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.

```
1  void qmc(bool* A, bool* O, int n){
2    for (int v = 0; v < n; v++){
3      unsigned int block_size = 1U << v;
4      unsigned int block_count = 1U << (n - v - 1);
5      bool* O_v = &O[((long) v) << (n - 1)];
6      for (int b = 0; b < block_count; b++) {
7          bool* A_b =  &A[2 * b * block_size];
8          for (int p = 0; p < block_size; p++) {
9              O_v[b * block_size + p] = A_b[p] && A_b[block_size + p];
10         }
11     }
12   }
13 }
```

   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.

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

   (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.

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.

```
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.

(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.

Example. The following example shows how we expect you to draw the cache. The example shows an initially empty cache with $(S, E, B) = (3, 2, 8)$ after $x[0].a$ was accessed. Note that this cache is different from the one specified in the exercise.

State of the cache after accessing $x[0].a$:

Set	0	1
0	$x_0.a$, $x_0.b$	
1		
2		

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.

```
1  // x, y, z are all of size n
2  void comp1(double *x, double *y, double *z, int n) {
3    for (int i = 0; i < n; i++) {
4      for (int j = 0; j < n; j++){
5        z[i] = z[i] + x[j] * y[j >> 4];
6      }
7    }
8  }
9
10 // A, B, C are all of size n * n
11 void comp2(double *A, double *B, double *C, double alpha, double beta, int n) {
12   for (int i = 0; i < n; i++) {
13     for (int j = 0; j < n; j++){
14       C[i*n+j] = C[i*n+j] + A[i*n+j]*B[i*n+j]*alpha + beta;
15 }}}
```

   (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.

```
1  double *A, *B, *C, *D;  // All these are n * n
2  double a0, a1, a2, a3, a4;
3  double b0, b1, b2, b3, b4;
4
5  // Initializing all the inputs
6
7  comp2(A, B, C, a0, b0, n);
8  comp2(B, D, C, a1, b1, n);
9  comp2(B, D, C, a2, b2, n);
10 comp2(D, D, C, a3, b3, n);
11 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.

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:

```
1  void blocked_mmm(double *A, double *B, double *C, int n, int b) {
2    for (int i = 0; i < n; i += b) {
3      for (int j = 0; j < n; j += b) {
4        for (int p = 0; p < n; p += b) {
5          for (int ii = i; ii < i + b; ii++) {
6            for (int pp = p; pp < p + b; pp++) {
7              double temp = A[ii*n + pp];
8              for (int jj = j; jj < j + b; jj++) {
9                C[ii*n + jj] += temp * B[pp*n + jj];
10             }
11           }
12         }
13       }
14     }
15   }
16 }
```

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).

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

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

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