Last name, first name: _____

Student number: _____

263-0007-00L: Advanced Systems Lab
ETH Computer Science, Spring 2025
Midterm Exam
Wednesday, April 16, 2025

Instructions

- Write your full name and student number on the front.
- Make sure that your exam is not missing any sheets.
- No extra sheets are allowed.
- The exam has a maximum score of 100 points.
- No books, notes, laptops, cell phones, or other electronic devices are allowed.
- Dedicated calculators are allowed, i.e not the ones in your mobile phones.

Problem 1 (21=4+2+2+2+3+3+5)	
Problem 2 (16=2+1+4+3+6)	
Problem 3 (14=6+4+4)	
Problem 4 (16=2+1+4+3+3+3)	
Problem 5 (15=1+1+6+4+3)	
Problem 6 (18=3+3+3+3+3+3)	
Total (100)	

Problem 1: Sampler (21=4+2+2+2+3+3+5)

Be brief in your answers, no need to show derivations unless indicated otherwise.

1. Reply to the following. Be precise.

   (a) What is the latency of an instruction?

   Solution: It is the number of cycles that occur between issuing an instruction and having the result in a register.

   (b) What is the throughput of an instruction?

   Solution: It is the number of independent instruction of the same type that can be issued at the same cycle.

   (c) What is the difference between a port and an execution unit of a CPU?

   Solution: Each port has multiple execution units. A port can issue an instruction per cycle. When an instruction is issued, the execution unit for that instructions is busy for $\lceil 1/throughput \rceil$ cycles.

2. Consider a CPU on which a floating point addition has inverse throughput $\frac{1}{3}$ and latency 2. How many independent additions would you need at least to expect maximal performance when executed?

   Solution: 6

3. Give two differences between the Skylake x86 processor and the Apple M-series Arm processors seen during the lectures.

   Solution: Multiple options are possible. Example of differences:

   - More execution ports
   - Vector size is only 128 bits
   - Asymmetric architecture (efficiency and performance cores)
   - Apple uses Neon, Skylake uses AVX/AVX2/SSE...

4. What is the output of this function?

```
1 __m256d foo() {
2     __m256d x = _mm256_set_pd(16.0, 4.0, 2.0, 5.0);
3     __m256d y = _mm256_set_pd(32.0, 7.0, 3.0, 1.0);
4     return _mm256_blend_pd(x, y, 0b0110); // last is the bit string 0110
5 }
```

   Solution: (5, 3, 7, 16)

5. How many FMA operations are required to compute ABC for $A \in \mathbb{R}^{n\times m}, B \in \mathbb{R}^{m\times k}, C \in \mathbb{R}^{k\times l}$? Assume the naive triple loop implementation for matrix multiplication. **Solution:** We can either compute $(AB)C$ or $A(BC)$. The optimal choice depends on m and k. Thus, we require

$$\min(nmk + nkl, nml + mkl) = \min(nk \cdot (m + l), ml \cdot (n + k))$$

flops.

6. Given a fixed computation and a direct mapped cache of size γ bytes. Is it true that doubling the size of the cache (i.e., to 2γ) would always reduce number of cache misses? If yes, explain why. If no, give a counterexample in words (kind of computation + how the cache is doubled in size).

   Solution: This is not always true. Consider a cache of size 32 B with block size 8B. If we load data from an array of doubles with stride 8, we have conflict misses in the first set of the cache. If we double the cache size by doubling the amout of sets (going from 4 sets to 8 sets), we still incur in the same number of misses.

7. Given the following linear transform algorithm:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix},$$

   where c, s are constants. Additionally, $s \neq 0$ and $c \neq 0$. How would you implement this using only FMAs? How many FMAs would you need? *Hint:* modify the algorithm into a mathematically equivalent form. Show then this form or just write out the needed FMAs.

   Solution: Rewrite the operation as such

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & c & 0 \\ 0 & 1 & 0 & s \\ 1 & 0 & -c & 0 \\ 0 & 1 & 0 & -s \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -s/c \\ 0 & 0 & 1 & c/s \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

   This can be implemented with 6 FMAs.

$$\begin{aligned} t_2 &= x_2 - \frac{s}{c} x_3 \\ t_3 &= x_2 + \frac{c}{s} x_3 \\ y_0 &= x_0 + c t_2 \\ y_1 &= x_1 + s t_3 \\ y_2 &= x_0 - c t_2 \\ y_3 &= x_1 - s t_3 \end{aligned}$$

Problem 2: Bounds (16=2+1+4+3+6)

Consider the following function:

```
1  void compute(double* x, double* y, double* z, int n) {
2      double C0 = 0.5, C1 = 3.14, C2 = 2.71;
3      for(int i = 0; i < n; i++){
4          double t0  = C0 + x[i];
5          double t1  = C1 + x[i];
6          double t2  = C1 + y[i];
7          double t3  = C2 + y[i];
8          double t4  = C2 * z[i];
9
10         // OP is defined in text
11         double t5  = t0 OP t1;
12         double t6  = t2 * t3;
13         double t7  = t4 + t3;
14
15         double t8 = t5 * t6;
16         double t9 = t6 * t7;
17
18         z[i] = t8 * t9;
19     }
20 }
```

Assume that the above code is executed on a computer with the following relevant latency, inverse throughput, and port information:

Instruction	Latency [cycles]	Inverse throughput [cycles/instruction]	Port(s)
`add`	2	0.5	0,1
`mult`	3	0.5	0,1
`div`	6	4	2

The processor does **not** support vector instructions. Further assume that:

1. You can ignore the latency and throughput of loads and stores, i.e., assume they have zero latency and infinite throughput,
2. The compiler does not apply any algebraic transformation: the operations are mapped to their respective assembly instructions,
3. Each instruction is mapped to its own execution unit,
4. Ignore integer operations,
5. A division counts as one floating-point operation.

Show enough detail with each answer so we understand your reasoning.

1. Determine the maximum theoretical floating-point peak performance in flops/cycle of the computer under consideration.

   Solution: We can schedule 2 additions/multiplications every cycle and one division every 4 cycles. This gives 9 flops every 4 cycles. Therefore peak performance is 2.25.

2. Determine the exact flop count $W(n)$ of the `compute` function. Assume that OP counts as one floating-point operation.

   Solution: $W(n) = 11n$

```
double t0  = C0 + x[i];
double t1  = C1 + x[i];
double t2  = C1 + y[i];
double t3  = C2 + y[i];
double t4  = C2 * z[i];

// OP is defined in text
double t5  = t0 OP t1;
double t6  = t2 * t3;
double t7  = t4 + t3;

double t8 = t5 * t6;
double t9 = t6 * t7;

z[i] = t8 * t9;
```

3. Assume that OP is a **multiplication**. Determine a lower bound (as tight as possible) for the runtime (in cycles) and an associated upper bound for the performance of the `compute` function based on the instruction mix, ignoring dependencies between instructions (i.e., don't consider latencies and assume full throughput).

   Solution: We have $5n$ additions and $6n$ multiplications. If we consider only instruction mix, we can schedule these in $5.5n$ cycles.

4. Repeat the previous task assuming now that OP is a **division**.

   Solution: We have $5n$ additions and $5n$ multiplications and n divisions. The bottleneck is given by the $10n$ additions and multiplications, giving a lower bound on the runtime of $5n$ cycles.

5. Estimate a lower bound (as tight as possible) for the number of cycles that the computation in the loop takes to complete. Now take latency, throughput and dependency information into account and assume that OP is a **division**. Draw the corresponding DAG of the computation.

   Solution: 14 cycles.

Problem 3: Operational Intensity (14=6+4+4)

Consider the following three programs `xAx`, `xUDUx` and `xLLx` that compute what is called quadratic forms. Mathematically, these programs compute the following equations:

- `xAx`: $y = x^T Ax$,
- `xUDUx`: $y = (Ux)^T D(Ux)$,
- `xLLx`: $y = (Lx)^T (Lx)$.

Here, y, x are vectors of length n, and A, U, D, L are $n \times n$-matrices. Additionally,

- D is a diagonal matrix and is stored as an array of length n,
- L is an upper triangular matrix stored as a dense $n \times n$ matrix (which thus contains the zeros of the lower triangle).

You are given the following implementations of the functions. Each function uses a temporary array t that contains only zeros of size n to store an intermediate result.

```
1  double xAx(double* A, double* x, double* t, int n) {
2      for (int i = 0; i < n; i++)
3          for (int j = 0; j < n; j++)
4              t[i] += A[i*n+j] * x[j];
5
6      double y = 0;
7      for (int k = 0; k < n; k++)
8          y += x[k] * t[k];
9      return y;
10 }
11
12 double xUDUx(double* U, double* d, double* x, double* t, int n) {
13     for (int i = 0; i < n; i++)
14         for (int j = 0; j < n; j++)
15             t[i] += U[i*n+j] * x[j];
16
17     double y = 0;
18     for (int k = 0; k < n; k++)
19         y += d[k] * t[k] * t[k];
20     return y;
21 }
22
23 double xLLx(double* L, double* x, double* t, int n) {
24     for (int i = 0; i < n; i++)
25         for (int j = i; j < n; j++) // Accessing only upper triangle!
26             t[i] += L[i*n+j] * x[j];
27
28     double y = 0;
29     for (int k = 0; k < n; k++)
30         y += t[k] * t[k];
31     return y;
32 }
```

1. Compute a hard upper bound on the operational intensity of `xAx`, `xUDUx` and `xLLx` (in flops/bytes). Assume

   - only compulsory misses,
   - cache block size of 1 double,
   - loop indices (i, j, k) and the scalar variable y remain in registers,
   - `sizeof(double) = 8`,
   - cold cache.

   Show your calculations.

   Solution:

	`xAx`	`xUDUx`	`xLLx`
$W(n)$	$2n^2 + 2n$	$2n^2 + 3n$	$2\frac{n(n+1)}{2} + 2n = n^2 + 3n$
$Q(n)$	$8(n^2 + 2n)$	$8(n^2 + 3n)$	$8(\frac{n(n+1)}{2} + 2n)$
$I(n) = \frac{W(n)}{Q(n)}$	$\frac{1}{4}\frac{n+1}{n+2}$	$\frac{1}{4}\frac{n+1.5}{n+3}$	$\frac{1}{4}\frac{n+3}{n+5}$

2. Which of the three implementations are memory-bound on a machine with a memory bandwidth of 8 bytes/cycle and a peak performance of 3 flops/cycle?

   Solution: The operational intensity of all versions is less than $frac14 < \frac{3}{8}$. So all of them are memory-bound.

3. For each of the three implementations either propose an optimization increasing performance on the machine described in (2), or argue why no straight-forward optimization exists.

   Solution: The scratch array `t` is first produced and then consumed element-by-element. Consequently, we can replace it by a local variable and merging the two loops together. This reduces Q and thus leads to a speed-up.

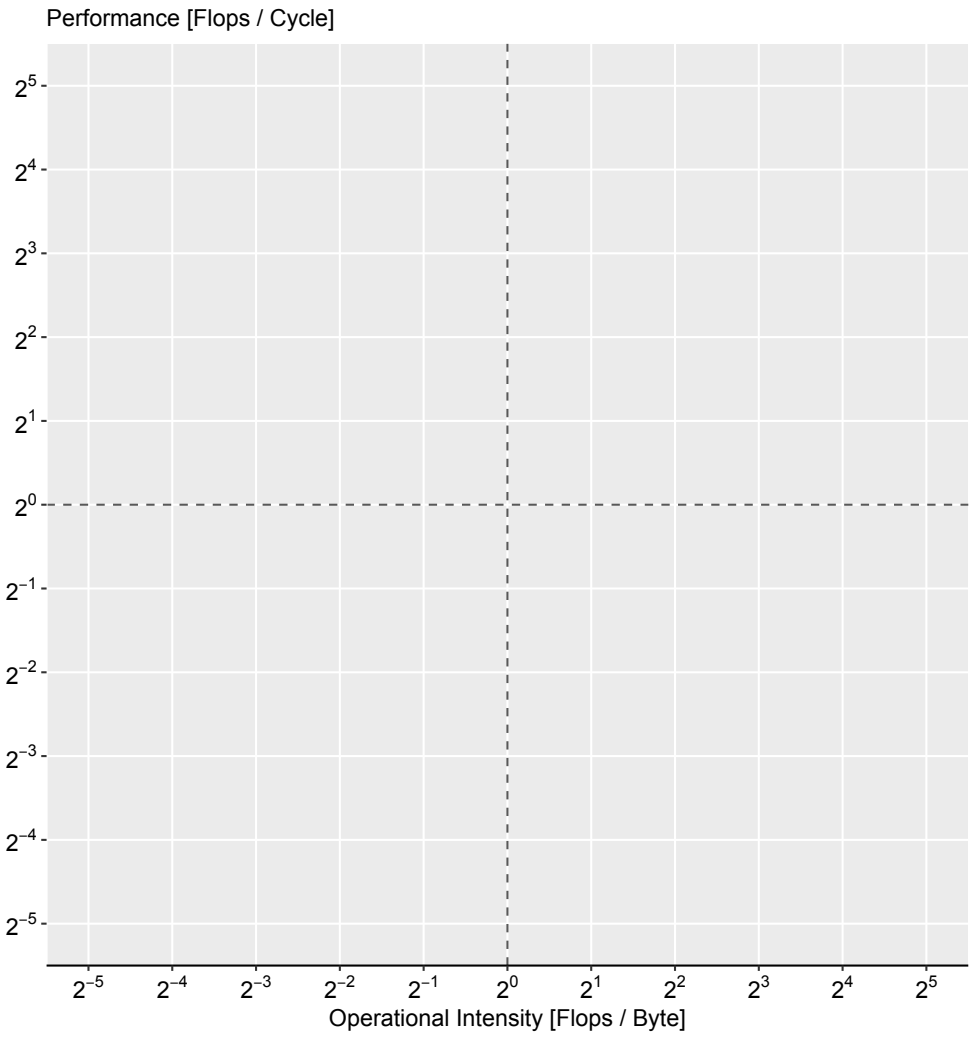
   We also accept as an answer that no significant optimization is possible, because almost all memory transfers come from the matrices and it is not possible to optimize there. Just stating that one cannot optimize is not enough.

   Also for `xUDUx` one can pull the diagonal matrix into the matrix `U` (by multiplying the rows with the square root of the diagonal entry).

Problem 4: Roofline (16=2+1+4+3+3+3)

Assume a computer with the following features:

- A CPU with the following **double-precision** floating-point ports:
  Port 1: FMA.
  Port 2: ADD, MUL.
  Port 3: MUL.
- Each of these instructions, on its own port, has a latency of 4 cycles and a throughput of 1 cycle.
- It supports 256-bit AVX operations with the same latency and throughput as the corresponding scalar operations.
- A write-back/write-allocate cache. The cache is initially cold.
- The read (memory) bandwidth is $\beta_{read} = 16$ bytes per cycle.
- `sizeof(double)=8`



Show enough detail in each answer so we can see your reasoning.

1. Draw the roofline plot for this computer into the above graph. Annotate the lines so we see your reasoning. Draw 2 horizontal rooflines, one considering AVX and one without.

   Solution:

   - $\pi_s = 4$
   - $\pi_v = 16$
   - $I'_s = 1/4$
   - $I'_v = 1$

Consider the following computation where x is an array of size $1024 \times n$, with $n > 1024$. Assume that x is cache-aligned, i.e., the first element of x maps to the start of the first cache block. Assume that x is fully loaded during the computation. For ease of notation, we write below `x[i][j]` in place of `x[i*n+j]`.

```
1  double compute(double* x, int n) {
2    double c = 0;
3    for (int i = 0; i < n; i++) {
4      for (int j = 0; j < n; j++) {
5        c += x[(i*j)%1024][i] * x[(i*j)%1024][j] +
6             x[(i*j)%1024][n - i - 1] + x[(i*j)%1024][n - j - 1];
7      }
8    }
9    return c;
10 }
```

2. What is the exact flop count $W(n)$ of the compute function?

   Solution: There are $W(n) = 4n^2$ flops.

3. Ignore the indexing computations. Based on the instruction mix (i.e., considering only throughput and ignoring dependencies), which performance is maximally achievable for this function and why? Draw an associated tighter horizontal roofline into the plot above. Assume that the FMA instruction unit is used optimally.

   Solution: Ignoring data dependencies, there are $3n^2$ additions and n^2 multiplications. This can be translated to n^2 FMAs and $2n^2$ additions. In n^2 cycles we can dispatch the FMAs and n^2 additions leaving n^2 additions for later. These can only take port 2 so we will need an additional n^2 cycles for those. The total amount of cycles will be $T(n) \geq 2n^2$ so the new upper bound for our performance is: $\pi_{new} = W(n)/T(n) = 2$ flops/cycle

4. At what operational intensity $I(n)$ does this new horizontal roofline intersect with the read memory roofline?

   Solution: $\beta I = 2 \Rightarrow I = 1/8$ flops/byte

5. Assume the cache is fully associative and compute, for the code above, an upper bound for the operational intensity $I(n)$ considering compulsory misses only.

   Solution: We have $W(n) = 4n^2$ and $Q(n) \geq 1024 \cdot 8n = 8192n$ so $I(n) \leq W(n)/Q(n) = \frac{1}{2048}n$

6. Based on $I(n)$, for which sizes of the array is the computation compute-bound? For which is it memory-bound? Consider the thighter bound found in (4).

   Solution: The turn-over point is at $I = 1/8$ so in our case we need: $\frac{1}{2048}n = \frac{1}{8} \Rightarrow n = \frac{2048}{8} = 256$. $I(n)$ Is increasing so:

   - For $n < 256$ the computation is memory-bound.
   - For $n > 256$ the computation is compute-bound.
   - For $n = 256$ it is both

Problem 5: Cache Mechanics (15=1+1+6+4+3)

Consider a machine with a direct-mapped write-back/write-allocate cache with blocks of size B=16 bytes and a total capacity of 128 bytes. Assume `sizeof(float)` = 4 bytes.

1. How many single-precision floating-point values can be stored in this cache? **Solution:** 128/4 = 32
2. How many cache sets does it have? **Solution:** There are 128/16=8 blocks in total. Cache sets = 16/4= 4.

Consider the following code. Assume

- memory accesses occur in exactly the order that they appear,
- scalar variables (result, i, j, x0, y0) remain in registers and do not cause cache misses,
- 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,
- x is a matrix of size 50×2 (of `data_t` elements), and array y is a matrix of size 2×2 (of `data_t` elements).

```
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   return result;
20 }
```

3. Determine the miss/hit pattern for x and y (something like x: MMHH..., y: MMMH...) at line 17, for $i = 15$. Consider all the accesses happening from $i = 0$ to $i = 15$ included ($i = 0, 5, 10, 15$).

   Solution: Miss/hit pattern:
   x: MHMHMMMH
   y: MHMHMMHH

4. Draw the state of the cache at line 17, for $i = 15$.

   Solution:

   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	$x_{30}.a$, $x_{30}.b$, $x_{31}.a$, $x_{31}.b$

5. Suppose the cache size can be doubled from 128 Bytes to 256 Bytes. Which of these do you expect will reduce the cache miss rate the most, doubling B, E, or S? Consider the whole computation.

   Solution: We expect the most improvement to come when doubling the associativity E, since it would allow to keep y in cache.

Problem 6: Sparse Linear Algebra (18=3+3+3+3+3+3)

Consider a sparse matrix-matrix multiplication in **double-precision** floating-point. In particular, consider the computation $C = AB + C$, where A, B, and C are $n \times n$ matrices. Additionally, A is sparse, with $6n$ nonzero elements, B and C are dense. A is stored using the CSR format seen in the lectures, where we use `uint_32t` integers for the `col_idx` and `row_start` arrays. B and C are stored as usual, linearized in memory. Assume the following:

- CPU peak performance of 4 flops/cycle (via 2 FMAs per cycle),
- No vectorization,
- `sizeof(double) = 8`,
- The load memory bandwidth is $\beta = 6$ bytes / cycle.
- The computation only incurs compulsory misses.
- A write-back/write-allocate cache. The cache is initially cold.

Answer the following.

1. Determine the total work $W(n)$ of this computation and provide an associated lower runtime bound.

   Solution: $W_{CSR}(n) = 2 * 6n * n = 12n^2, T(n) \geq 3n^2$

2. Determine the data movement $Q(n)$ (considering loads only) in bytes of this computation and provide an associated lower runtime bound.

   Solution: $Q_{CSR}(n) = 6n(8+4)+4(n+1)+2n^2*8 = 16n^2+76n+4, T(n) \geq \frac{8}{3}n^2+\frac{38}{3}n+\frac{2}{3}$

3. Based on the lower runtime bounds found in (1) and (2), is the computation memory bound or compute bound, assuming large enough n? Briefly explain.

   Solution: For large enough n it is sufficient to look at the coefficient of the highest order term n^2. Since $3 > 8/3$ the bound based on work is tighter, so compute bound.

We now change the format of the A matrix to BCSR with blocks of size 4×4, assume that n is divisible by 4 and we still have $6n$ nonzero entries. We again use `uint_32t` integers for the `b_col_idx` and `b_row_start` arrays. For the following question assume that in this BCSR we have K nonzero blocks, which then satisfies $\lceil 6n/16 \rceil \leq K \leq n^2/16$. Consider, again, only scalar code (no vector code).

4. Determine the total work $W(K, n)$ of this computation.

   Solution: $W_{BCSR}(K, n) = 2 * 16 * K * n$

5. Determine the data movement $Q(K, n)$ (considering loads only) in bytes of this computation.

   Solution: $Q_{BCSR}(K, n) = K(16*8+4)+4(n/K+1)+16n^2 = 16n^2+132K+4n/K+4$

6. Argue why this BCSR cannot improve (make lower) the tightest lower bound found for CSR.

   Solution: BCSR cannot decrease the work (but can decrease the data movement). Since CSR was compute bound, i.e., the lower bound was based on the work, BCSR cannot improve this bound.