ETH login ID:
(Please print in capital letters)

Full name: ______________________________

263-2300: How to Write Fast Numerical Code
ETH/CS, Spring 2013
Midterm Exam
Friday, April 19, 2013

Instructions

- Make sure that your exam is not missing any sheets, then write your full name and login ID on the front.
- The exam has a maximum score of 100 points.
- No books, notes, calculators, laptops, cell phones, or other electronic devices are allowed.

Problem 1 (22 = 10 + 10 + 2)	
Problem 2 (10)	
Problem 3 (16)	
Problem 4 (16 = 6 + 6 + 4)	
Problem 5 (16 = 7 + 7 + 2)	
Problem 6 (20 = 4 + 8 + 8)	
Total (100)	

Problem 1 (22 = 10 + 10 + 2 points)

For this problem we make the following assumptions:

- All caches are fully associative, with LRU eviction policy.
- All caches are write-back/write-allocate.
- All caches are empty at the beginning of an execution (cold cache).
- The variables i, j, and k are stored in registers.
- A float is 4 bytes.

The function mmm multiplies two $N \times N$ matrices A and B storing the result in C. For simplicity, we assume that C is initialized to all zeros.

```
void mmm(float A[N][N], float B[N][N], float C[N][N]) {
  int i,j,k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

1. Consider the executions of mmm with $N = 2$ and $N = 4$ on a 64-byte cache with 4-byte cache blocks. Fill in the table below with the number of cache misses caused by accesses to each of the matrices A, B, and C, assuming that all these arrays are 16-byte aligned. Show your work or briefly explain.

N	A	B	C
2			
4			

2. Now suppose we consider the previous experiment on a 64-byte cache with 16-byte cache blocks. Fill in the table below with the number of cache misses due to each matrix, assuming that all these arrays are 16-byte aligned. Show your work or briefly explain.

N	A	B	C
2			
4			

3. Even if C is initialized to all zeros and the program executes uninterrupted, after the execution of `mmm` C will not necessarily contain the product of A and B (or an approximation if floating point errors are taken into account). Give an example where C will not be equal to AB.

Problem 2 (10 points)

A fully associative cache imposes the least restrictions on the placements of blocks transferred from memory. This lead to the following question: does a fully associative cache always produce less or at most the same number of cache misses than a not fully-associative cache of the same size and with the same block size (we assume LRU replacement)?

Formally, if S is the number of sets, E the associativity, and B the block size in bytes, then a fully associative cache is described by $(S, E, B) = (1, E, B)$, i.e., the size is EB bytes. A not fully associative cache of the same size and with the same block size is given by (S', E', B) with $S' > 1$ and $S'E' = E$.

If this is true, argue (= provide an informal proof) why this is the case. Otherwise find a counterexample (i.e., find an array access pattern where it does not hold).

Problem 3 (16 points)

Consider the following code, which computes the LU factorization of a given $N \times N$ matrix A. (Note that for this question it does not matter what the function does.)

```
void lu(float A[N][N]) {
  int i,j,k;
  double c;

  for (i = 0; i < N-1; i++) {
    c = 1/A[i][i];
    for (j = i+1; j < N; j++) {
      A[j][i] = c*A[j][i];
      for (k = i+1; k < N; k++)
        A[j][k] = A[j][k] - A[j][i]*A[i][k];
    }
  }
}
```

We assume the following cost measure: floating point addition and multiplication both count 1, and floating point division counts 20. Integer operation are ignored. Compute the cost C(N) of the function `lu`.

Note: Lower-order terms (and only those) may be expressed using big-O notation (this means: as the final result something like $3n + O(\log(n))$ is ok but $O(n)$ is not).

The following formulas may be helpful:

- $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n)$
- $\sum_{i=0}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6} = \frac{n^3}{3} + O(n^2)$

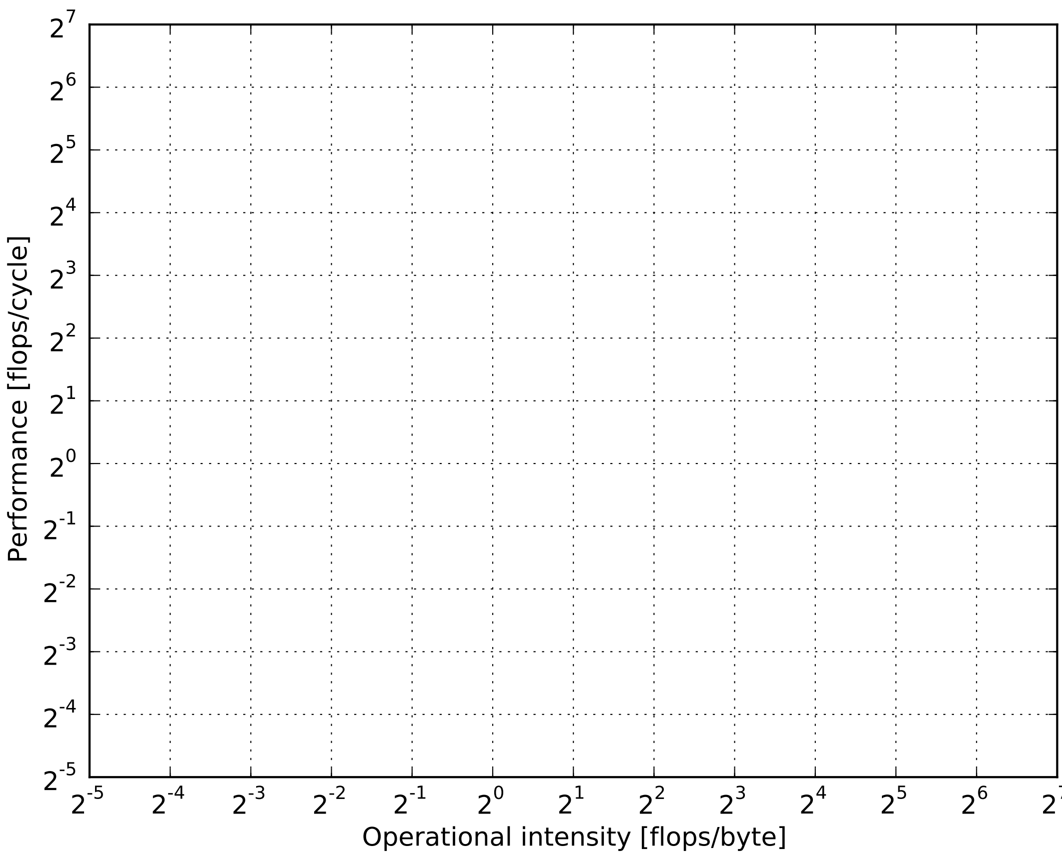
Problem 4 (16 = 6 + 6 + 4 points)

Assume you are using a system with the following features:

- A processor with a peak performance of 8 Gflop/s (double precision), and a CPU frequency of 4 GHz.
- The interconnection between CPU and main memory has a maximal bandwidth of 16 Gbyte/s.

Answer the following questions:

1. Draw the roofline plot for this system. The units for x-axis and y-axis are performance in flops/byte and operational intensity in flops/cycle, both in log scale. The plot will contain two lines determining upper bounds on the achievable performance.



2. Consider the following code:

```
double array[501][501];
double array_out[501][501];

 for(i = 1; i < 500; i++) {
  for(k = 1; k < 500; k++) {
   double t0 = array[i][k];
   double t1 = array[i-1][k-1];
   double t2 = array[i-1][k];
   double t3 = array[i-1][k+1];
   double t4 = array[i][k-1];
   double t5 = array[i][k+1];
   double t6 = array[i+1][k-1];
   double t7 = array[i+1][k];
   double t8 = array[i+1][k+1];

   double a1 = t1 + t2;
   double a2 = t3 + t4;
   double a3 = t5 + t6;
   double a4 = t7 + t8;
   double m1 = a1 * a2;
   double m2 = a3 * a4;
   double m4 = m1 * m2;
   t0 = t0 * m4;

   array_out[i][k] = t0;
  }
}
```

Assuming a cold write-back/write-allocate cache, and that the cache can hold both arrays, compute the operational intensity of this code (ignore write-backs). Show your work.

3. Based on the results of (1) and (2): What is the achievable performance of this code on the given platform? Show how do you compute it.

Problem 5 (16 = 7 + 7 + 2 points)

Consider the following program used to compute $y = y + Ax$ where A is an $N \times N$ sparse matrix stored in CSR format (see Fig. 1 as an example for this format). The matrix A has K non-zero elements, and x and y are (of course) vectors of length N.

```
void smvm(int n, const double* values, const int* col_idx,
      const int* row_start, double* x, double* y)
{
  int i,j;
  double d;

  /* loop over N rows */
  for (i = 0; i < n; i++) {
    d = y[i]; /* scalar replacement since reused */
    /* loop over non-zero elements in row i */
    for (j = row_start[i]; j < row_start[i+1]; j++)
      d += values[j] * x[col_idx[j]];
    y[i] = d;
  }
}
```

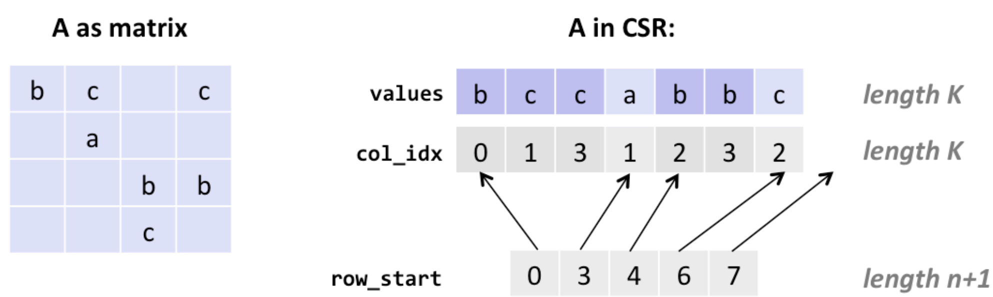


Figure 1: Compressed sparse row (CSR) format.

We assume that every row and every column of A has at least one non-zero element, and that the variables i and j are stored in registers. Further, we assume a cold (empty) cache with a cache block size of 8 bytes. Answer the following questions and provide enough detail so we see how you got to a solution.

1. Compute an upper bound for the operational intensity (unit: flops/byte) assuming only compulsory misses happen.

2. Compute a lower bound for the operational intensity assuming that all array accesses lead to misses.

3. Simplify the two bounds assuming $K = 2N$.

Problem 6 (20 = 4 + 8 + 8 points)

We consider the following program to be run on some desktop system with a recent Intel processor. Even though we compiled the program with the best optimization flags, the performance is far away from the peak. Especially on Parts A, B, and C marked in the code with comments the performance seems very low.

```
#include <assert.h>
#include <math.h>

typedef struct Cube
{
  double c[512][512][512];
  int someattribute;
} cube_t;

/* Returns the element [k][m][n] of the cube */
double get_elt(cube_t* cube, int k, int m, int n);

/* Sets the element [k][m][n] of cube to x */
void set_elt(cube_t* cube, int k, int m, int n, double x);

// this a call to code we cannot modify and that does not modify the input array
double librarycall(double* array, int);

double reduction (double* arr, int i)
{
  // Start Part A
  double sum = 0;
  for (int j = 0; j < 512; j++)
    sum += arr[i*512+j];
  return sum;
  // End Part A
}

double myfunction (cube_t* cube)
{
  double array[512][512]; // we assume it is initialized with 0

  // Start Part B
  for (int i = 1; i < 511; i++)
  {
    for (int k = 1; k < 511; k++)
    {
      double t0 = get_elt(cube,i,k,0);
      double t1 = get_elt(cube,i-1,k-1,0);
      double t2 = get_elt(cube,i-1,k  ,0);
      double t3 = get_elt(cube,i-1,k+1,0);
      double t4 = get_elt(cube,i,  k-1,0);
      double t5 = get_elt(cube,i,  k+1,0);
      double t6 = get_elt(cube,i+1,k-1,0);
      double t7 = get_elt(cube,i+1,k  ,0);
      double t8 = get_elt(cube,i+1,k+1,0);
```

```
      double a1 = t1 + t2;
      double a2 = t3 + t4;
      double a3 = t5 + t6;
      double a4 = t7 + t8;

      double m1 = a1 * a2;
      double m2 = a3 * a4;
      double m4 = m1 * m2;
      t0 = t0 * m4;

      array[i][k] = t0;
    }
  }
  // End Part B

  // Start Part C
  for (int i = 0; i < 512; i++)
  {
    for (int j = 0; j < 512; j++)
    {
      array[j][i] = array[j][i] + librarycall((double *)array, j);
      // librarycall is an external function where after inspecting the source
      // you know that it does not modify the array (no side effect)
    }
  }
  // End Part C

  double max = 0;
  for (int i = 0; i < 512; i++)
  {
    double t = reduction((double*) array, i);
    if (t > max) max = t;
  }
  return max;
}
```

For each of the Parts A, B, C describe a possible reason (or reasons) why the performance is so far from the peak. For each reason very briefly explain how you would try resolve it.

1. Part A:

2. Part B:

3. Part C: