**ETH login ID:**
(Please print in capital letters)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |

**Full name:** _____

**263-2300: How to Write Fast Numerical Code**
ETH Computer Science, Spring 2014
Midterm Exam
Monday, April 14, 2014

**Instructions**

- Make sure that your exam is not missing any sheets, then write your full name and login ID on the front.
- 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 (12) | |
| Problem 2 (15 = 3 + 12) | |
| Problem 3 (17 = 6 + 5 + 6) | |
| Problem 4 (15 = 3 + 3 + 3 + 3 + 3) | |
| Problem 5 (17 = 14 + 3) | |
| Problem 6 (24 = 3 + 3 + 3 + 3 + 12) | |
| **Total** (100) | |

# Problem 1 (12 points)

The function `vecsum` implements $z = x + y$, where $z$, $x$, and $y$ are vectors of length $N$.

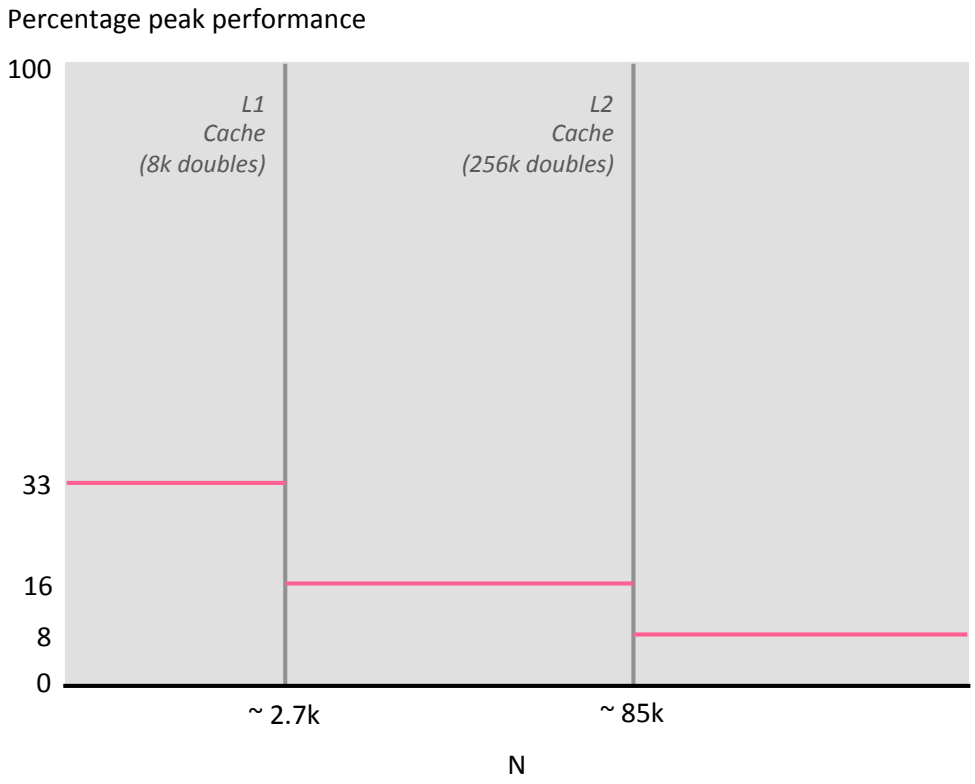
```
void vecsum(double * z, const double * x, const double * y, size_t N) {
  int i;
  for (i = 0; i < N; i++)
    z[i] = x[i] + y[i];
}
```

We make the following assumptions:

- The peak performance of the CPU is $\pi = 1$ add/cycle.
- The system has two levels of cache.
- All caches are write-back/write-allocate.
- L1 cache size: 64 kB; L1 read bandwidth: 1 double/cycle.
- L2 cache size: 2 MB; L2 read bandwidth: 0.5 double/cycle.
- RAM read bandwidth: 0.25 double/cycle.
- The variables $i$ and $N$ are stored in registers.
- A double is 8 bytes.

The performance of `vecsum` is measured as average over many executions. Sketch the expected performance plot for $N$ up to 200'000. $N$ is on the x-axis and the y-axis shows the percentage of peak performance (between 0% and 100%) achieved. Provide enough details and also short explanations so we can verify your reasoning.

**Solution**:


Percentage peak performance
100
33
16
8
0
L1
Cache
(8k doubles)
L2
Cache
(256k doubles)
~ 2.7k
~ 85k
N

# Problem 2 (15 = 3 + 12 points)

Consider the following code, which processes an $M \times N$ matrix $A$. (Note that for this question it does not matter what the function does.)

```
void func(float A[M][N], float th) {

  int i,j,k,l;
  float r,c,t;

  srand(time(NULL));

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++) {

      r = c = 0.f;

      for (k = j+1; k < N; k++) {
        t = (float)(rand())/RAND_MAX;
        c += t*a[i][k];
      }
      for (l = i+1; l < M; l++) {
        t = (float)(rand())/RAND_MAX;
        if (t > th)
          r += a[l][j];
        else
          r -= a[l][j];
      }

      a[i][j] += c*r;
    }
}
```

1. Define a detailed floating point cost measure $C(M, N)$ for the function `func`. Ignore integer operations, function calls, and comparisons.

2. Compute the cost $C(M, N)$.

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

The following formula may be helpful: $\sum_{i=0}^{n-1}(n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + O(n)$.

**Solution**:

1. $C(M,N) = (\text{adds}(M,N), \text{mults}(M,N), \text{divs}(M,N))$.

2.

$$\begin{aligned}
\text{divs}(M,N) &= \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(\sum_{k=j+1}^{N-1} 1 + \sum_{l=i+1}^{M-1} 1\right) \\
&= \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}(N-j-1+M-i-1) \\
&= \sum_{i=0}^{M-1}\left[N(M-i-2)+\frac{N^2}{2}+O(N)\right] = \frac{M^2N}{2}+\frac{MN^2}{2}+O(MN)
\end{aligned}$$

$$\text{adds}(M,N) = \text{divs}(M,N)$$

$$\begin{aligned}
\text{mults}(M,N) &= \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(1+\sum_{k=j+1}^{N-1} 1\right) \\
&= \frac{MN^2}{2}+O(MN)
\end{aligned}$$

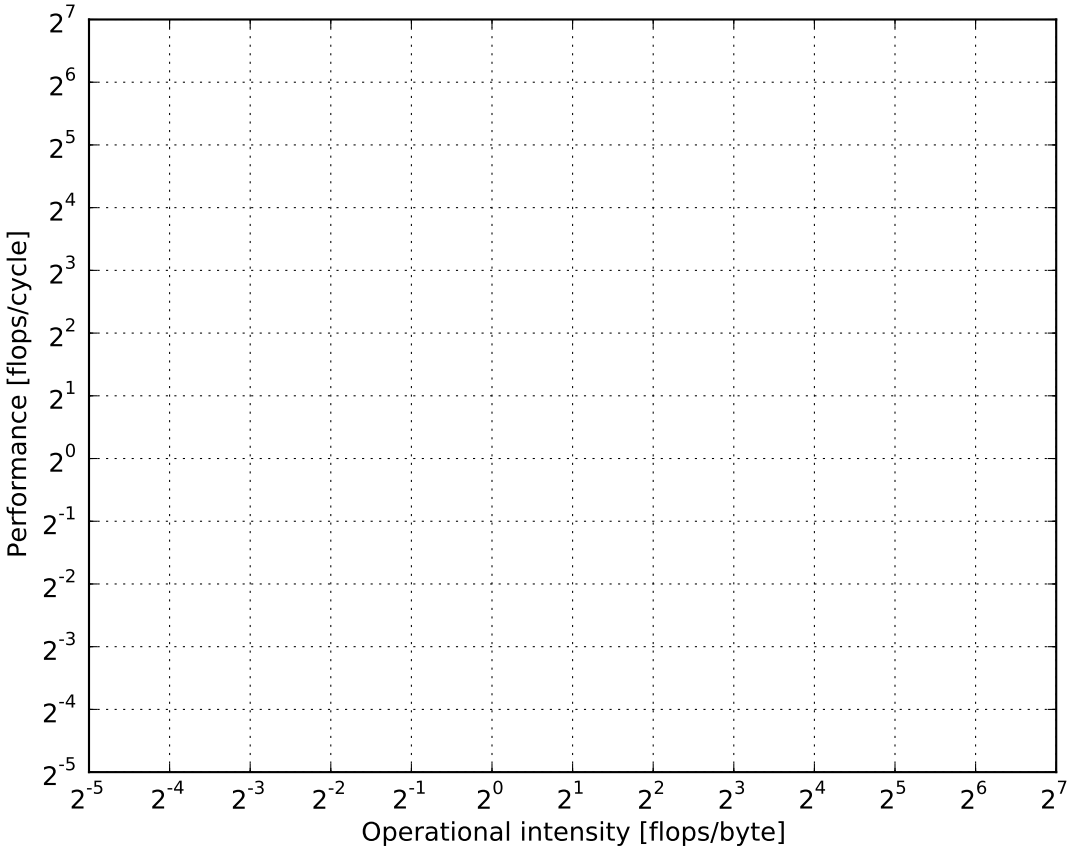
# Problem 3 (17 = 6 + 5 + 6 points)

Assume you are using a system with the following features:

- A CPU that can issue 3 double precision multiplications and 1 double precision addition per cycle.
- The interconnection between CPU and main memory has a maximal bandwidth of 4 bytes/cycle.

Answer the following two questions:

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



2. Consider the following code:

```
void  filter(double m[64][64], double r[62][62]) {

  for(i = 1; i < 63; i++)
    for(j = 1; j < 63; j++)
      r[i-1][j-1] = r[i-1][j-1]
                  + m[i-1][j-1] - m[i-1][j] + m[i-1][j+1]
                  - 2*m[i][j-1] + 2*m[i][j+1]
                  + m[i+1][j];

}
```

Assuming a cold write-back/write-allocate cache with block size $B > 8$ bytes and that the cache can hold the whole matrices `m` and `r`, compute the following. You can approximate 62 with 64 where convenient:

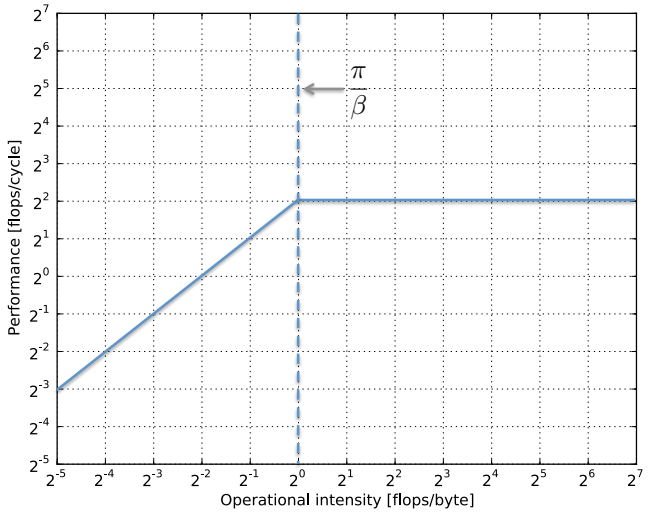
(a) The operational intensity of this code (ignore write-backs).

(b) An upper bound (as tight as possible) on performance.

Show your work.

**Solution**:

1.




2. (a) $I \leq \frac{8 \cdot 62^2}{8 \cdot (64^2 + 62^2)} \approx 0.5$ flop/byte (memory bound on system under consideration).

   (b) The presence of 6 additions in the loop body (8 flops) doesn't reflect the instruction balance required to take maximum advantage from the CPU in the assumptions. Taking this additional limitation into account, we can estimate a bound on performance as:

$$p \leq \min\left(I\beta, \frac{8}{6}\right) \text{ flops/byte } = 1.33 \text{ flops/byte.}$$

# Problem 4 (15 = 3 + 3 + 3 + 3 + 3 points)

Mark the following statements as true (T) or false (F). Explanations are not needed. We denote with $I(n)$ the operational intensity of a function executed on some input of size $n$. Wrong answers give negative points but you cannot get less than 0 points for this problem. You can leave questions unanswered.

☐ Doubling the cache size doubles $I$.

☐ Doubling the cache size can increase $I$.

☐ Assume that we can compute the cost (flop count) of an algorithm for a certain input of size $n$. Then, assuming a cold-cache scenario, we can compute a valid (possibly loose) upper bound for $I(n)$ for all possible C functions that implement this algorithm executed on that input.

☐ A function with a cost in $O(n^2)$ is certainly compute bound.

☐ A function with $I(n) \in O(1)$ is certainly memory bound.

**Solution**:

1. False. Assume the execution of a function which input/output fits completely in cache. Doubling the cache would not affect its operational intensity.
2. True. Assume the execution of a function which input/output produces many capacity misses. Doubling the cache could reduce data traffic leading to an increase of the operational intensity.
3. True. Every C function implementing a given algorithm would have the same flop count of the algorithm itself and at least the same amount of data accesses.
4. False. A possible counterexample would be the case where $W(n), Q(n) \in \Theta(n^2)$ on a system with peak $\pi$ and maximal bandwidth $\beta$ such that $I(n) \leq \frac{W(n)}{Q(n)} \in O(1) < \frac{\pi}{\beta}$.
5. False. As in 4 the answer also depends on the system. In this case just assume to have $\pi$ and $\beta$ such that $I(n) > \frac{\pi}{\beta}$.

# Problem 5 (17 = 14 + 3 points)

Consider the following code for a matrix-matrix multiplication ($c = ab + c$):

```
// a, b, c are n x n matrices (data type double)
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Assume the code is run on a system with a last-level cache of size $C$ bytes and with a cache block size of $B = 32$ bytes. Further, we assume that $40n < C < 8n^2$.

1. Estimate the number of cache misses as a function of $n$. Ignore accesses to the matrix $c$ and possible conflict misses.

2. Compute the operational intensity $I(n)$ based on the previous result.

**Solution**:

1. The cache can contain at least one row of `a` and four columns of `b` but is not able to contain a complete $n \times n$ matrix. For a single `i`-iteration we have the following compulsory and capacity misses:

$$\mathrm{CM}_i(n) = \underbrace{\frac{n}{4}}_{\text{row of a}} + \underbrace{\frac{n^2}{4}}_{\text{entire b}}$$

   The total amount of misses is therefore

$$\mathrm{CM}(n) = n \cdot \mathrm{CM}_i(n) = \frac{n^3 + n^2}{4}.$$

2.

$$I(n) \leq \frac{2n^3}{32\mathrm{CM(n)}} \simeq 0.25 \text{ flop/byte}.$$

# Problem 6 (24 = 3 + 3 + 3 + 3 + 12 points)

We define an image as follow (of course you know that `sizeof(char) == 1`):

```
typedef struct {
  unsigned char   red;
  unsigned char   green;
  unsigned char   blue;
  unsigned char   alpha;
} pixel_t;

pixel_t image[N][16];
```

And we consider a system based on the following assumptions:

- The system has a write-back/write-allocate cache of size $C$ bytes.
- The cache is 4-way set-associative.
- The cache block size is $B = 64$ bytes.
- The cache uses an LRU replacement policy.

Answer the following questions (note: the image is read pixel by pixel):

1. What is the miss rate if we read the whole `image` row-wise and $C = 64N$ bytes?
2. What is the miss rate if we read the whole `image` row-wise and $C = 4N$ bytes?
3. What is the miss rate if we read the whole `image` column-wise and $C = 64N$ bytes?
4. What is the miss rate if we read the whole `image` column-wise and $C = 4N$ bytes?
5. We now assume an image of size 4-by-4 pixels and a small cache with block size $B = 8$ bytes, one set, and a total size of 16 bytes. Provide the hit/miss sequence for the following code. All assignments are executed processing first the right-hand side from left to right. The entire sequence will have a length of $4 \times 6 = 24$.

   ```
   int i,j;
   for(i = 1; i < 3; i++)
     for(j = 1; j < 3; j++) {
       image[i][j].alpha = image[i-1][j].alpha * image[i][j-1].alpha;
       image[i][j].red   = image[i-1][j].red * image[i][j-1].red;
     }
   ```

   Note it helps to sketch the cache and image.

**Solution**:

1. `image` fits completely in cache and a cache block can hold 16 pixels. The miss rate is $\text{MR} = \frac{1}{16}$.

2. This cache provides the same support for spatial locality as the one in question 1. The miss rate is still $\text{MR} = \frac{1}{16}$.

3. Although we access column-wise, `image` still fits completely in cache producing only compulsory misses. Also in this case $\text{MR} = \frac{1}{16}$.

4. The cache can contain up to $\frac{1}{16}$ of `image`. By the time we access the last pixel of column $i$ the first pixel of column $i+1$ is no longer in cache. This leads to a miss rate $\text{MR} = 1$.

5. The cache in this case is fully associative with $E = 2$. A cache block can contain two pixels. The code execution produces the following hit/miss sequence:

   MMH HHH MHM MMM HMH HHH MHM MMM.