263-2300-00: How To Write Fast Numerical Code
Assignment 4: 120 points
Due Date: Th, April 11th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2019/
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=10968.
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework, meaning submit latest 48 hours after the due time. For example, submitting 1 hour late costs 1 late day. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's, Tyler's or Gagandeep's office.
- (Plots)
  For plots/benchmarks, **provide (concise) necessary information for the experimental setup (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions**. Follow (at least to a reasonable extent) the small guide to making plots from the lecture.
- (Code)
  When compiling the final code, ensure that you use optimization flags (e.g. for GCC use the flag "-O3").
- (Neatness)
  5 points in a homework are given for neatness.

Exercises:

1. *Cache mechanics (30 pts)* Consider the following code, executed on a machine with a write-back and write-allocate direct-mapped cache with blocks of size 16 bytes and a total capacity of 4096 bytes. This code takes a matrix and for every element, adds to it the element in the row above it. Memory accesses occur in exactly the order that they appear. The variables `i,j,n,a,` and `b` remain in registers and do not cause cache misses. Assume that A starts at address 0, and n is divisible by 4.

```
1 void add_rows(float * A, int n) {
2     for (int i = 1; i < n; i++) {
3         for (int j = 0; j < n; j++) {
4             float a = A[i*n + j];
5             float b = A[(i-1)*n + j];
6             A[i*n + j] = a + b;
7         }
8     }
9 }
```

   Counting cache misses from both reads and writes, answer the following:

   (a) What is the cache miss rate if $n = 256$?
   Solution:
   Each row is 1 KiB, and by the time we get to the next row, the previous is still in cache. Each element is loaded once and therefore there are $n^2/4$ misses. With $3n(n-1)$ total accesses, it has a miss rate of approximately 1/12.

(b) What is the cache miss rate if $n = 2048$?

Solution:

In this case a row no longer fits in cache. Furthermore, every element of A maps to the same cache block as the element in the row above it. The elements `A[i*n + j]` and `A[(i-1)*n + j]` are 8192 bytes apart, and this is a multiple of the size of the cache. Accessing `A[i*n + j]` bumps `A[(i-1)*n + j]` out of cache and vice-versa.

Three out of every four iterations, `A[i*n+j]` is still in cache from the previous iteration. This is because it is the address right after `A[i*n+j-1]`, and each cache block can hold 4 floats. Then the memory access during `float a = A[i*n+j]` is a miss 25% of the time, the access during `float b = A[(i-1)*n+j]` is always a miss, and the access during `A[i*n + j] = a + b` is always a miss.

The overall miss-rate is then 75%.

(c) What is the cache miss rate if $n = 2052$?

Solution:

In this case a row no longer fits in cache. But the elements `A[i*n + j]` and `A[(i-1)*n + j]` no longer map to the same cache set. Even though n is larger, there are fewer misses. There are $n(n-1)/2$ misses, for a cache miss rate of $1/6$.

2. *LRU vs MRU (20 pts)* When loading a block into cache, set-associative caches require a *policy* to determine which block to evict if the set is already full. A cache with a *least-recently used* (LRU) policy will evict the block that has been used least recently. Another possible heuristic is that of *most-recently used* (MRU), which will evict the block that has been used most recently. If the set is not already full, the policy does not evict anything.

   Consider a 2-way set-associative write-back and write-allocate cache with blocksize 16 bytes. Assume that the cache is empty at the beginning of an operation. Is the cache miss rate for LRU always better than MRU? If yes, provide a proof. If no, provide a counter-example and provide enough detail to explain why MRU is better in this case.

   Solution:

   LRU is not always better than MRU. We now provide a counterexample. Consider a cache with K blocks, K divisible by 2. In the computation below, let x be an array of size n and A and B be an arrays of size n^2. The array x starts at address 0 and A and B start starting immediately after x. Assume all local variables stay in registers. Finally assume that memory accesses occur in exactly the order that they appear.

```
1  void computation(double *x, double *A, double *B, int n) {
2      for(int i = 0; i < n; i++) {
3          x[i] = 0.0;
4      }
5
6      for(int j = 0; j < n; j++) {
7          for(int i = 0; i < n; i+=2) {
8              double a0 = A[i   + j*n];
9              double a1 = A[i+1 + j*n];
10             double b0 = B[i   + j*n];
11             double b1 = B[i+1 + j*n];
12
13             x[i]   += a0 + b0;
14             x[i+1] += a1 + b1;
15         }
16     }
17 }
```

   Let $n = K$. Then the array x fills exactly half of the cache. The elements `A[i + j*n]`, `A[i+1 + j*n]`, `B[i + j*n]`, `B[i+1 + j*n]`, `x[i]`, and `x[i+1]` all map to the same cache set. We unroll by 2 to avoid excess conflict misses. With MRU, the array x is loaded into the first slot in each cache set during the first loop and remains there for the entire computation. With LRU, when `B[i + j*n]` is accessed, the elements $x[i]$ and $x[i+1]$ are evicted from cache. In either case, each element of A and B is loaded into cache exactly one time. LRU performs worse in this case because x must be loaded into cache once per outer loop iteration, whereas in the MRU case, x is loaded into cache once.

3. *Rooflines (40 pts)* You are given a computer with the following parameters:

   - It has a SIMD vector length of 4 single-precision floats.
   - It has two execution ports that can execute floating point operations. Behind each port there is one execution unit that executes multiplications, one that executes additions, and one that executes FMAs.
   - All execution units have a throughput of 1 operation/cycle for both scalar and vector operations.
   - Each instruction has a latency of 1 cycle.
   - The frequency of the CPU is 1 GHz.
   - It has a read bandwidth from main memory of 12 GB/s.

   (a) Draw a roofline plot for the machine. Consider only single-precision floating point arithmetic. Consider only reads. Include a roofline for when vector instructions are not used and for when vector instructions are used.

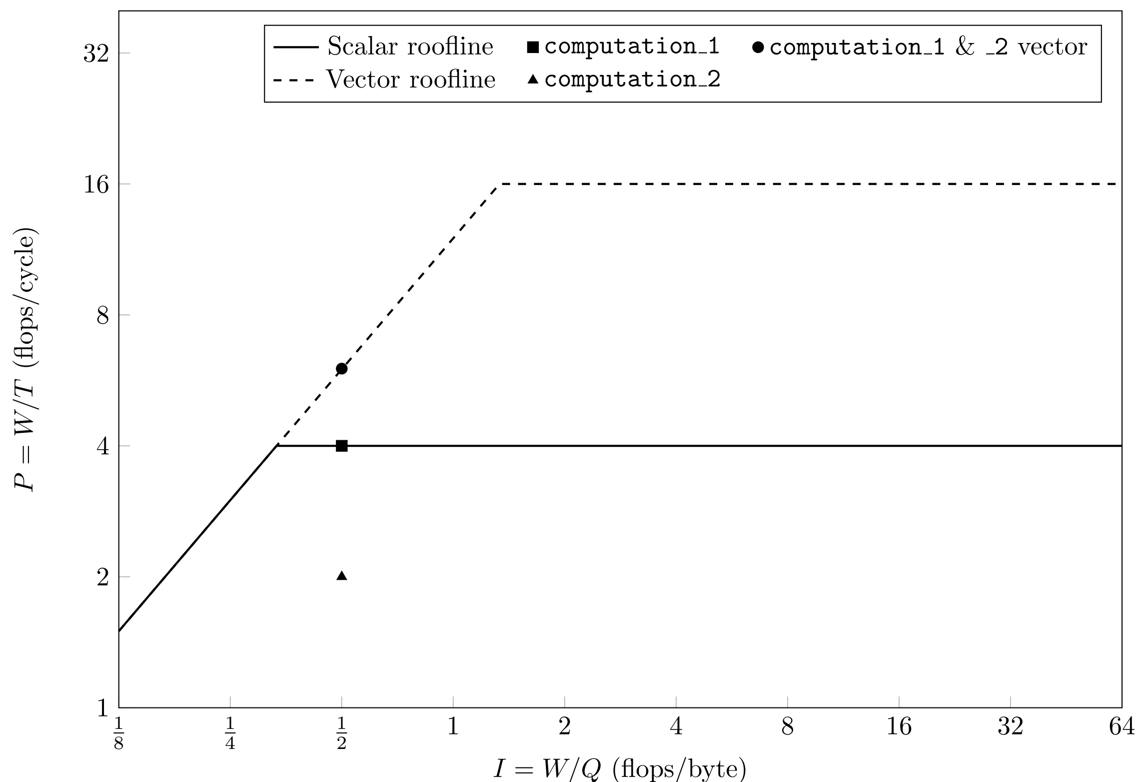
(b) Consider the following functions. For each, assume that vector instructions are not used, and derive hard upper bounds on its operational intensity and performance based on its instruction mix and compulsory misses. Ignore the effects of aliasing. Assume you write code that attains these bounds, and add the performance to the roofline plot (there should be two dots).

```
1 void computation_1(float *x1, float *x2, float *y1, float *y2,
2     float a, float b, float c, float d, int n)
3 {
4     for(int i = 0; i < n; i++) {
5         x1[i] = x1[i] + a * y1[i] + b * y2[i];
6         x2[i] = x2[i] + c * y1[i] + d * y2[i];
7     }
8 }
```

```
1 void computation_2(float *x1, float *x2, float *y1, float *y2,
2     float a, float b, float c, float d, int n)
3 {
4     for(int i = 0; i < n; i++) {
5         x1[i] = x1[i] + a + y1[i] + b + y2[i];
6         x2[i] = x2[i] + c + y1[i] + d + y2[i];
7     }
8 }
```

(c) For each computation, what is the maximum speedup you could achieve by parallelizing it with vector intrinsics?

Solution:



Without vectorization, computation_1 can do at most 4 flops per cycle, and computation_2 can do 2. The function computation_1 can achieve peak scalar performance, but computation_2 can only achieve half of peak performance because it cannot use FMA instructions. After vectorization, they both become bandwidth limited, and could do 6.02 flops per cycle. This gives a speedup of roughly one and a half times faster for computation_1 and roughly three times faster for computation_2.

4. *Balance Principles (25 pts)* The I/O cost (Q in class) of an algorithm is the amount of data that is read from and written to main memory during the execution of the algorithm. The ordinary matrix-matrix multiplication (MMM) operation $C := AB + C$ takes $2n^3$ flops. When n is large and ignoring lower-order terms, the optimal I/O cost when executing MMM is $\frac{2n^3}{\sqrt{m}}$ matrix elements transferred, where m is the number of elements that can fit in cache.

   (a) Given a machine with a peak performance of π double-precision flops/cycle and a cache size of γ bytes, what is the minimum value of β such that double-precision MMM can achieve peak performance, where β is the memory bandwidth in bytes/cycle?

   Solution:

   In order to achieve peak performance, if Q is the I/O cost and W is the number of flops, then $\frac{Q}{\beta} \leq \frac{W}{\pi}$. Thus in order to reach peak we require that

   $$\frac{2n^3}{\sqrt{m}\beta} \leq \frac{2n^3}{\pi}$$

   if m is the capacity of cache in elements. Substituting $m = \gamma/8$ gives us

   $$\frac{4\ sqrt 2n^3}{\sqrt{\gamma}\beta} \leq \frac{2n^3}{\pi}$$

   Solve for β:

   $$\beta \geq \frac{2\sqrt{2}\pi}{\sqrt{\gamma}} \text{ bytes/cycle} \tag{1}$$

   (b) If γ is doubled and π stays fixed, how does this change the minimum value of β required for MMM to achieve peak performance?

   Solution:

   We use equation 1 from (a). Doubling γ gives us a requirement of $\beta \geq \frac{2\pi}{\sqrt{\gamma}}$, so the minimum value of β is divided by $\sqrt{2}$.

   (c) If π is doubled and β stays fixed, how does this change the minimum value of γ required for MMM to achieve peak performance?

   Solution:

   We rewrite our equation to solve for γ.

   $$\beta \geq \frac{2\sqrt{2}\pi}{\sqrt{\gamma}}$$

   $$\sqrt{\gamma} \geq \frac{2\sqrt{2}\pi}{\beta}$$

   $$\gamma \geq 8\left(\frac{\pi}{\beta}\right)^2 \text{ bytes}$$

   Then if we double π, and hold β constant, the lower bound on γ is multiplied by 4.