Last name, first name: ____________________________________

Student number: ____________________________________

263-0007-00L: Advanced Systems Lab

ETH Computer Science, Spring 2021
Midterm Exam
Wednesday, April 21, 2021

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, calculators, laptops, cell phones, or other electronic devices are allowed.

Problem 1 (18 = 2+2+4+6+4)	
Problem 2 (12 = 4+4+4)	
Problem 3 (20 = 2+2+6+6+4)	
Problem 4 (20 = 2+2+2+4+4+3+3)	
Problem 5 (18 = 7+5+6)	
Problem 6 (12 = 2+2+2+2+2+2)	
Total (100)	

Problem 1: Bounds (18 = 2+2+4+6+4)

Consider the following two functions:

```c
void f1 (double *x, double *y, double *z, int n){
  double t;
  for(int i=0; i < n; i++){
    double a = x[i];
    double b = y[i];
    double c = z[i];
    z[i] = (5.0*a*a + 4.0*b*b) + 3.0*c;
  }
}

void f2 (double *x, double *y, double *z, int n){
  double t;
  for(int i=0; i < n; i++){
    double a = x[i];
    double b = y[i];
    double c = z[i];
    z[i] = (a*a)/4.0 + b*b - c*c;
  }
}
```

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

Instruction	Latency	Gap (inverse throughput)	Port
	[cycles]	[cycles/instruction]	
`add/subtract`	2	0.5	0/1
`mult`	3	1	1
`div`	5	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 assembly instructions as shown.
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 flop/cycle of the machine under consideration.

   Solution: Every four cycles the processor can schedule at most 4 additions, 4 multiplications and 1 division, resulting in a peak performance of $\frac{9}{4} = 2.25$ flops/cycle.

2. Determine the flop count $W(n)$ of `f1` and `f2`.

   Solution:
   `f1:` $W(n) = 7n$

   `f2:` $W(n) = 6n$

3. Determine for each function: a lower bound (as tight as possible) for the runtime (in cycles) and an associated upper bound for the performance of `f1` and `f2` based on the instruction mix, ignoring dependencies between instructions (i.e., don't consider latencies and assume full throughput).

   Solution:
   `f1:` Multiplications are the bottleneck. Thus, the runtime is at least $5n$ and the performance is at most $\frac{7n}{5n} = 1.4$ flops/cycle.

   `f2:` Divisions are the bottleneck. Thus, the runtime is at least $4n$ and the performance is at most $\frac{6n}{4n} = 1.5$ flops/cycle.

4. Suggest an improvement for each function, i.e. algebraic transformations, that produces the same result in real arithmetic but results in smaller lower bounds for their runtime. Again, only consider the instruction mix, ignoring dependencies. State also these new lower bounds for the runtime of `f1` and `f2`.

   Solution:
   `f1:` An option is to replace `3.0*c` with `c + c + c`. This results in $4n$ multiplications and $4n$ additions which can be executed in different ports. Thus, the runtime is $4n$ cycles for this case.

   `f2:` We can rewrite the whole computation as `0.25*a*a + (b+c)*(b-c)`. This results in $3n$ multiplications and $3n$ additions which can be executed in different ports. Thus, the runtime is $3n$ cycles for this case.

5. Estimate for the original function `f1` (i.e. not the one with improvement done in task 4) a lower bound for its runtime (as tight as possible) of one loop iteration taking latency, throughput and dependency information into account. Draw the corresponding DAG of one iteration of function `f1`.

   Solution:
   `f1:` 11 cycles.

Problem 2: Operational Intensity (12 = 4+4+4)

Consider the computation $z = Ax + Cy$ where x, y, z are column vectors of doubles of length n and A, C are $n \times n$ matrices of doubles. No temporary arrays are used in these computations. We assume a write-back/write-allocate cache with blocks of size 32 bytes and a cold cache at the start of the computation. `sizeof(double) = 8`. 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 intensity $I(n)$ [flops/byte] of the computation considering only compulsory misses. Consider both reads and writes. **Note**: The upper bound should hold for all cases, independently of the memory layout of the operands and the implementation.

   Solution:

$$W(n) = 2 \cdot 2n^2 + n \approx 4n^2$$

$$Q(n) \geq 8 \cdot (2n^2 + 4n) \approx 16n^2$$

$$I(n) \leq \frac{W(n)}{Q(n)} \approx \frac{1}{4} \text{ flops/byte}$$

2. You are given a machine which has 2 ports. Each port can execute one add or one multiplication per cycle (no FMA, and no vector instructions). For what values of the memory bandwith β (in bytes/cycle) is the computation guaranteed to be memory bound?

   Solution: The peak performance of the machine is $\pi = 2$ flops/cycle. The computation is memory bound when

$$\beta \leq \frac{\pi}{I(n)} \approx 8 \text{ bytes/cycle}$$

3. Compute now a lower bound for the operational intensity assuming that every memory access leads to a misse. Exclude read/write acesses to z in the analysis.

   Solution:

$$Q(n) \leq 32 \cdot 4n^2$$

$$I(n) \geq \frac{W(n)}{Q(n)} \approx \frac{1}{32} \text{ flops/byte}$$

Problem 3: Cache Mechanics (20 = 2+2+6+6+4)

You are given a cache with 4 sets and LRU replacement policy. Its block size is 16 bytes, and the capacity is 128 bytes. Consider the following code. `sizeof(double) = 8`.

```c
double a[40];
double x = 0;
for (int i = 0; i < 20; ++i) {
  double lhs = a[2*i];
  double rhs = a[f(i)];
  x += lhs * rhs;
}
```

Assume that array `a` starts at the memory address `0`. Variables `x`, `i`, `lhs` and `rhs` are stored in registers. Memory accesses happen in exactly the order that they appear. Answer the following. Show your work. Hint: It helps to draw the cache.

1. How many doubles fit into this cache?

   Solution: $128/8 = 16$ doubles.

2. What is the associativity of this cache?

   Solution: $e = 128/16/4 = 2$ (2-way set associative).

3. For each of the following definitions of `f(i)` do the following two things: i) determine the miss rate; ii) draw the state of the cache at the end of the computation. Show your work.

   (a) `f(i) = (2*i) % 8`:

   Solution: This run will have only compulsory misses. Due to temporal locality, the elements from `a[0]` to `a[7]`, when loaded, will always remain in the cache. Therefore, the hit-miss pattern will be 20 repetitions of MH, resulting in a 50% miss rate. The final state of the cache is:

Set	0	1
0	a_0,a_1	a_{32},a_{33}
1	a_2,a_3	a_{34},a_{35}
2	a_4,a_5	a_{36},a_{37}
3	a_6,a_7	a_{38},a_{39}

   (b) `f(i) = (2*i + 4) % 40`:

   Solution: The hit-miss pattern is MMMM followed by 18 occurrences of MH: 22 misses in total. Thus, the miss rate is $\frac{22\cdot 100}{40} = 55\%$. The final state of the cache is:

Set	0	1
0	a_{32},a_{33}	a_0, a_1
1	a_{34},a_{35}	a_2, a_3
2	a_{36},a_{37}	a_{28},a_{29}
3	a_{38},a_{39}	a_{30},a_{31}

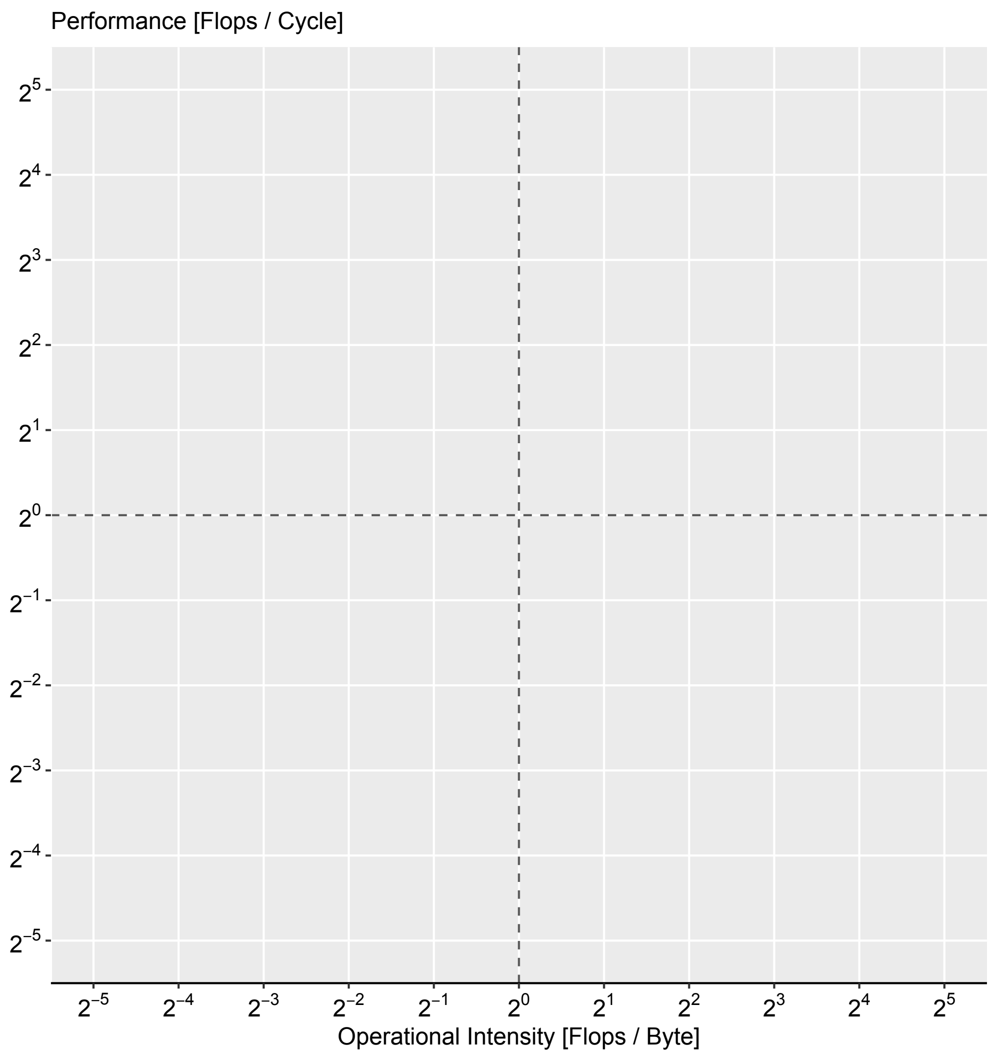
4. Give an example of `f(i)` that yields the maximal miss rate. All memory accesses have to be valid (no index out of bounds accesses). Show your reasoning.

   Solution:
   `f(i) = (2*i + 16) % 40`. The miss rate is 100%: Every access to `a` yields a miss, either compulsory or conflict. The right-hand side values of the multiplication are evicted from the cache by the time they are accessed again as the left-hand side values.

Problem 4: Roofline (20 = 2+2+2+4+4+3+3)

Assume a computer with the following features:

- A CPU that supports single precision floating point operations. The relevant port information is as follows:
  Port 1: fma, mul, add.
  Port 2: fma, mul.
- Each of these operations has a throughput of 1 per port and a latency of 4 cycles.
- It does not support any SIMD operations.
- A write-back/write-allocate cache of size 2 MiB with cache block size $B = 64$ bytes. The cache is initially cold.
- The read (memory) bandwidth is 8 floats per cycle. `sizeof(float) = 4`.



1. Draw the roofline plot for this computer into the above graph. Annotate the lines so we see your reasoning.

   Solution:

   $\pi = 4$ flops/cycle

   $\beta = 8 \cdot 4 = 32$ bytes/cycle

   $\pi/\beta = 1/8$ flops/byte

2. Consider the following computation where x, y, and z are arrays. Assume that x, y, and z are cache-aligned allocated (i.e., the address of an array maps with the first cache block):

```c
void compute(float* x,float* y,float* z, int n){
    for(int i=0; i < n; i++){
        y[i+1] = (x[i]*y[i] + y[i]) + z[i];
    }
}
```

   Based on the instruction mix (i.e. ignoring dependencies), which performance is maximally achievable for this function and why? Draw an associated tighter horizontal roofline into the plot above.

   Solution:

   If we have no data dependencies we could schedule n FMAs to one port (takes n cycles), and then schedule the remaining n ADDs in the other port. Then, we have n cycles in total. The performance is $3n$ flops in n cycles. Thus, 3 flops/cycle

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

   Solution:

   Solving $\beta \cdot I = 3$, yields $3/32$ flops/byte.

4. Based on the instruction mix **and** on data dependencies, which performance is maximally achievable for this function and why? We repeat the code for convenience:

```c
void compute(float* x,float* y,float* z, int n){
    for(int i=0; i < n; i++){
        y[i+1] = (x[i]*y[i] + y[i]) + z[i];
    }
}
```

   Solution: Due to dependencies between iterations, we can only schedule an FMA 4 cycles and an ADD of 4 cycles each iteration, thus the max performance is 3/8 flops/cycle.

5. Assume the cache is directly mapped, initially cold, and the expressions are evaluated from left to right ignoring data dependency. Assume the same array alignment as in Task 2.

   (a) Compute an upper bound (as tight as possible) for the operational intensity for large n. Consider only reads (i.e., ignore write-backs).

   Solution: We always have conflicts between x, y, z. In addition, we have extra eviction in write to y. Overall, we will load $64n$ bytes for x, $2 \cdot 64n$ bytes for y, and $64n$ bytes for z. Total memory loads are $4 \cdot 64n$ bytes. Thus $I = W/Q = 3/(4 \cdot 64) = 3/256$.

   (b) Based on this $I(n)$, which peak performance is achievable on the specified system taking into account instruction mix (i.e. the setting of Task 2)? In addition, indicate whether the computation is compute or memory bound.

   Solution: The computation is memory bound. Thus, the performance is $\beta \cdot I = 32 \cdot \frac{3}{256} = \frac{3}{8}$ flops/cycle.

6. What minimal associativity should the cache have to achieve the same performance as with a fully associative cache.

   Solution: We want to load x, y, z in cache at the same time. Thus, we need at least 3-way associativity. We accept also 4-way set associative as an answer since an associative of 3 is not common.

Problem 5: Cache Miss Analysis (18 = 7+5+6)

Consider the following computation that takes as input matrices X, Y and Z of size $n \times n$. `sizeof(double) = 8`. (X, Y, Z are not aliased).

```
 1 /* NOTE: Assume that the notation A[i][j] is transformed to A[i*n + j].
 2  *       We use the notation A[i][j] for readability only. */
 3 void f(double *X, double *Y, double *Z, int n){
 4   double t1, t2;
 5   for (int i = 1; i < n-1; i += 1) {
 6     for (int j = 1; j < n-1; j += 1){
 7       t1 = X[i][j] + 0.25*(X[i][j-1] + X[i][j+1] + X[i-1][j] + X[i+1][j]);
 8       t2 = Y[j][0] + Y[j][4] + Y[j][8];
 9       Z[i][j] = t1 + t2;
10     }
11   }
12 }
```

Assume a fully associative write-back/write-allocate cache of size γ bytes, a cache block size of 64 bytes, and only one cache. Further assume that n is much larger than γ. In the following we perform two cache miss analyses assuming an initially cold cache. **In the derivations you can omit lower order terms** (writing $\approx$ instead of $=$). Show your work.

1. Considering cache misses from both reads and writes, estimate the number of cache misses incurred by `f` as a function of n.

   Solution: In every iteration of the i loop, $\frac{3n}{8}$ blocks from X are loaded into cache, $2n$ blocks are loaded from Y, and $\frac{n}{8}$ are loaded from Z. Thus the total number of misses is $n \cdot (\frac{3n}{8} + 2n + \frac{n}{8}) = 5n^2/2$.

2. Assume that we apply loop interchange, i.e., we swap the lines 5 and 6 in the code above. Estimate again the number of cache misses incurred by `f` when this optimization is applied. The code looks now as follows:

```
void f(double *X, double *Y, double *Z, int n){
  double t1, t2;
  for (int j = 1; j < n-1; j += 1){
    for (int i = 1; i < n-1; i += 1) {
      t1 = X[i][j] + 0.25*(X[i][j-1] + X[i][j+1] + X[i-1][j] + X[i+1][j]);
      t2 = Y[j][0] + Y[j][4] + Y[j][8];
      Z[i][j] = t1 + t2;
    }
  }
}
```

   Solution: In every iteration of the j loop (the outer loop), n blocks from X are loaded into cache. Further, $2n$ additional blocks are loaded every eight iterations (when the three columns of X are not in the same block). Only two blocks from Y are loaded every iteration of j, and n blocks are loaded from Z. Thus the total number of misses is $n \cdot (n + \frac{2n}{8} + 2 + n) \approx 9n^2/4$.

3. Now we try to reduce the number of misses by blocking the computation into blocks of size $b \times b$, b a multiple of 8. This means it now has the following loop structure (we ignore clean-up code if b does not divide $n-2$):

```c
void f_blocked(double *X, double *Y, double *Z, int n){
  double t1, t2;
  for (int i = 1; i < n-1; i += b)
    for (int j = 1; j < n-1; j += b)
      for (int i1 = i; i1 < i+b; i1 += 1)
        for (int j1 = j; j1 < j+b; j1 += 1){
          t1 = X[i1][j1] + 0.25*(X[i1][j1-1] + X[i1][j1+1]
                               + X[i1-1][j1] + X[i1+1][j1]);
          t2 = Y[j1][0] + Y[j1][4] + Y[j1][8];
          Z[i1][j1] = t1 + t2;
        }
}
```

Estimate the number of cache misses incurred by `f_blocked`. In doing so, upper bound the size of b so that you achieve good cache locality. Show also this bound.

Solution: For each $b \times b$ block, we load $\frac{b^2}{8}$ cache blocks from X, $\frac{2b}{8} + 2b$ cache blocks from other iterations of X, $2b$ cache blocks from Y, and $\frac{b^2}{8}$ cache blocks from Z. Then in a block we have to load $\approx \frac{1}{4}b^2 + \frac{17}{4}b$ cache misses. There are $\approx \frac{n^2}{b^2}$ blocks. Multiplying this out, we have

$$\approx (\frac{1}{4} + \frac{17}{4b})n^2$$

cache misses in total. Assuming b large, this is approximately $n^2/4$ misses.

We can place a bound on b based on $3b/8$ cache blocks due to X and $2b$ cache blocks due to Y that must fit in cache. Since Z does not exhibit temporal locality, it is enough to keep one cache block which is a lower order term. Thus the bound is:

$$64 \cdot (3b/8 + 2b) = 152b \leq \gamma$$

$$b \leq \gamma/152$$

Problem 6: Sampler (12 = 2+2+2+2+2+2)

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

1. You are given a computation of the form:

```
for(int i = 0; i < n; ++i) {
  acc = acc OP x[i];
}
```

   where `OP` is a binary operation with gap = 0.5 cycles/issue and latency L. Assume that we unroll the loop by a factor of K and use K accumulators to improve ILP (the computation stays the same when $K = 1$). Assume that n is large. How many accumulators should you use at least to achieve the highest performance?

   Solution:
$$K = \lceil 2L \rceil$$

2. Give an example of a SIMD intrinsic that is not mapped to a single assembly instruction.

   Solution: E.g. `_mm256_set_pd`.

3. Explain what `_mm256_loadu_pd` does.

   Solution: `_mm256_loadu_pd(double* mem)` loads 256-bits (composed of 4 double-precision floating-point elements) from memory. `mem` does not necessarily has to be aligned on a 32-byte boundary.

4. Consider the following computation where x and y are arrays.

```
void compute(float* x, float* y, int n){
  for(int i=0; i < n; i++){
    y[i] += x[i]*y[i];
  }
}

...
int reps = 100000;
uint64_t t1 = time();
for(int r=0; r < reps; r++){
  compute(x, y, N);
}
double mean_duration = (time() - t1)/(double)reps;
```

   Assume $N = 128$. Would the code above provide a reliable measurement on existing modern CPUs, if we plan to measure the performance of the function for cold caches? Explain why. How can we modify the code to achieve that goal?

   Solution: The problem is that N is too small and after a first iteration the x and y will be in L1 cache (as we need only 128*8 bytes = 1KiB). To make the code reliable for measuring performance for cold cache scenario we could call compute for new untouched x and y. We can achieve it with the following code:

```c
uint64_t t1 = time();
for(int r=0; r < reps; r++){
  compute(x + r*N, y + r*N, N); // x and y have reps*N elements.
}
double mean_duration = (time() - t1)/(double)reps;
```

5. Consider a processor with a 16-way set associative cache of size 128KiB and block size 32 bytes. Propose a suitable page size (as small as possible) such that the cache look up can start concurrently with the TLB look up.

   Solution: The page size should be at least 128KiB/16 = 8KiB.

6. Provide col_idx(M) and row_start(M) of the below matrix when expressed in Compressed Sparse Row (CSR) format.

$$M = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 9 & 0 & 5 & 0 \\ 7 & 0 & 0 & 4 \end{pmatrix}$$

   Solution:

   col_idx(M) $= (1, 2, 1, 0, 2, 0, 3)$.

   row_start(M) $= (0, 2, 3, 5, 7)$.