263-0007-00: Advanced Systems Lab
Assignment 4: 120 points
Due Date: April 10th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2025/
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
- (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.

The exercises start from the next page.

Exercises

1. *Stride Access (30 pts)*

   Consider the following code executed on a machine with a cache with blocks of size 32 bytes, a total capacity of 2 KiB and a write-back/write-allocate policy. Assume that the only memory accesses are to entries of O and A and occur in the order that they appear (from right to left when in the same line). The cache is initially cold and array A begins at memory address 0, while array O begins immediately after the last element of A. You can assume that A is of size 2^n, O is of size $n \times 2^{n-1}$ and $n \leq 32$. As the size of a boolean is implementation-specific, you can assume it to be 4 bytes.

```c
void qmc(bool* A, bool* O, int n){
  for (int v = 0; v < n; v++){
    unsigned int block_size = 1U << v;
    unsigned int block_count = 1U << (n - v - 1);
    bool* O_v = &O[((long) v) << (n - 1)];
    for (int b = 0; b < block_count; b++) {
        bool* A_b =  &A[2 * b * block_size];
        for (int p = 0; p < block_size; p++) {
            O_v[b * block_size + p] = A_b[p] && A_b[block_size + p];
        }
    }
  }
}
```

   Answer the following. Justify your answers. In case you use a script to compute any of your answers, hand in the script as well, use only Python.

   (a) Consider a direct-mapped cache, determine the largest n such that only compulsory misses occur.

   Solution: Since A and O are stored continuously and we have a direct-mapped cache, we can find two sufficient conditions for having only compulsory misses:

   i. A and B completely fit into cache.

   ii. A and all but the last row of B fit into cache and there are at least 28 additional bytes: this holds because parts of A may be evicted element-by-element after their last use. The additional space is required because the size of a cache line does not match the size of a boolean and hence, when the position in A and O is mapped to the same cache line, the two need to be far enough apart from one another to avoid evicting the current line.

   For $n \geq 4$ the second condition is more tight, so:

$$\text{Only compulsory misses occur} \Leftrightarrow 4 \cdot (2^n + (n-1) \cdot 2^{n-1}) + 28 \leq 2048$$

   For $n = 7$, the inequality does not hold (by a difference of 28) and for $n = 6$ it does, so this is the largest n such that only compulsory misses occur.

   (b) Consider a direct-mapped cache, determine the miss rate when $n = 8$.

   Solution: Consider the for-loop over v:

   For the first two iterations, all of A and the first two rows of O fit into cache leading to only compulsory misses and A being still completely contained in cache afterwards.

   The third iteration will cause the first half of A to be evicted completely (because the addresses of the elements of O are mapped to the same cache positions). The block size is going to be $2^2 = 4$, leading to each block taking up half a cache line. Consequently, due to the access pattern, none of these evicted elements will be read back directly after the first iteration of the for-loop over b (A is accessed before the conflicting element of B). For the first iteration, however, the cache line will be read again 3 times from A and 3 times from B (because the positions of A and O fall into the same line).

   The fourth iteration will cause the second half of A to be evicted completely. Some of the elements will already be read back. We also have 7 non-compulsary misses for A and O due to a collision at the tail of the row of O (block size is 8). However, the remainder will be read back during the fifth iteration anyways.

Apart from these misses, no further misses are caused by the fifth and sixth iteration.

The seventh iteration will cause the first half of A to be evicted again and will also contribute non-compulsory misses during its first iteration of the for-loop over b. The block size is $2^6 = 64$, so 56 (7 per cache line) misses are going to occur from A and B each. At the end of this iteration, only the second half of A is going to be in cache.

In the eighth iteration, the phenomenon that we observed with the conflicting addresses is going to occur between the second (instead of the first) block and the output array leading to a total of 112 misses to A and O each (block size is $2^7 = 128$). Also the lower half of A will be read again completely.

So in total the described phenomenon causes $2 \cdot (3 + 7 + 56 + 112) = 356$ non-compulsory misses. Additionaly, 32 misses occur because A has to be read back once completely (after iterations three and four) and 16 misses occur when the first half of A is read back during the eighth iteration. Consequently, $356 + 32 + 16 = 404$ non-compulsory misses occur. There are also $\frac{2^8+8\cdot 2^7}{8} = 160$ compulsory misses, resulting in 564 total misses. On the other hand, there are $3 \cdot 8 \cdot 2^7 = 3072$ accesses, resulting in a miss rate of around 0.1836.

(c) Consider a 2-way associative cache with a LRU replacement policy, determine the miss rate when $n = 8$. The size of the cache does not change.

Solution: A is read completely in each iteration of the for-loop over v. So during its first iteration, A will be read into the first set and O will be read into the lower half of the second set. In subsequent iterations, the elements of O will have been less recently accessed than the ones of A, leading to the eviction of the (no longer used) elements of O and keeping A in cache. So only the 160 compulsory misses occur while the number of accesses stays at 3072 (see (b)). So the miss-rate is around 0.0521.

2. *Cache Mechanics (25 pts)*

Consider the following code executed on a machine with a direct-mapped write-back/write-allocate cache with blocks of size 16 bytes and a total capacity of 128 bytes. Assume that memory accesses occur in exactly the order that they appear, and that all scalar variables (result, i, j, x0, y0) remain in registers and do not cause cache misses.

Array x is cache-aligned (first element goes into first cache block) and the first element of y is immediately after the last element of x in memory.

Array x is a matrix of size 50×2, and array y is a matrix of size 2×2. `sizeof(float)` $= 4$ bytes.

```c
1  struct data_t {
2    float a;
3    float b;
4  };
5
6  #define N 50
7  #define M 2
8
9  float comp(data_t* x, data_t* y) {
10   float result = 0;
11   for(int i = 0; i < N; i+=5){
12     for(int j = 0; j < M; j++){
13       float x0 = x[i*M+j].a;
14       float y0 = y[(i%M)*M+j].b;
15       result += x0 * y0;
16     }
17     // Draw state of cache and write hit miss pattern here for i == 15
18   }
19   // Draw state of cache and write hit miss pattern here
20   return result;
21 }
```

Show your work. In case you use a script to compute any of your answers, hand in the script as well, use only Python.

(a) Considering the cache misses of the computation, compute the following:

i. determine the miss/hit pattern for x and y (something like x: MMHH..., y: MMMH...) at line 16, for $i = 15$. Consider all the accesses happening from $i = 0$ to $i = 15$ included (i=0, 5, 10, 15).

ii. draw the state of the cache at line 16, for $i = 15$.

iii. draw the state of the cache at the end of the computation.

Solution:

Miss/hit pattern:
x: MHMHMMMH
y: MHMHMMHH

State of the cache:

$i = 15$

Set	0
0	$x_0.a, x_0.b, x_1.a, x_1.b$
1	
2	$y_0.a, y_0.b, y_1.a, y_1.b$
3	$y_2.a, y_2.b, y_3.a, y_3.b$
4	
5	$x_{10}.a, x_{10}.b, x_{11}.a, x_{11}.b$
6	
7	$x_{30}.a, x_{30}.b, x_{31}.a, x_{31}.b$

end

Set	0
0	$\mathrm{x}_{80}.a, x_{80}.b, x_{81}.a, x_{81}.b$
1	$\mathrm{x}_{50}.a, x_{50}.b, x_{51}.a, x_{51}.b$
2	$\mathrm{y}_{0}.a, y_{0}.b, y_{1}.a, y_{1}.b$
3	$\mathrm{y}_{2}.a, y_{2}.b, y_{3}.a, y_{3}.b$
4	$\mathrm{x}_{40}.a, x_{40}.b, x_{41}.a, x_{41}.b$
5	$\mathrm{x}_{90}.a, x_{90}.b, x_{91}.a, x_{91}.b$
6	$\mathrm{x}_{60}.a, x_{60}.b, x_{61}.a, x_{61}.b$
7	$\mathrm{x}_{30}.a, x_{30}.b, x_{31}.a, x_{31}.b$

(b) Repeat the previous task assuming now that the cache is 4-way set associative and uses a LRU replacement policy. The cache size and block size stay the same.

Solution:

Miss/hit pattern:
x: MHMHMHMH
y: MHMHHHHH

State of the cache:

$i = 15$

Set	0	1	2	3
0	$\mathrm{y}_{0}.a, y_{0}.b, y_{1}.a, y_{1}.b$	$\mathrm{x}_{20}.a, x_{20}.b, x_{21}.a, x_{21}.b$	$\mathrm{x}_{0}.a, x_{0}.b, x_{1}.a, x_{1}.b$	
1	$\mathrm{y}_{2}.a, y_{2}.b, y_{3}.a, y_{3}.b$	$\mathrm{x}_{30}.a, x_{30}.b, x_{31}.a, x_{31}.b$	$\mathrm{x}_{10}.a, x_{10}.b, x_{11}.a, x_{11}.b$	

end

Set	0	1	2	3
0	$\mathrm{y}_{0}.a, y_{0}.b, y_{1}.a, y_{1}.b$	$\mathrm{x}_{80}.a, x_{80}.b, x_{81}.a, x_{81}.b$	$\mathrm{x}_{60}.a, x_{60}.b, x_{61}.a, x_{61}.b$	$\mathrm{x}_{40}.a, x_{40}.b, x_{41}.a, x_{41}.b$
1	$\mathrm{y}_{2}.a, y_{2}.b, y_{3}.a, y_{3}.b$	$\mathrm{x}_{90}.a, x_{90}.b, x_{91}.a, x_{91}.b$	$\mathrm{x}_{70}.a, x_{70}.b, x_{71}.a, x_{71}.b$	$\mathrm{x}_{50}.a, x_{50}.b, x_{51}.a, x_{51}.b$

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

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

  Each can issue 1 instruction per cycle and each instruction has a latency of 1.
- One write-back/write-allocate cache with blocks of size 64 bytes.
- Read bandwidth from the main memory is 9.6 GB/s.
- The processor's frequency is 3 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.

(b) Consider the following functions. For each, assume that vector instructions are not used, and *derive hard upper bounds on its operational intensity* (consider only **loads**) based on its **instruction mix** and **compulsory misses**. Ignore the effects of aliasing and assume that no optimizations that change operational intensity are performed (the computation stays as is). All arrays are cache-aligned (first element goes into first cache set) and don't overlap in memory. You can further assume that all variables stay in registers.

```
// x, y, z are all of size n
void comp1(double *x, double *y, double *z, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++){
      z[i] = z[i] + x[j] * y[j >> 4];
    }
  }
}

// A, B, C are all of size n * n
void comp2(double *A, double *B, double *C, double alpha, double beta, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++){
      C[i*n+j] = C[i*n+j] + A[i*n+j]*B[i*n+j]*alpha + beta;
}}}
```

(c) For each computation, derive the maximum performance for $n = 5$, $n = 20$, $n = 40$. Assume you write code that attains the performance and operational intensity bounds, and **add the performance to the roofline plot** (there should be six dots, one for each n for the two functions).

(d) For each computation, what is the maximum speedup you could achieve by parallelizing it with vector intrinsics? Add the points corresponding to $n = 5$, $n = 20$, $n = 40$ on the roofline plot.

(e) Consider now this computation. Assume that n^2 is much larger than the last level cache. Meaning each function call needs to reload all the input matrices.

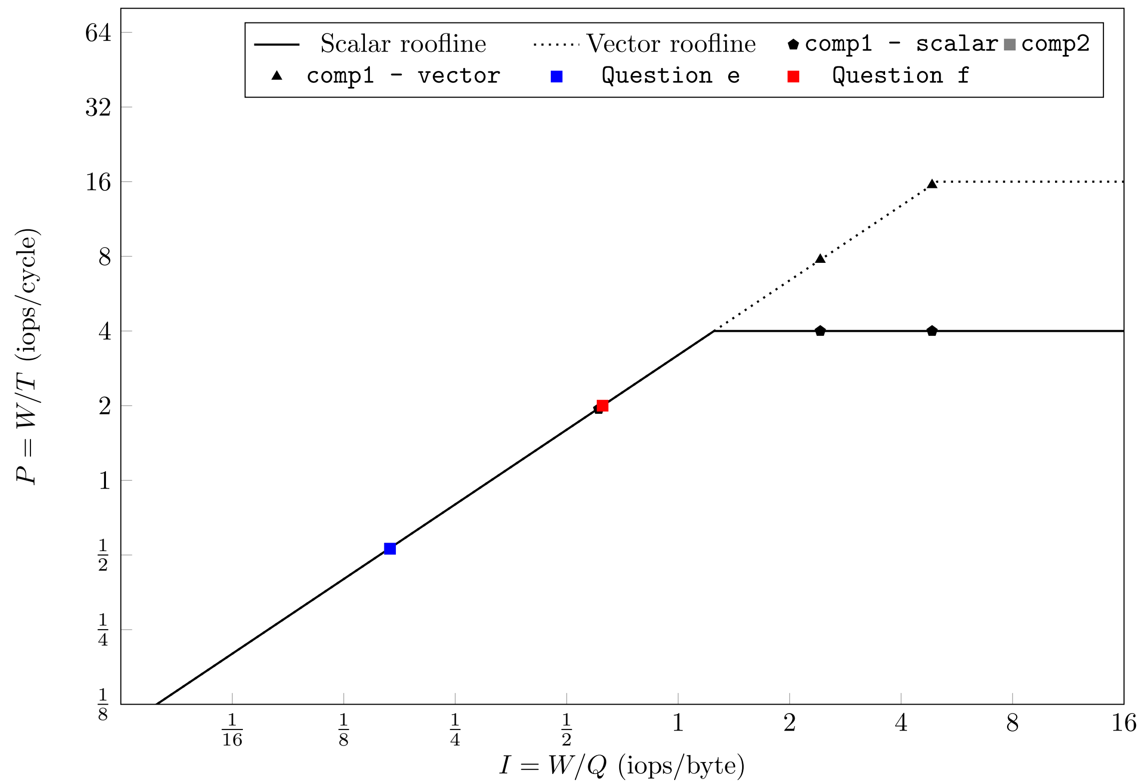
```
double *A, *B, *C, *D;  // All these are n * n
double a0, a1, a2, a3, a4;
double b0, b1, b2, b3, b4;

// Initializing all the inputs

comp2(A, B, C, a0, b0, n);
comp2(B, D, C, a1, b1, n);
comp2(B, D, C, a2, b2, n);
comp2(D, D, C, a3, b3, n);
comp2(A, D, C, a4, b4, n);
```

Considering the bound found in (c), derive a lower bound on the runtime of the code for $n = 50$. You are not allowed to change the code.

(f) Now, assume you are allowed to change the code above, how would you optimize it? The result in the end needs to remain the same, i.e $C = a0 \cdot AB + b0 + \ldots a4 \cdot AD + b4$. Explain the optimization and give the new bound on the operational intensity, and on the runtime for $n = 50$. Add the point on the roofline plot for the optimized code.

Solution:

Roofline plot:



(a) $\beta = \frac{9.6}{3} = 3.2$ bytes/cycle, Max performance is obtained when doing 2 FMAs at the same cycle $\implies$ 4 flops. Memory bound threshold, $\beta I = \pi \implies I = 4/3.2 = 1.25$, $\beta I = \pi_v \implies I = 16/3.2 = 5$

(b) Considering only compulsory misses.
For comp1: $Q(n) \geq 8(2n + \frac{n}{16}) = 16n + \frac{n}{2} = \frac{33n}{2}$, $W(n) = 2n^2$, $I(n) \leq \frac{4n^2}{33n} = \frac{4n}{33}$
For comp2 : $Q(n) \geq 8 \cdot 3n^2$, $W(n) = 4n^2$, $I(n) = \frac{4n^2}{24n^2} = 1/6$.

(c) For comp1:
  i. n = 5: $I(5) \approx 0.61 < 1.25$. Memory bound. $\beta I(5) \approx 1.94$
  ii. n = 20: $I(20) \approx 2.42 > 1.25$. Compute bound. Peak performance 4
  iii. n = 40: $I(40) \approx 4.85 > 1.25$. Compute bound. Peak performance 4

For comp2, the operational intensity is constant. Therefore, the performance is bound by 0.53 flops/cycles.

(d) For comp1:
  i. n = 5: $I(5) \approx 0.61 < 5$. Memory bound. $\beta I(5) \approx 1.94$. Speedup is 1
  ii. n = 20: $I(20) \approx 2.42 < 5$. Memory bound. $\beta I(20) \approx 7.76$. Speedup is $\frac{7.76}{4} = 1.94$
  iii. n = 40: $I(40) \approx 4.85 < 5$. Memory bound. $\beta I(40) \approx 15.52$. Speedup is $\frac{15.52}{4} = 3.88$

Comp2 cannot be improved. The points are on the roofline plot.

(e) Since comp2 is memory bound, and we need to load all three matrices at every function call, the final runtime will be $T > \frac{5 \cdot 4n^2}{0.5}$, for $n = 50$ we have $T > 100k$ cycles.

(f) Now we can change the code, since all matrices are of the same size we can easily merge the loops. This reduces $Q(n)$ significantly from $5 \cdot 24n^2$ to $32n^2$. $W(n) = 20n^2$, $I(n) = 20/32$, max performance is now 2 flops/cycle. Therefore, $T > 25k$ cycles, $4\times$ increase in performance (decrease in runtime).

4. *Cache Miss Analysis (20 pts)*

Consider the following computation that performs a blocked matrix multiplication

$$C = C + A \cdot B,$$

for square matrices A, B, and C of size $n \times n$ using a tiled *i-p-j* loop order:

```c
void blocked_mmm(double *A, double *B, double *C, int n, int b) {
  for (int i = 0; i < n; i += b) {
    for (int j = 0; j < n; j += b) {
      for (int p = 0; p < n; p += b) {
        for (int ii = i; ii < i + b; ii++) {
          for (int pp = p; pp < p + b; pp++) {
            double temp = A[ii*n + pp];
            for (int jj = j; jj < j + b; jj++) {
              C[ii*n + jj] += temp * B[pp*n + jj];
            }
          }
        }
      }
    }
  }
}
```

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 γ doubles and with a LRU replacement policy. Assume that

- n is divisible by b,
- b is a multiple of 8,
- cold caches,
- all matrices are cache-aligned.

Justify your answers. In case you use a script to compute any of your answers, hand in the script as well, use only Python.

(a) Derive, as precisely as possible, an exact expression (in terms of n and b) for the total number of cache misses incurred for each matrix. Assume that, within each matrix block of size $b \times b$, each cache line is loaded exactly once (there are only compulsory misses).

Solution: Each $b \times b$ block of A is stored in row-major order; each row needs $\frac{b}{8}$ cache lines, so one block incurs $\frac{b^2}{8}$ misses. With $\frac{n^2}{b^2}$ blocks, each matrix has $\frac{n^2}{8}$ misses.

(b) Give the exact total number of cache misses for the entire computation.

Solution: Total misses $= \frac{n^2}{8}$ (for A) $+ \frac{n^2}{8}$ (for B) $+ \frac{n^2}{8}$ (for C) $= \frac{3n^2}{8}$.

(c) For each matrix, specify which type(s) of locality (spatial and/or temporal) are exploited by the blocked algorithm.

Solution:

- A: spatial (contiguous rows) and temporal (reuse within block).
- B: spatial (contiguous rows) and temporal (reuse within block).
- C: spatial (row-wise updates) and temporal (accumulation within block).

(d) Determine the minimum cache capacity $\gamma_{\min}$ (in doubles) required so that the computation suffers only compulsory misses.

Solution: There must be space for all three blocks, therefore $\gamma_{\min} = 3b^2$ doubles.