263-0007-00: Advanced Systems Lab
Assignment 4: 120 points
Due Date: April 17th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2021/
Questions: fastcode@lists.inf.ethz.ch

Academic integrity:
All homeworks in this course are single-student homeworks. The work must be all your own. Do not copy any parts of any of the homeworks from anyone including the web. Do not look at other students' code, papers, or exams. Do not make any parts of your homework available to anyone, and make sure no one can read your files. The university policies on academic integrity will be applied rigorously.

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=14942.
- (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 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.
- (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 points in a homework are given for neatness.

Exercises

1. *Associativity (20 pts)*

   Consider the following function, executed on a machine with a write-back/write-allocate cache with blocks of size 16 bytes, a total capacity of 128 bytes and with a LRU replacement policy. Arrays x, y and z are cache-aligned (first element goes into first cache block). Assume that memory accesses occur in exactly the order that they appear in the code. Thus, no optimizations are performed that reduce the memory accesses or reorder computations. The variables `i` and `t` remain in registers and do not cause cache misses.

```
1 void calculate1 (double* x, double* y, double* z) {
2     double t = 0.0;
3     for (int i = 1; i <= 10; i++) {
4         t += x[(4*i)% 12];
5         t += y[(3*i-2)% 8];
6         z[(4*i-1)% 8] = t;
7     }
8 }
```

   (a) Considering cache misses from both **reads and writes**, compute the following two things: i) the **miss/hit pattern** for x, y and z (something like `x:MMHHM..., y:MMMH...`); ii) the **operational intensity** (in flops/byte) of the above computation for the following cases. For the operational intensity assume empty caches and consider only reads but note: arrays that are only written are also read and this read should be included. Show your work.

i. Miss/hit pattern and operational intensity when the cache is 2-way set associative.
Solution:
`x:MMMHHHHHMMH, y:MMMMHMHHMH, z:MMHHHHHHHHH`.
$I_{read} = \frac{W}{Q_{read}} = \frac{20}{13 \cdot 16} \approx 0.0961$.

ii. Miss/hit pattern and operational intensity when the cache is 4-way set associative.
Solution:
`x:MMMMHHMMHH, y:MMMMHMHHHM, z:MMHHHHHHHH`.
$I_{read} = \frac{W}{Q_{read}} = \frac{20}{14 \cdot 16} \approx 0.0893$.

(b) Assuming a 2-way set associative cache (motivate your answers):

i. What kind(s) of locality do the accesses to array x have?
Solution: Temporal.

ii. What kind(s) of locality do the accesses to array y have?
Solution: Spatial and Temporal.

2. *Cache Mechanics (35 pts)* Consider the following code, executed on a machine with a write-back/write-allocate direct-mapped cache with blocks of size 32 bytes, a total capacity of 12KiB and with a LRU replacement policy. Assume that memory accesses occur in exactly the order that they appear. The variables `i,j,k,t` and `n` remain in registers and do not cause cache misses. Arrays A and B are cache-aligned (first element goes into first cache block). For this and the following exercises, assume a cold cache scenario. `sizeof(double) = sizeof(uint64_t) = 8`.

```
1  #define PADDING_SIZE 1
2  typedef struct {
3      double   v;
4      double   d[3];
5      uint64_t u[3];
6      uint64_t pad[PADDING_SIZE];
7  } struct_t;
8
9  void calculate1(struct_t *A, struct_t *B, int n) {
10     double t;
11     for (int i = 0; i < n; ++i) {
12         for (int j = 0; j < n; ++j) {
13             t = A[i*n + j].v;
14             for (int k = 0; k < 3; k++) {
15                 t += A[i*n + j].d[k];
16             }
17             for (int k = 0; k < 3; k++) {
18                 A[i*n + j].u[k] = 0;
19             }
20             t += B[j*n].v;
21             B[j*n].v = t;
22         }
23     }
24 }
```

Considering cache misses from both **reads and writes**, compute i) the **cache miss rate** and ii) the **operational intensity** (in flops/byte) of the above computation for the following cases. For the operational intensity assume only reads, i.e., data movement from main memory to cache. Show enough details so we can see your reasoning.

(a) Miss rate and operational intensity for $n = 8$.
Solution: Array A is accessed $7n^2$ times and array B is accessed $2n^2$ times, resulting in $9n^2$ memory accesses. Since an element `struct_t` fit in two cache blocks, accesses to A will result in 2 compulsory misses in every iteration. This results in a total of $2n^2$ misses when accessing A. Array B accesses the same n elements for all i and the elements are accessed with a stride n. Since B fits completely in cache for $n = 8$ it will not have conflicts misses with itself. For $i = 0$, there will be n compulsory misses when accessing B. For $i = 1$, array A will conflict with B resulting in one conflict miss. Array A will continue conflicting with and element of B in the next

iterations of i. Thus, there will be $n-1$ conflict misses and n compulsory misses when accessing B in total. The miss rate is therefore $\frac{2n^2+2n-1}{9n^2} = \frac{143}{576} \approx 0.248$. The operational intensity is $I = \frac{4n^2}{143 \cdot 32} = \frac{256}{143 \cdot 32} \approx 0.056$.

(b) Miss rate and operational intensity for $n = 16$.

Solution: Similarly to the previous task, array A produces $2n^2$ compulsory misses. For $i = 0$, there will be n compulsory misses when accessing B. In this case, array B does not completely fits in cache. In fact, there will be conflict between its first four accesses (`B[0], B[16], B[32], B[48]`) and its last four accesses (`B[176], B[192], B[208], B[224]`). Thus, for $i = 1, \ldots, n-1$, there will be 8 conflict misses of array B with itself, resulting in $8(n-1)$ conflict misses. Further, for $i = 4, \ldots, i = 11$ there will be an additional conlfict miss with array A, resulting in 8 extra misses. The miss rate is therefore $\frac{2n^2+n+8(n-1)+8}{9n^2} = \frac{2n^2+9n}{9n^2} = \frac{656}{2304} \approx 0.285$. The operational intensity is $I = \frac{4n^2}{656 \cdot 32} = \frac{1024}{656 \cdot 32} \approx 0.0488$.

(c) Miss rate and operational intensity for $n = 16$ and `PADDING_SIZE` $= 5$.

Solution: Array A produces the same misses as before. For $i = 0$, array B produces n compulsory misses. For $i = 1$, the first 8 accesses of B are already evicted from cache due to conflicts with the last 8 accesses. This will result in n conflicts misses. The same pattern repeats for $i = 2, \cdots, n-1$. The miss rate is therefore $\frac{2n^2+n^2}{9n^2} = \frac{3n^2}{9n^2} = \frac{1}{3} \approx 0.333$. The operational intensity is $I = \frac{4n^2}{3n^2 \cdot 32} = \frac{4}{3 \cdot 32} \approx 0.0417$.

3. *Rooflines (40 pt)* Consider a processor with the following hardware parameters (assume $1\text{GB} = 10^9\text{B}$):

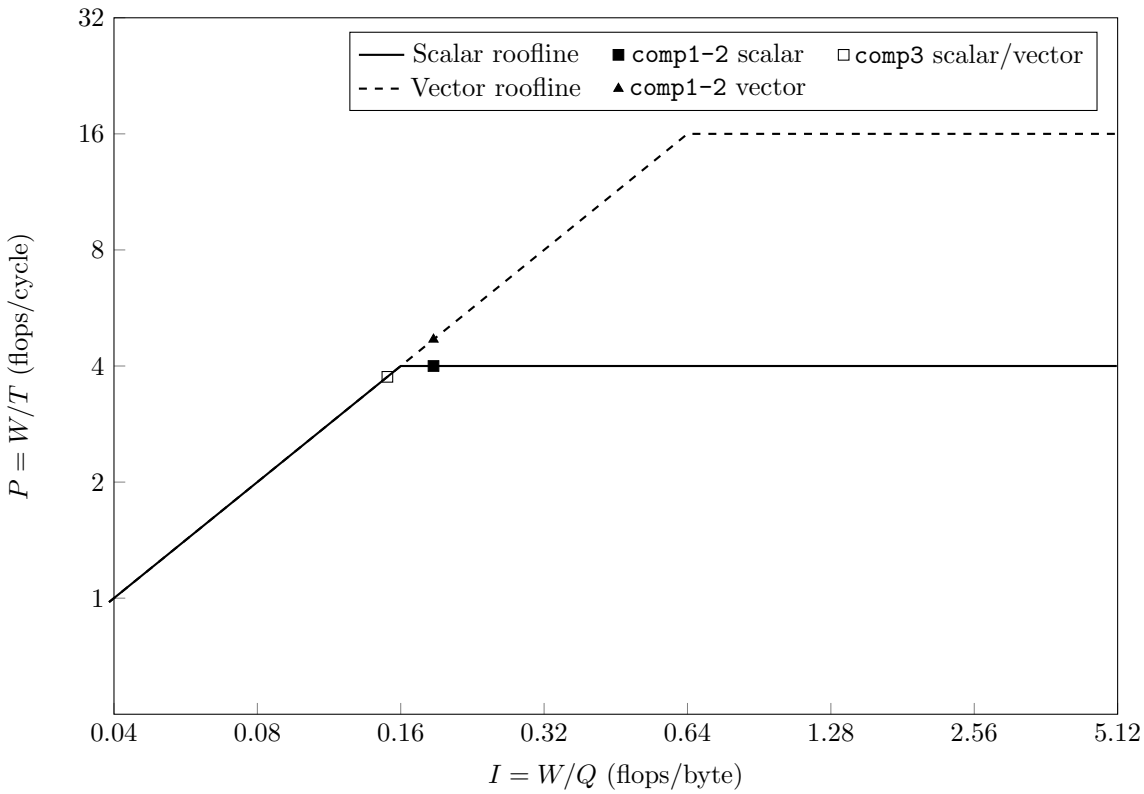
- SIMD vector length of 256 bits.
- Two instruction ports that execute floating point operations:
  - Port 0 (P0): FMA, ADD, MUL
  - Port 1 (P1): FMA, ADD, MUL

  Each port can issue 1 operation per cycle. Each operation has a latency of 1.
- One write-back/write-allocate cache.
- Read bandwidth from the main memory is 50 GB/s.
- Processor frequency is 2 GHz.

(a) Draw a roofline plot for the machine. Consider only double-precision floating point arithmetic. Consider only reads. Include a roofline for when vector instructions are not used and for when vector instructions are used.

Solution: The memory bandwidth $\beta = 50 \cdot 10^9$ bytes / $(2 \cdot 10^9$ cycles$) = 25$ bytes/cycle. The peak scalar performance is $\pi_s = 4$ flops/cycle, due to being capable of performing 2 FMA instructions per cycle. Thus, the scalar operations become memory bound at $I = \pi_s/\beta = 4/25 = 0.16$ flops/byte. A SIMD vector of 256 bits can fit $256/64 = 4$ double precision floating point numbers, so the peak vector performance grows fourfold: $\pi_v = 16$ flops/cycle. Thus, the vector operations become memory bound at $I = \pi_s/\beta = 16/25 = 0.64$ flops/byte.

Roofline plot:



(b) Compute a hard upper bound on the operational intensity I of the functions below based on compulsory misses. Based on this I alone, i.e. ignoring instruction mix, add the maximum performance of each function to the roofline plot assuming first that vector instructions are not used (three dots). Then, assume that vector instructions are used to speedup the computations and add their new maximum performance (three additional dots). At the end, there should be six dots in the roofline. Consider only reads, cold-cache scenario, only compulsory misses. Ignore the effects of aliasing and assume that no optimizations that change operational intensity are performed (the computation stays as is).

```
1 void comp1(double *x, double *y, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     x[i] += y[i] + c * y[i + 32];
4   }
5 }
```

```
1 void comp2(double *x, double *y, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     x[i] += y[i] + c + y[i + 32];
4   }
5 }
```

```
1 void comp3(double *A , double *u, double *v, int n) {
2   const int r = 3;
3   for (int i = 0; i < r; i++) {
4     for (int j = 0; j < n; j++) {
5       A[i * n + j] += u[j] * v[j];
6     }
7   }
8 }
```

Solution:
(The performance dots have been placed on the plot in (a)).

Scalar:

- `comp1`: With $I = 3$ flops / 16 bytes $= 0.1875$ flops/byte, we reach the peak scalar performance $P = 4$ flops/cycle.
- `comp2`: Same as above.
- `comp3`: With $I = 6n$ flops / $(24 + 16)n$ bytes $= 0.15$ flops/byte, we reach $P = 3.75$ flops/cycle.

Vectorized:

- `comp1`: With vectorization, we now achieve $P = 4.6875$ flops/cycle.
- `comp2`: Same as above.
- `comp3`: We reach $P = 3.75$ flops/cycle, same as prior to vectorization. The operational intensity is too low to benefit from vector instructions.

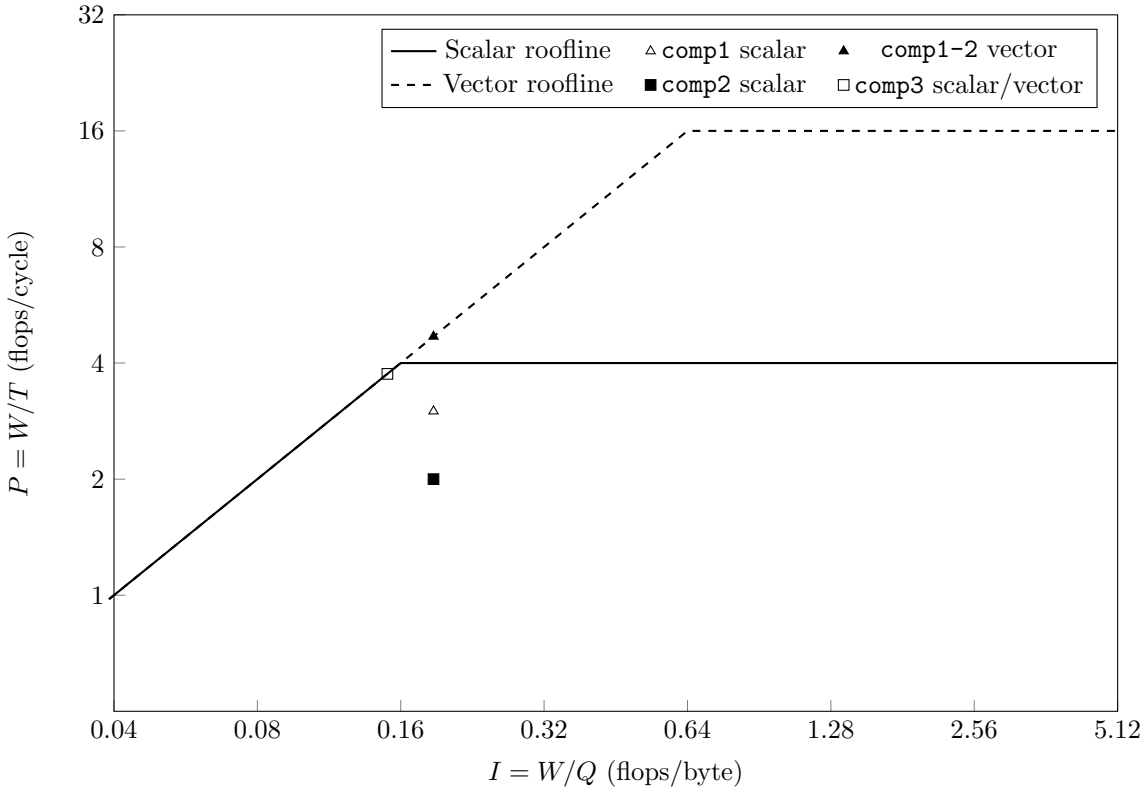
(c) Now, derive a hard upper bound on the performance of each function based on the **instruction mix** and compulsory misses. Again, assume that no optimizations that change operational intensity are performed, and FMAs are used to fuse an addition with a multiplication whenever applicable. For readability, place the new performance dots on a separate roofline plot (there should be six dots).

Solution:
Scalar:

- `comp1`: We can only reach 3 flops/cycle (1 FMA and 1 ADD per cycle).
- `comp2`: We can only reach 2 flops/cycle (2 ADDs per cycle).
- `comp3`: The maximal performance does not change.

In the vectorized versions, the maximal performances remain the same as in the previous subtask.

Roofline plot:



(d) How will the operational intensity and performance of `comp3` change if we increase r (i.e. if we manually assign higher values to r in line 2)? What is the best possible performance of `comp3` when increasing r and when implemented with and without vector instructions?

Solution: The operational intensity will increase alongside with r. Assuming that we have infinitely large cache, as r grows, we approach I of value:

$$\lim_{r\to\infty} \frac{2rn}{8rn+16n} = \frac{1}{4} = 0.25\text{flops/byte}$$

With this operational intensity, the performance becomes compute bound when using scalar instructions yielding a performance of 4 flops/cycle. With vector instructions the performance is 6.25 flops/cycle.

4. *Cache Miss Analysis (20 pts)*

Consider the following computation that performs the matrix multiplication of a triangular matrix A with a square matrix B of size $n \times n$. This computation is illustrated in Figure 1;

```
1 void mmm_triangular(double *A, double *B, double *C, int n) {
2     for( int i = 0; i < n; i++ )
3         for( int j = 0; j < n; j++ )
4             for( int k = i; k < n; k++ )
5                 C[n*i + j] += A[n*i + k] * B[n*k + j];
6 }
```

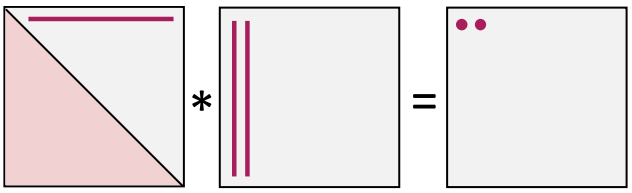


Figure 1: MMM with triangular matrix.

Assume that the code is executed on a machine with a write-back/write-allocate fully-associative cache with blocks of size 64 bytes, a total capacity of $\gamma = 8$KiB and with a LRU replacement policy. Assume that $\gamma << n$.

(a) Calculate the total number of cache misses that this computation has. You can ignore lower order terms. Show your work.

Solution: Line 5 in the code is executed $\frac{n^2(n+1)}{2} \approx \frac{n^3}{2}$ times. The accesses to A will benefit from spatial locality. Thus, only $\frac{1}{8}$ of its accesses will be misses. Array B is accessed by column; thus, there is no spacial locality and every access will be a miss. Finally, array C will benefit from temporal locality in the innermost loop. Thus, the cache misses from C will be $O(n^2)$. The total number of cache misses is $\frac{1}{8} \cdot \frac{n^3}{2} + 1 \cdot \frac{n^3}{2} + O(n^2) \approx \frac{9n^3}{16}$.

Now assume that we use blocking to improve the locality of the computation. For this case, we tile the matrices using blocks of size $b \times b$ and some triangular blocks in the diagonal of matrix A. b is divisible by 8. Figure 2 shows the strategy used. Answer the following. Show enough details so we can see your reasoning.

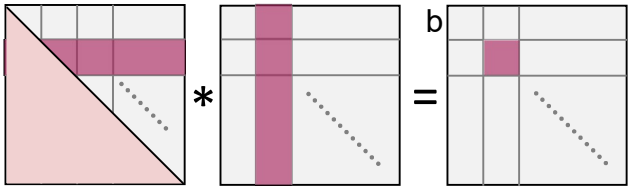


Figure 2: MMM with triangular matrix after blocking.

(b) Based on the characteristics of the machine, determine a suitable value of b that improves locality, i.e. that reduces cache misses.

Solution: There are two option:

i. We choose to fit a block of A, a block of B, and a block of C in cache. In this case we have that $3b^2 \leq \gamma$, which results in $b \leq 18.5$. We choose $b = 16$ to be block aligned.

ii. Following the slides of the course (linear-algebra-MMM, p.21), we can give a closer analysis of the working set needed. In this case, we choose to fit a complete block of B, two row of A and one row plus one element of C. This is to take into account LRU replacement policy. Thus, $\lceil \frac{b^2}{8} \rceil + 3\lceil \frac{b}{8} \rceil + 1 \leq \frac{\gamma}{8}$, which yeilds $b \leq 30$. We choose $b = 24$ to be block aligned.

(c) Calculate the total number of cache misses that this computation has when using blocking. You can ignore lower order terms.

Solution: Similar to task a), there are $\frac{(\frac{n}{b})^3}{2}$ matrix multiplications that are performed using blocks. We simply have to determine now the number of misses required to bring the blocks needed in each blocked matrix multiplication. For the two options of blocking mentioned in b):

i. We need to bring the block of A and B, the block of C will be reused due to temporal locality and becomes a lower order term. Loading a block produces $\frac{b^2}{8}$ misses. Thus, the number of misses is $\frac{2b^2}{8} \cdot \frac{(\frac{n}{b})^3}{2} = \frac{n^3}{8b}$

ii. This case is similar, but now there is no reuse in C for the next blocked matrix multiplication. Thus, the number of cache misses is $\frac{3b^2}{8} \cdot \frac{(\frac{n}{b})^3}{2} = \frac{3n^3}{16b}$