and compute bound for different values of N. However for big enough N (N = 96), it will be able to reach the maximal performance of 32 flops/cycle.

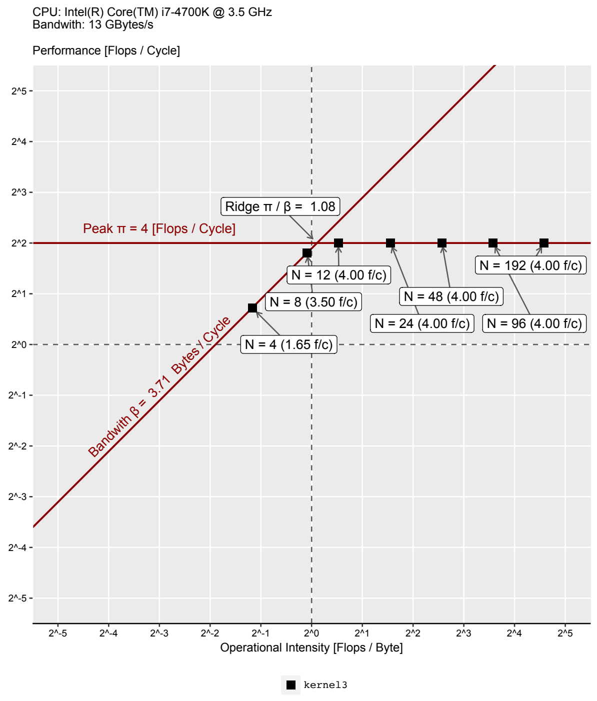
`kernel3`: This kernel will be memory bound only for $N \in \{1, 2, 3, 4\}$, and almost always compute bound. As a result, the maximal theoretical performance for this kernel will be 4 flops/cycle.



(a) Scalar `kernel1` and `kernel2`



(b) Vectorized `kernel1` and `kernel2`



(c) Scalar `kernel3` and `kernel2`

Figure 2: `kernel1`, `kernel2` and `kernel3` located on the roofline, assuming maximal theoretical performance