ETH login ID:
(Please print in capital letters)

Full name: ______________________

263-2300: How to Write Fast Numerical Code
ETH Computer Science, Spring 2019
Midterm Exam
Monday, April 15, 2019

Instructions

- Write your full name and login ID 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 (17 = 4+4+4+5)	
Problem 2 (16 = 5+2+4+3+2)	
Problem 3 (24 = 2+2+6+6+8)	
Problem 4 (14 = 6+8)	
Problem 5 (15 = 2+9+4)	
Problem 6 (14 = 5+2+3+2+2)	
Total (100)	

Problem 1: Bounds (17 = 4+4+4+5)

Consider an imaginary processor with the following throughputs for different floating point operations and the five ports on which they are executed:

Instruction	Max. throughput (all ports) [instructions/cycle]	Port number
`store`	1	0
`load`	2	1 and 2
`add/subtract`	1	3
`mult`	2	3 and 4

Consider the three functions `f1`, `f2`, and `f3` below that produce equivalent outputs:

```
void f1 (const double *x, const double *y, double *res, size_t N){
  size_t i;
  for(i=0; i < N; i++){
    double a = x[i];
    double b = y[i];
    res[i] = a*a + b*b + 2*a*b;
  }
}

void f2 (const double *x, const double *y, double *res, size_t N){
  size_t i;
  for(i=0; i < N; i++){
    double a = x[i];
    double b = y[i];
    res[i] = (a+b) * (a+b);
  }
}

void f3 (const double *x, const double *y, double *res, size_t N){
  size_t i;
  for(i=0; i < N; i++){
    double a = x[i];
    double b = y[i];
    res[i] = (a-b) * (a-b) + 4*a*b;
  }
}
```

Assume the following:

1. The working set of `f1`, `f2`, and `f3` fits L1 cache and is already loaded into L1.
2. No algebraic compiler transformations are applied: the operations are mapped to assembly instructions as shown.
3. Ignore integer operations.

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

1. 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`, `f2`, and `f3` based on the instruction mix (i.e, ignore dependencies between operations).

   `f1`: Each iteration has 4 mults and 2 adds. There are 6 flops and the processor can execute 2 flops per cycle (either an add and a mult or 2 mults).

   Runtime lower bound: $3N$ cycles.
   Performance upper bound: 2 flops/cycle.

   `f2`: Each iteration has 2 adds and a mult. The mult can always execute on port 4 while the adds execute on port 3. Then a performance lower bound can be based entirely on the adds.

   Runtime lower bound: $2N$ cycles.
   Performance upper bound: 1.5 flops/cycle.

   `f3`: Each iteration has 3 adds and 3 mults. Assuming no dependencies, the processor can always execute one add on port 3 and one mult on port 4 every cycle.

   Runtime lower bound: $3N$ cycles.
   Performance upper bound: 2 flops/cycle.

2. Now we move the multiplier at port 3 to a new port numbered 5. Does this change any of the bounds? Explain.

   Yes. Now, every cycle 2 mults can be executed using ports 4 and 5 and 1 add can be executed on port 1, so the processor can now execute 3 flops/cycle (instead of 2). `f1` now has a runtime lower bound of $2N$ cycles and a performance upper bound of 3 flops per cycle. The performance and runtime bounds for `f2` and `f3` remain the same because both computations were already bounded by the number of adds.

Problem 2: Operational Intensity (16 = 5+2+4+3+2)

In the following computations, x, y, z are vectors of doubles of length n and A, B, X, Y are $n \times n$ matrices of doubles. No temporary arrays are used in these computations. We assume a write-back/write-allocate cache and a cold cache (of size γ bytes) 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. (a) Determine an upper bound for the operational intensity $I(n)$ of the computation $y = Ax + Bz + y$ considering only compulsory data movement.

   Solution:

$$W(n) \approx 4n^2 \text{ flops}$$
$$Q(n) \approx \text{sizeof(double)} \cdot 2n^2 = 16n^2 \text{ bytes}$$
$$I(n) \leq W(n)/Q(n) \approx 1/4 \text{ flops/byte}$$

   (b) Assume a processor with a peak performance $\pi = 4$ flops/cycle. What is the least memory bandwidth such that the computation could become compute bound?

   Solution: It is compute bound at:

$$\frac{4n^2}{\pi} \geq \frac{16n^2}{\beta}$$
$$\beta \geq 16 \text{ bytes/cycle}$$

2. (a) Determine an upper bound for the operational intensity $I(n)$ of the computation $Y = AX - XB$ considering only compulsory data movement.

   Solution:

$$W(n) \approx 4n^3 \text{ flops}$$
$$Q(n) \approx \text{sizeof(double)} \cdot 4n^2 = 32n^2 \text{ bytes}$$
$$I(n) \leq W(n)/Q(n) \approx n/8 \text{ flops/byte}$$

   Note: Since the question doesn't specify whether to consider write-backs or not, the answer $Q(n) \approx 40n^2$ and $I(n) \leq n/10$ is also considered correct.

(b) For which sizes n would you expect $I(n)$ to be roughly accurate?

Solution: We can expect $I(n)$ to be roughly accurate if all four matrices fit in cache.

$$\text{sizeof(double)} \cdot 4n^2 \leq \gamma$$

$$n \leq \sqrt{\frac{\gamma}{32}}$$

(c) Would the operational intensity of the computation be different on a write-through/no-write-allocate cache? If no, explain. If yes, give the bound for $I(n)$ in this case.

Solution: In a write-through cache, the matrix Y must be written but it is never read. Thus, for this case the operational intensity remains the same:

$$Q(n) \approx \text{sizeof(double)} \cdot 4n^2 = 32n^2 \text{ bytes}$$

$$I(n) \leq W(n)/Q(n) \approx n/8 \text{ flops/byte}$$

Problem 3: Cache Mechanics (24 = 2+2+6+6+8)

Consider the following code. You are given a 2-way set associative write-back/write-allocate cache with LRU replacement. Its block size is 16 bytes, and the capacity is 128 bytes.

Consider the following code; n is the common length of the arrays a, b, c.

```c
void compute(double *a, double *b, double *c, int n)
{
  // first loop
  for(int i = 0; i < n; i++) {
    double x = a[i];
    double y = b[i];
    c[i] = x + y;
  }
  // Show state of cache at this point
  // second loop
  for(int i = n-1; i >= 0; i--) {
    double x = a[i];
    double y = b[i];
    c[i] = x + y;
  }
}
```

Assume a starts at memory address 0, b directly follows a in memory and c directly follows b. Memory accesses happen in exactly the order that they appear. Hint: It helps to draw the cache.

1. What is the capacity of the cache in doubles?

   Solution: 16 doubles.

2. How many sets does the cache have?

   Solution: 4 sets.

3. For each of the following values of n do the following three things: i) determine the number of hits and misses for executing the first loop; ii) draw the state of the cache after the first loop; iii) determine the number of hits and misses in the second loop. Show your work.

   (a) $n = 4$?

   Solution:

   (i) The first loop has 6 hits and 6 misses.

   (ii) The state of the cache after the first loop is as follows. Italics signify that the quantity is LRU in that set.

Set	0	1
0	*a0,a1*	c0,c1
1	*a2,a3*	c2,c3
2	b0,b1	
3	b2,b3	

   (iii) The second loop has 12 hits and 0 misses.

   (b) $n = 8$?

   Solution:

   (i) The first loop has 0 hits and 24 misses.

   (ii) The state of the cache after the first loop is as follows. Italics signify that the quantity is LRU in that set.

Set	0	1
0	*b0,b1*	c0,c1
1	*b2,b3*	c2,c3
2	*b4,b5*	c4,c5
3	*b6,b7*	c6,c7

   (iii) The second loop has 0 hits and 24 misses.

   (c) $n = 12$?

   Solution:

   (i) The first loop has 18 hits and 18 misses.

   (ii) The state of the cache after the first loop is as follows. Italics signify that the quantity is LRU in that set.

Set	0	1
0	c8,c9	*a8,a9*
1	c10,c11	*a10,a11*
2	*c4,c5*	b8,b9
3	*c6,c7*	b10,b11

   (iii) The first 4 iterations of the second loop each have 3 hits and 0 misses. Subsequent odd-indexed iterations have 0 hits and 3 misses, and subsequent even-indexed iterations have 3 hits and 0 misses, for a total of 24 hits and 12 misses during the second loop.

Problem 4: Blocking (14 = 6+8)

Given are two arrays A and B of integers, each of length n. `sizeof(int) = 4`. The longest common subsequence (LCS) can be computed using the following dynamic program that fills an $n \times n$ table L: (A, B, L are not aliased)

```c
// assume sizeof(int) = 4
// assume all needed L[0][j] and L[i][0] are initialized to 0
void LCS(int **L, int *A, int *B, int n){
  int i, j, k;

  for (i = 1; i < n; i += 1) {
    for (j = 1; j < n; j += 1){
      L[i][j] = max(L[i-1][j], L[i][j-1], L[i-1,j-1]);
      if (A[i-1] == B[j-1]) L[i][j] = L[i][j]+1;
    }
  }
}
```

Assume a fully-associative write-back/write-allocate cache of size γ bytes, a cache block size of 32 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. Estimate the number of cache misses incurred by LCS as a function of n.

   Solution: Each iteration of the j loop, the row $L[i][j]$ and $L[i][j-1]$ must be read into cache and B must be read into cache. These loops have good spatial locality so the number of misses is $\approx 3n^2/8$.

2. 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):

```c
// assume sizeof(int) = 4
// assume all needed L[0][j] and L[i][0] are initialized to 0
void LCSblocked(int **L, int *A, int *B, int n){
  int i, j;

  for (i = 1; i < n; i += b) {
    for (j = 1; j < n; j += b){
      for (i1 = i; i1 < i+b-1; i1 += 1){
        for (j1 = j; j1 < j+b-1; j1 += 1){
          L[i1][j1] = max(L[i1-1][j1], L[i1][j1-1], L[i1-1,j1-1]);
          if (A[i1-1] == B[j1-1]) L[i1][j1] = L[i1][j1]+1;
      }
    }
  }
}
```

Estimate the number of cache misses incurred by LCSblocked. 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 access b elements of A, b elements of B, $b \times b$ elements of the current block of L, and $2b - 1$ elements of previous blocks of L. We have good spatial locality on all data accessed in a block except for the accesses to the column of the previous block of L. Then in a block we have $\approx 3b/8 + b + b^2/8$ cache misses. There are $\approx n^2/b^2$ blocks. Multiplying this out, we have

$$\approx n^2/8$$

cache misses, plus lower order terms.

We can place a bound on b based on the $b \times b$ block of L that must fit in cache.

$$4b^2 \leq \gamma$$

$$b \leq 1/2\sqrt{\gamma}$$

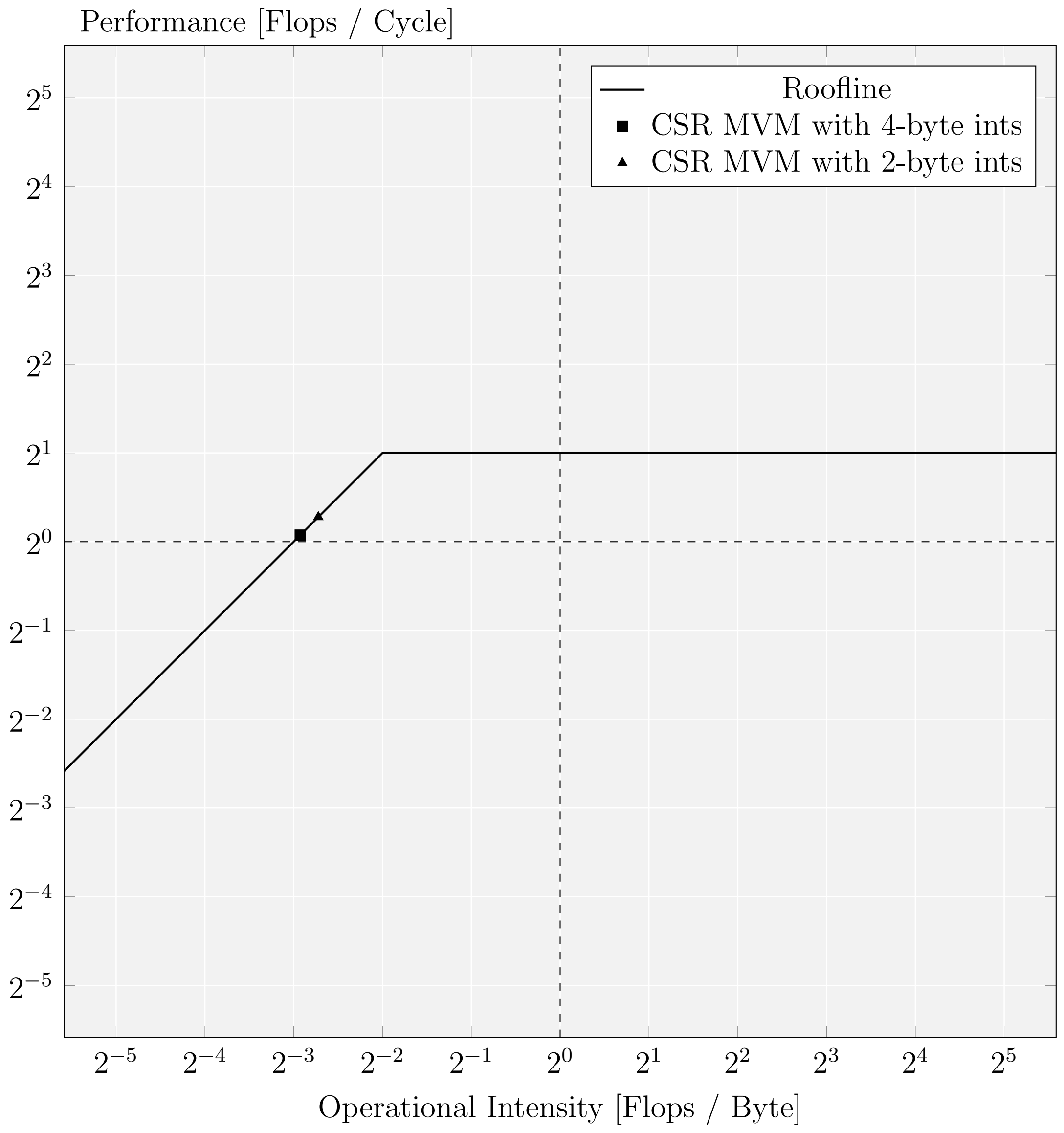
Problem 5: Roofline and Sparse MVM (15 = 2+9+4)

We consider sparse matrix-vector multiplication, in double precision floating point, of the form $y = Ax + y$, where x, y are of length n and A is $n \times n$ and sparse. The computation is done with A in CSR (compressed sparse row) format where indices are represented by (4-byte) integers. You know that A is invertible and has $5n$ many entries.

The single-core computer you run on has a memory read bandwidth of 8 bytes/cycle and a peak performance of 2 because it can execute 1 FMA/cycle.

1. Use this information to draw a roofline plot for this processor.

Solution:



2. From the above information determine a hard upper bound for the performance of the sparse MVM assuming a cold cache. Show your derivation and visualize the result in the above roofline plot.

   Solution: The amount of work done is $W = 10n$ flops. The matrix is stored as three arrays:

   - `A`, the nonzero elements, with $5n$ doubles.
   - `IA`, storing the row indices and row lengths, with $n + 1$ integers.
   - `JA`, with the column index of each nonzero element, with $5n$ integers.

   The vectors are stored in arrays containing a total of $7n$ doubles. There are a total of $7n$ doubles and $\approx 6n$ 4-byte integers. The compulsory data movement satisfies

   $$Q(n) \geq 80n \text{ bytes.}$$

   The operational intensity satisfies

   $$I \leq 1/8 \text{ flops/byte.}$$

   This is bandwidth limited according to the roofline plot. This gives a maximum performance of 1 flop/cycle.

3. Assume now that all indices in CSR are stored as 2-byte short integers. By how much can the performance bound be improved? Show your work.

   Solution: Now, instead of leading $6n$ 4-byte integers, we must load $6n$ 2-byte integers. The compulsory data movement satisfies

   $$Q(n) \geq 68n \text{ bytes.}$$

   The operational intensity satisfies

   $$I \leq 10/68 = 0.147 \text{ flops/byte.}$$

   This gives a maximum performance of $80/68 = 20/17 = 1.176$ flops / cycle, or a 17.6% improvement.

Problem 6: Sampler (14 = 5+2+3+2+2)

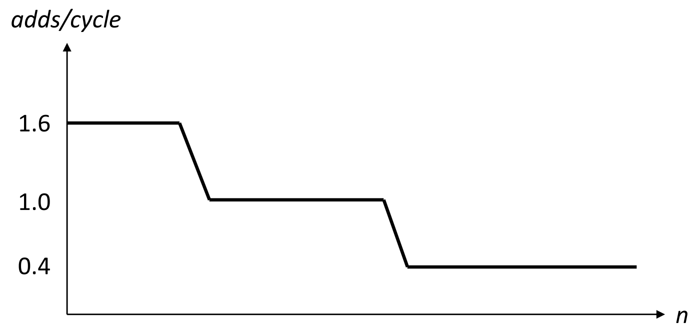
Be brief in your answers, no need to show derivations.

1. Consider the following function

```c
void vecsum(float *a, float *b, float *c, float d*, float e*, int n){
  int i;

  for (i = 0; i < n; i += 1) {
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
  }
}
```

It is run on an Intel-like single-core computer that can perform 2 additions per cycle, without SIMD vector executions. For different input sizes n (starting with very small n), warm-cache measurement yields the following performance plot.



How many caches are there?

Solution: We can see three distinct levels of the memory hierarchy — 2 levels of cache.

Estimate the read bandwidths in bytes/cycle to the caches and memory.

Solution:

L1: 8 bytes/cycle
L2: 5 bytes/cycle
DRAM: 2 bytes/cycle

2. Which types of dependencies can be resolved by renaming? How is renaming done dynamically by the processor?

Solution: Write after read (WAR) and write after write (WAW). This is accomplished by register renaming using the reorder buffer.

3. Consider the following function

```c
void vecsum(float *a, float *b, int n){
  int i;

  for (i = 0; i < n; i += 1) {
    a[i] = a[i] + b[i];
  }
}
```

compiled in isolation. Can the compiler vectorize (use SIMD vector instructions for) this function? Explain.

Solution: The compiler can vectorize this code but only if it inserts a runtime check to make sure that a and b are not aliased to the same data.

4. Earlier Pentium processors had an L1 cache only half the size of Sandybridge or Haswell. When doubling the size, Intel decided to double the associativity. Why?

Solution: An instruction uses a virtual address, but caches use physical addresses. The lowest-order 12 bits of a memory address are the address within a 4096-byte page, so the lowest-order 12 bits of a virtual address and its corresponding physical address are the same.

On early Pentium processors, the block size and the number of sets were both 64, so the lowest order 12 bits of an address were used to map an address to its (potential) location in cache. This way, the TLB lookup and the cache lookup could happen in parallel, but increasing either the number of sets or the block size would disrupt this property. Therefore, Intel decided to double the associativity.

5. With a CPU that can perform 2 FMAs per cycle with a latency of 5 cycles, how many accumulators would you use for a dot-product to (hopefully) achieve the peak performance?

Solution: In order to reach peak, 10 accumulators should be used to avoid read after write dependencies.