Last name, first name: ______________________________

Student number: ______________________________

263-0007-00L: Advanced Systems Lab
ETH Computer Science, Spring 2024
Midterm Exam
Wednesday, April 24, 2024

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 (20=2+2+2+2+4+4+4)	
Problem 2 (13=2+2+4+5)	
Problem 3 (15=4+2+3+6)	
Problem 4 (14=3+2+2+2+2+3)	
Problem 5 (15=2+2+5+6)	
Problem 6 (23=1+2+2+2+6+4+2+4)	
Total (100)	

Problem 1: Sampler (20 = 2+2+2+2+4+4+4)

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

1. What does the AVX2 intrinsic `_mm256_fmadd_ps(__m256 a,__m256 b,__m256 c)` do? How many flops does it execute?

   Solution: $r = a * b + c$. 16 flops.

2. In an assembly file you see these two `VADDSD` instructions:

   (a) `vaddsd xmm0, DWORD PTR [rdx+rax*4]`

   (b) `vaddsd xmm0, xmm1.`

   Which one do you expect to have longer latency? Why?

   Solution: (a) will probably be slower because it reads one operand from memory.

3. What is the purpose of scalar replacement? Give two benefits.

   Solution:

   - Removing aliasing
   - Enable efficient register allocation by the compiler

4. Which dependencies are avoided in SSA style code?

   Solution: WAR and WAW

5. Consider a processor with an 8-way set associative cache of size 256KiB and block size 64 bytes. Give the smallest page size such that the cache look up can start concurrently with the TLB look up.

   Solution: The page size should be at least 256KiB/8 = 32KiB.

6. Suppose a matrix of doubles of size $m \times n$ is 80% sparse, meaning 80% of elements are zero. We wonder whether the CSR format can reduce the storage in this case if we can store the column and row indices using 16-bit integers. In percentage, how much do we approximately reduce or increase storage when we use CSR compared to dense storage in this case? Show your work.

   Solution: Size of dense matrix: $8mn$ bytes

   Size of CSR 80% sparse matrix: $0.2 \cdot 8mn + 2(m + 1 + 0.2mn)$

   Therefore we reduce by $\approx 8mn/0.2(10mn) = 4\times$ meaning 75% reduction.

7. Assuming a block size of 32 bytes and double types for the arrays and ignoring non-array variables (i.e., indices and accumulators), explain what kind of locality occurs in the following code snippets. Be explicit and mention the locality kind(s) for each of the occurring arrays.

(a)

```
for (int i = 1; i < N; i += 64)
    for (int j = 0; j < M - i; j += i / 2 + 1)
        A[i] = i * j;
```

Solution: Temporal locality on A. No spatial locality since we increase i by 64.

(b)

```
// Assume that N, M, L, O are large powers of 2
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    for (int k = 0; k < L; k++)
      for (int m = 0; m < O; m++)
        B[i * M * L + j * L + k] = A[j * M + i] + m ;
```

Solution: Temporal and spatial locality on B since we repeatedly access the same element, and k is increased by one. Only temporal locality on A, since we increase j before i.

Problem 2: Bounds (13 = 2+2+4+5)

Consider the following function:

```
void compute(double* x, double* y, double* z, int n) {
  double a = 0.3;
  double b = 0.1;
  double c = 0.4;
  for(int i = 0; i < n; i++){
    // OP is provided in text
    z[i] = (c * z[i]) / ((a + x[i] + y[i]) * (b OP y[i]));
  }
}
```

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

Instruction	Latency	Inverse throughput	Port(s)
	[cycles]	[cycles/instruction]	
add	1	0.33	0,1,2
mult	2	0.5	0,3
div	7	4	0

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. Ignore integer operations.
4. 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 3 additions and one multiplication every cycle. Thus, the peak performance is 4 flops/cycle.

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

   Solution: $W(n) = 6n$

3. 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). Consider the following two cases:

   (a) assume that OP is a **multiplication** operation.

   $$T(n) \geq 4n \text{ cycles}$$

   .

   $$P(n) \leq 6n/4n = 1.5 \text{ flop/cycle}$$

   (b) assume that OP is a **division** operation.

   $$T(n) \geq 8n \text{ cycles}$$

   .

   $$P(n) \leq 6n/8n = 0.75 \text{ flop/cycle}$$

4. Estimate a lower bound (as tight as possible) for the number of cycles that the computation in line 7 takes to complete. Take latency, throughput and dependency information into account and assume that OP is a **division** operation. Draw the corresponding DAG of the computation performed in line 7.

   Solution: 16 cycles.

Problem 3: Operational Intensity (15=4+2+3+6)

Consider the computation

$$y = UDx, \tag{1}$$

with x, y column vectors of doubles of length n (a power of 2), and U, D are an $n \times n$ matrices of doubles. Additionally, D is a *diagonal* matrix. Matrices are stored in *row-major* order. Consider the following two C functions, *comp1* and *comp2*, that implement (1).

```c
void comp1(double* U, double* D, double* x, double* y, double* t, int n) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      t[j] += D[i*n+j] * x[j]
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      y[j] += U[i*n+j] * t[j]
}

void comp2(double* U, double* d, double* x, double* y, int n) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      y[j] += U[i*n+j] * d[j] * x[j]
}
```

Comp1 calculates (1) as as $y = U(Dx)$, while *comp2* uses the fact that D is a diagonal matrix, so it is treated as a vector d that contains only the diagonal (nonzero) entries. Make the following assumptions:

- `sizeof(double)` $= 8$.
- A write-back/write-allocate cold cache.
- n is a multiple of the cache block of size $B = 64$ bytes.
- The machine has 2 ports. Each port can execute one add or one multiplication per cycle (no FMA, and no vector instructions).

In the derivations you can omit lower order terms (writing $\approx$ instead of $=$). Show your work.

1. Determine a hard upper bound for the operational intensities $I_1(n)$ and $I_2(n)$, expressed in [flops/byte], of the computations *comp1* and *comp2*, respectively, by considering only compulsory misses. Consider both reads andwrites.

   Solution:

   - *comp1*: $W_1(n) = 4n^2$, $Q_1(n) \approx 16n^2$, $\implies I_1(n) = 1/4$
   - *comp2*: $W_2(n) = 3n^2$, $Q_2(n) \approx 8n^2$, $\implies I_2(n) = 3/8$

2. Using the above bound on the operational intensity, for what values of the memory bandwith β, expressed in bytes/cycle, is the computation *comp1* guaranteed to be memory bound?

   Solution: *comp1* is memory bound if $I_1 < 2/\beta \implies \beta < 8$ bytes/cycle.

3. Propose an optimization for *comp2* that yields the same result in real arithmetic, but has a work $W(n) \approx 2n^2$. Explain your optimization.

   Solution: Instead of computing everything in one double loop, we can precompute $t = dx$ in n flops and compute $y = Ut$ in $2n^2$ flops. Meaning $W(n) \approx 2n^2$.

Consider now the computation *comp3* that is similar to *comp2* but performed on a submatrix U' and subvectors d', x' obtained by accessing U and d, x with a stride s that is a power of 2.

```c
void comp3(double* U, double* d, double* x, double* y, int n, int s) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j+=s)
      y[j] += U[i*n+j] * d[j] * x[j]
}
```

4. Consider only reads and ignore writes. Assume a memory bandwith $\beta = 20$ [bytes/-cycle]. Determine the smallest value of the stride s (*power of 2*) such that *comp3* is memory bound. Considering only compulsory data movement. Briefly explain your answer.

   Solution: The work is always $W(n, s) = 3n^2/s$.

   The number of bytes loaded varies depending on s. We have two cases: $s \geq 8$ and $s < 8$.

   $$Q(n, s) = \begin{cases} 8n^2 & \text{if } s < 8, \\ 64n^2/s & \text{if } s \geq 8. \end{cases}$$

   Therefore

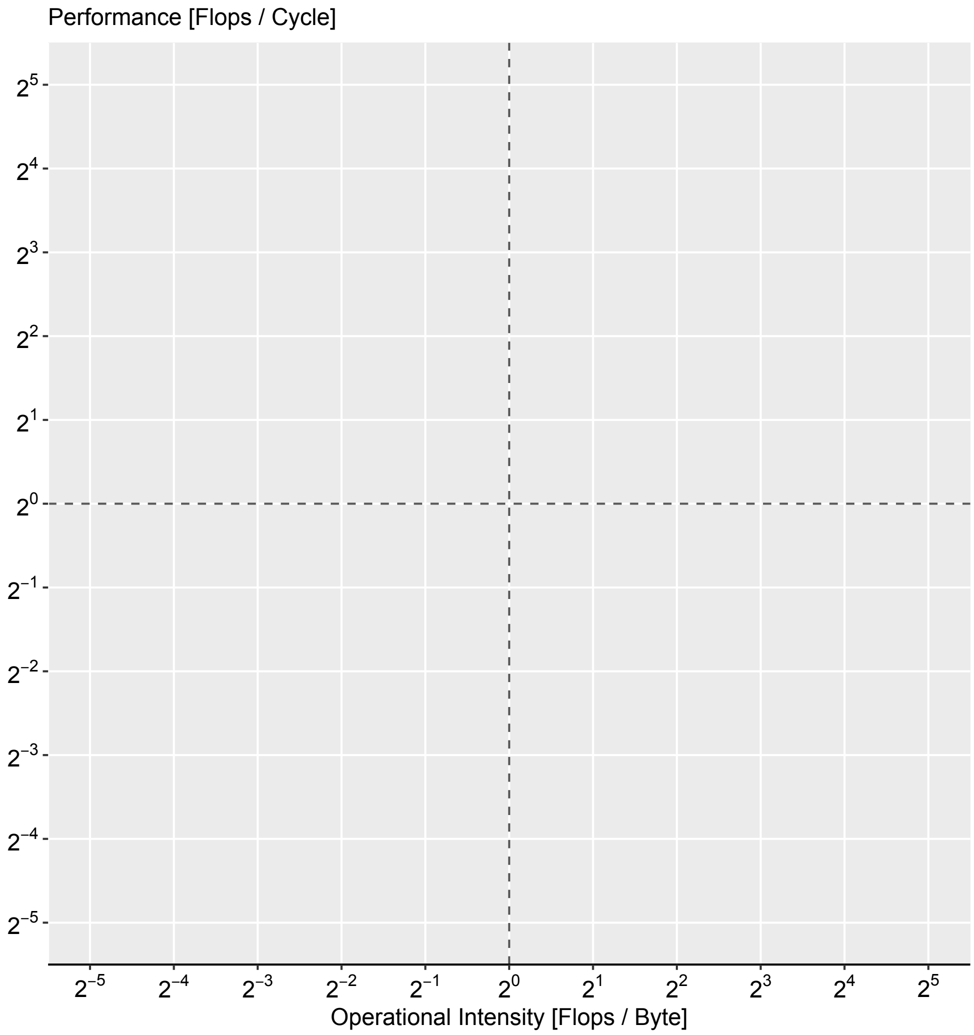
   $$I(n, s) = \begin{cases} \frac{3n^2/s}{8n^2} & \text{if } s < 8, \\ \frac{3n^2/s}{64n^2/s} & \text{if } s \geq 8. \end{cases}$$

   We analyse first $s < 8$, we have $I(n, s) \leq 3/8s$. *comp3* is memory bound if $I_3(n, s) \leq 2/20 = 1/10$. Hence we want $3/8s < 1/10 \implies 30 < 8s \implies s = 4$.

Problem 4: Roofline (14=3+2+2+2+2+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: ADD, MUL.
- Each of these instructions has a latency of 4 cycles and a throughput of 1 cycle.
- It supports 256-bit AVX operations with the same latency and inverse throughput as the corresponding scalar operations.
- A write-back/write-allocate cache. The cache is initially cold.
- The read (memory) bandwidth is $\beta_{read} = 8$ 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:

   $\pi_{scalar} = 4$ flops/cycle

   $\pi_{vector} = 16$ flops/cycle

   $\beta_{read} = 8$ bytes/cycle

   $\pi_{scalar}/\beta = 1/2$ flops/byte

   $\pi_{vector}/\beta = 2$ flops/byte

2. Consider the following code acting on an array x of size n + 8. Assume that x is cache-aligned, i.e the first element of x maps to the start of the first cache block.

```
void compute(double *x, int n){
  for(int i = 8; i < n + 8; i++){
    for(int j = 0; j < 8; j++){
      x[i] *= x[i-j];
    }
  }
}
```

   For parts (a) to (d), assume that

   i. No FMA instructions are used.

   ii. No Vector operations take place.

   (a) 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.

   Solution: The program performs multiplications only. Since we have multiplication units only in ports 2 and 3 we can perform maximum 2 flops/cycle.

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

   Solution: $\beta I = 2 \implies I = 1/4$ flops/bytes

   (c) 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. Based on this I(n), which peak performance is achievable? Consider only reads (i.e., ignore write-backs).

   Solution: We have $W(n) = 8n$, $Q(n) \geq 8n$, $I(n) \leq W/Q = 1$. The computation is compute bound and the max performance is 2 flops/cycle.

3. Now assume that the compiler can rewrite $a * b$ as $a * b + 0$ to take advantage of the port with the FMA instruction. Assuming *only scalar instructions*, what is the new maximally achievable performance? Is the computation now memory or compute-bound? Consider again instruction mix and only reads.

   Solution: We can now perform 3 multiplications per cycle. This roofline now intersects with the memory roofline at $\beta I = 3 \implies I = \frac{3}{8}$ flops/byte. Since $1 > \frac{3}{8}$ the computation is still compute-bound.

4. Keeping the previous assumptions, i.e fma, instruction mix and only reads. Consider now that vector instructions are used. Is the program memory or compute bound? What is the best achievable performance?

   Solution: With the AVX instructions we can perform $3 \cdot 4 = 12$ flops/cycle. This new roofline intersects with the memory roofline at $12/8 > 1$ flops/byte which makes the computation now memory bound.

Problem 5: Cache Mechanics (15=2+2+5+6)

1. Given a cache with block size B (in bytes), S number of sets and associativity level E (assuming that all numbers are powers of two):

   (a) How many doubles (`sizeof(double)=8` bytes) fit in that cache?

   Solution: $B * S * E/8$

   (b) How many bits are required for the address tag as a function of B, S and E with a 64bit address?

   Solution: 64 - $(log_2(S) + log_2(B))$

2. Given an array A of type double and assuming

   - A is cache-aligned
   - A[0] is mapped to the first cache set
   - Cold cache with LRU replacement policy

   Given the following sequence of memory accesses:

   A[4], A[0], A[8], A[5], A[0], A[9]

   How many cache misses occur in the following scenarios?

   (a) The cache has a block size of 16 bytes, 4 sets and no associativity (direct mapped). Draw the state of the cache after the last access A[9].

   Solution: M, M, M (evict [0,1]), H, M (evict [8,9]), M (evict [0,1]) total: 5 misses

   (b) The cache has a block size of 16 bytes, 2 sets, 2-way associativity and LRU replacement. Draw the state of the cache after the last access A[9].

   Solution: M, M, M (evict [4,5]), M (evict [0,1]), M (evict [8,9]), M (evict [4,5]), total: 6 misses

Problem 6: Cache Miss Analysis (23=1+2+2+2+6+4+2+4)

As specified in the lectures, we can specify cache architecture with 3 parameters each of which is a power of 2: B = block size (**in bytes**), S = set size, E = associativity. Consider the following code.

```
void F1(double *arr) {
    size_t offset;
    for (size_t i = 0; i < 4; i++) {
        double a0 = arr[10 + i];
        double a1 = arr[4  + i];
        arr[i] = a0 + a1;
    }
}

void F2(double *arr) {
    size_t offset;
    for (size_t i = 0; i < 4; i++) {
        offset = 4 + 2 * i;
        double a0 = arr[offset];
        double a1 = arr[offset + 1];
        arr[i] = a0 * a1;
    }
}
}
```

`sizeof(double)` = 8 bytes. Assume that array arr starts at the address 0 (and is thus cache-aligned) and a cold cache. Consider variables i, and offset to be stored in registers. Memory accesses happen in exactly the order that they appear. Answer the following. Show your work.

1. Consider the function F1 and a cache with parameters (B, S, E) = $(32, 2, 1)$, with LRU replacement policy.

   (a) Write down the sequence of indices of `arr` that are accessed by function F1.

   Solution: 10, 4, 0, 11, 5, 1, 12, 6, 2, 13, 7, 3

   (b) Write down the hit-miss pattern for function F1.

   Solution: MMM M**H**M MM**H** MM**H**

   (c) Draw the state of the cache after executing F1.

   Solution: $\begin{bmatrix} arr[0] & arr[1] & arr[2] & arr[3] \\ arr[4] & arr[5] & arr[6] & arr[7] \end{bmatrix}$

2. Consider a fully associative cache (S = 1), with LRU replacement policy. We assume that executing F2 yields the following hit-miss sequence **MHMMHHMHMMHH**. Your goal is to find parameters B (at least 8, i.e., 1 double) and E that yield this trace with a *minimal* cache size. To do so, go through the following steps.

(a) Write down the sequence of indices of `arr` that are accessed by function F2.

Solution: 4, 5, 0, 6, 7, 1, 8, 9, 2, 10, 11, 3

(b) First determine the block size B. For each of the following B= $8, 16, 32, \geq 64$, determine which cannot produce the above sequence and explain why.

Solution: Since we have hits on arr[i], we know E is at least 2. Now:

- $B = 8$ we would have 0 hits
- $B = 32$ we would have hits also on the second and fourth iteration, since we know that $E > 1$.
- $B > 64$ we would have only hits.
- $B = 16$ is the only possibility.

(c) With the block size B determined above, find the smallest associativity parameter E that yields the given hit-miss pattern.

Solution: Now that we know $B = 16$, we can deduce that we need at least two sets to store arr[offset] and arr[offset+1] (MH) and one to store arr[i].

3. For the parameters found in question 2, draw the state of the cache after executing F2. **Solution:**

Solution: $\left[arr[10] \quad arr[11]\right] \left[arr[2] \quad arr[3]\right]$

4. Suppose you can double *exactly* one of the cache parameters B, S, E. Which of these choices improves the miss rate of the function F2? Would it then be optimal or not and why?

Solution: Miss rate can be improved by increasing the block size (B) to leverage more spatial locality. With 2× larger B the trace will look like:
MHM—HHH—MHH—HHH. This solution is optimal since we have only compulsory misses. If we were to consider a bigger block size, then we would have conflict misses in the first two iterations since `offset` starts at 4.