263-0007-00: Advanced Systems Lab
Assignment 4: 120 points
Due Date: April 20th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2023/
Questions: fastcode@lists.inf.ethz.ch

Academic integrity:
All homeworks in this course are single-student homeworks. The work must be all your own. Do not copy any parts of any of the homeworks from anyone including the web. Do not look at other students' code, papers, or exams. Do not make any parts of your homework available to anyone, and make sure no one can read your files. The university policies on academic integrity will be applied rigorously.

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system
- (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 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.
- (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.
- (Neatness)
  5 points in a homework are given for neatness.

Exercises

1. *Stride Access (15 pts)*

   Consider the following code executed on a machine with a direct-mapped cache with blocks of size 32 bytes and a total capacity of 1 KiB. Assume that the only memory accesses are to entries of x and occur in the order that they appear (from left to right when in the same line). The cache is initially cold and array x begins at memory address 0.

```
1  double comp(double x[256], int s1, int s2) {
2      double sum = 0.0;
3      for (int i = 0; i < 64; i++) {
4          int j = i + 64;
5          sum += x[s1*i] * x[s2*j];
6      }
7      return sum;
8  }
```

   Asnswer the following. Justify your answers:

   (a) Determine the miss rate when $s_1 = 1$ and $s_2 = 1$.
   Solution: Accesses to x_i and x_j don't have conflicts. Thus, the miss rate is 25%.

   (b) Determine the miss rate when $s_1 = 2$ and $s_2 = 2$.
   Solution: Accesses to x_i and x_j conflict with each other. Thus, the miss rate is 100%.

(c) Determine the miss rate when $s_1 = 2$ and $s_2 = 1$.

Solution: Accesses to x_i and x_j don't conflict with each other. Further, the last 32 iterations of x_i benefit from temporal locality (previously accessed by x_j) and are always hit. Thus, the miss rate is 25%

(d) Repeat b) assuming now that the cache is 2-way set associative with a LRU replacement policy. The cache size and block size stay the same.

Solution: Accesses to x_i and x_j are mapped into the same set but they are not evicted due to the higher associativity. Thus, the computation benefit from spacial locality and has a miss rate of 50%.

2. *Cache Mechanics (35 pts)*

Consider the following code executed on a machine with a direct-mapped write-back/write-allocate cache with blocks of size 8 bytes and a total capacity of 64 bytes. Assume that memory accesses occur in exactly the order that they appear. The variables `i,j,m` and `sum` remain in registers and do not cause cache misses. Array x is cache-aligned (first element goes into first cache block) and the first element of y is immediately after the last element of x in memory. Both arrays have size $n = 12$. Assume a cold cache scenario. `sizeof(float)` = `sizeof(int32_t)` = 4 bytes.

```
1  struct data_t {
2      float   a;
3      float   b;
4      int32_t u[3];
5  };
6
7  double comp(data_t x[12], data_t y[12]) {
8      float sum = 0;
9      int m = 6;
10     for (int i = 0; i < 3; i++) {
11         for (int j = 0; j <= 9; j+=3) { // j incremented by 3
12             sum += x[(2*i+j)%m].a;
13             sum += y[(4*i+j)%m].b;
14             sum += x[(4*i+j)%m].b;
15         }
16         m = m - 1;
17         // Show state of cache here
18     }
19     return sum;
20 }
```

(a) Considering the cache misses of the computation, do the following two things for each iteration of the outermost loop: i) determine the miss/hit pattern for x and y (something like x: MMHH..., y: MMMH...); ii) draw the state of the cache at the end of each iteration (i.e. at line 17). See the example below that shows how to draw the cache. Show your work.

i. Miss/hit pattern and state of the cache in the first iteration ($i = 0$).

ii. Miss/hit pattern and state of the cache in the second iteration ($i = 1$).

iii. Miss/hit pattern and state of the cache in the third iteration ($i = 2$).

Solution:

Miss/hit pattern:

i	x	y
0	MHMMMHHM	MMMM
1	MMMHHHMM	MMMM
2	MHHMHHHM	MMHM

State of the cache:

$i = 0$

Set	0
0	$x_3.b$, $x_3.u$
1	
2	
3	
4	
5	
6	$y_3.b$, $y_3.u$
7	$x_2.u$, $x_3.a$

$i = 1$

Set	0
0	$x_3.b$, $x_3.u$
1	
2	$x_0.u$, $x_1.a$
3	$y_2.a$, $y_2.b$
4	
5	$x_2.a$, $x_2.b$
6	$y_3.b$, $y_3.u$
7	$x_2.u$, $x_3.a$

$i = 2$

Set	0
0	$x_3.b$, $x_3.u$
1	$y_1.b$, $y_1.u$
2	$x_0.u$, $x_1.a$
3	$x_1.b$, $x_1.u$
4	
5	$x_2.a$, $x_2.b$
6	$y_3.b$, $y_3.u$
7	$x_2.u$, $x_3.a$

(b) Repeat the previous task assuming now that the cache is 2-way set associative and uses a LRU replacement policy. The cache size and block size stay the same.

Solution:

Miss/hit pattern:

i	x	y
0	MHMMHHHHH	MMHH
1	MMMHHHMM	MMMM
2	HHHHHHMM	MHHM

State of the cache:

$i = 0$

Set	0	1
0	$x_3.b$, $x_3.u$	$x_0.a$, $x_0.b$
1		
2	$y_3.b$, $y_3.u$	$y_0.a$, $y_0.b$
3	$x_2.u$, $x_3.a$	

$i = 1$

Set	0	1
0	$x_3.b$, $x_3.u$	$x_0.a$, $x_0.b$
1	$x_2.a$, $x_2.b$	
2	$y_3.b$, $y_3.u$	$x_0.u$, $x_1.a$
3	$x_2.u$, $x_3.a$	$y_2.a$, $y_2.b$

$i = 2$

Set	0	1
0	$x_3.b$, $x_3.u$	$x_0.a$, $x_0.b$
1	$x_2.a$, $x_2.b$	$y_1.b$, $y_1.u$
2	$y_3.b$, $y_3.u$	$x_0.u$, $x_1.a$
3	$x_1.b$, $x_1.u$	$y_2.a$, $y_2.b$

3. *Rooflines (40 pt)* Consider a processor with the following hardware parameters (assume 1GB = 10^9B):

   - SIMD vector length of 256 bits.
   - The following instruction ports that execute floating point operations:
     - Port 0 (P0): FMA, ADD, MUL
     - Port 1 (P1): ADD, MAX

     Each can issue 1 instruction per cycle and each instruction has a latency of 1.
   - One write-back/write-allocate cache with blocks of size 64 bytes.
   - Read bandwidth from the main memory is 48 GB/s.
   - Processor frequency is 2 GHz.

   (a) Draw a roofline plot for the machine. Consider only double-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 performance and operational intensity (consider only **reads**) based on its **instruction mix** and **compulsory misses**. Ignore the effects of aliasing and assume that no optimizations that change operational intensity are performed (the computation stays as is). FMAs are used to fuse an addition with a multiplication whenever applicable. All arrays are cache-aligned (first element goes into first cache set) and don't overlap in memory. You can further assume that the *max* function is translated into its respective instruction by the compiler and that variables a, b, c, n and i stay in registers. Assume you write code that attains these bounds, and add the performance to the roofline plot (there should be three dots).

```
1 void comp1(double *x, double *y, double *z, double a, double b, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     z[i] = a * x[i] + y[i] + z[i] * max(x[i] + b, y[i] + c);
4   }
5 }
```

```
1 void comp2(double *x, double *y, double *z, double a, double b, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     z[i] = a + x[i] + y[i] + z[i] + max(x[i] + b, y[i] + c);
4   }
5 }
```

```
1 void comp3(double *x, double *y, double *z, double a, double b, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     z[i] = a * x[i] * y[i] * z[i] * max(x[i] * b, y[i] * c);
4   }
5 }
```

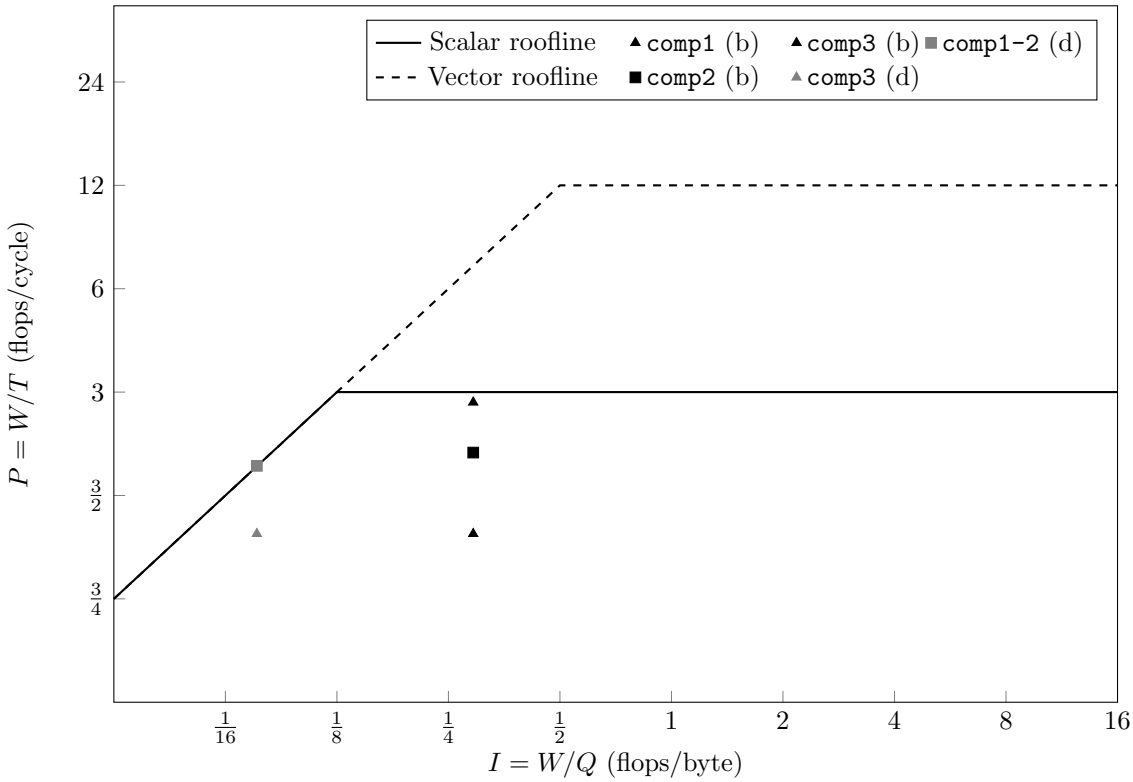
   (c) For each computation, what is the maximum speedup you could achieve by parallelizing it with vector intrinsics?

   (d) Repeat part (b) assuming the following modification in the memory access pattern (strided access). We only show `computation1`, but assume the same modification in `computation2` and `computation3`. Arrays x and y have an according larger size. Add the new performance of each function to the roofline plot (three additional dots).

```
1 void comp1(double *x, double *y, double *z, double a, double b, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     z[i] = a * x[i] + y[i] + z[i] * max(x[16*i] + b, y[2*i] + c);
4   }
5 }
```

Solution:

Roofline plot:



(a) The maximum performance is achieved when executing 1 FMAs and 1 ADD every cycle. Thus, the maximum performance is 3 flops/cycle and 12 flops/cycle for scalar and vectorized code respectively. The rooflines are $P \leq \pi_s$, $P \leq \pi_v$ and $P \leq I \cdot \beta$, where $\pi_s = 3$, $\pi_v = 12$ and $\beta = \frac{48 \cdot 10^9}{2 \cdot 10^9} = 24$ bytes/cycle.

(b) Considering only reads and compulsory misses, a lower bound on the number of bytes transferred is $Q(n) \geq 8(3n) = 24n$. The number of flops in the computations is $W(n) = 7n$ and their operational intensity is therefore $I(n) \leq \frac{7n}{24n} = \frac{7}{24}$ flops/byte. This means that the computations are compute bound and their performance is upper bounded by the roofline $\pi_s = 3$ flops/cycle.
Based only on the instruction mix, `comp1` performs $2n$ ADDs, $2n$ FMAs and n MAXs. Thus, its runtime is $T_1 \geq 2.5n$ and its performance is $p_1 \leq \frac{7n}{2.5n} \approx 2.8$ flops/cycle. `comp2` performs $6n$ ADDs and n MAXs. Thus, its runtime is $T_2 \geq 3.5n$ and its performance is $p_2 \leq \frac{7n}{3.5n} = 2$ flops/cycle. Finally, `comp3` performs $6n$ MULs and n MAXs. The bottleneck are the multiplications; hence, its runtime is $T_3 \geq 6n$ and its performance is $p_2 \leq \frac{7n}{6n} \approx 1.16$ flops/cycle.

(c) The operational intensity is the same as in (b), i.e., $I(n) \leq \frac{7}{24}$ but the computations are now memory bound compared to the roofline using vector instructions. Thus, the performance for `comp1` and `comp2` are now $p_1 = p_2 \leq \beta \cdot I = 24 \cdot \frac{7}{24} = 7$ flops/cycle. The performance of `comp3` does not reach the roofline: $p_3 \approx 4.66$. Thus,

$$speedup_1 \leq 7/2.8 = 2.5$$
$$speedup_2 \leq 7/2 = 3.5$$
$$speedup_3 \leq 4$$

(d) With this access pattern and considering that the block size is 64 bytes, the new operational intensity in this case is $I(n) \leq \frac{W}{Q} = \frac{7n}{8 \cdot 11.5n} \approx 0.076$. With this intensity, the computations becomes memory bound and the performance of `comp1` and `p2` is $p_1 = p_2 \leq I \cdot \beta = \frac{7}{8 \cdot 11.5} \cdot 24 \approx 1.83$ flops/cycle. The performance of `comp3` does not reach the roofline and stays at $p_3 \leq 1.16$.

4. *Cache Miss Analysis (25 pts)*

Consider the following computation that performs a matrix multiplication $C = C + AB^T$ of square matrices A, B and C of size $n \times n$ using a j-i-k loop.

```
1 void mmm_jik(double *A, double *B, double *C, int n) {
2     for(int j = 0; j < n; j++)
3         for(int i = 0; i < n; i++)
4             for(int k = 0; k < n; k+=2)
5                 C[n*i + j] += A[n*i + k]*B[n*j + k] + A[n*i + k+1]*B[n*j + k+1];
6 }
```

Assume that the code is executed on a machine with a write-back/write-allocate fully-associative cache with blocks of size 64 bytes, a total capacity of γ doubles and with a LRU replacement policy. Assume that n is divisible by 8, cold caches, and that all matrices are cache-aligned. Justify all your answers.

(a) Assume that $\gamma << n$ and determine, as precise as possible, the total number of cache misses that the computation has. For each of the matrices (A, B and C), state also the kind(s) of locality it benefits from to reduce misses.

Solution: Accesses to matrices A and B benefit from spacial locality only and produce $n^3/8$ misses each. Accesses to C benefits from temporal locality only and produces n^2 misses. Thus, the total number of misses is $n^3/4 + n^2$.

(b) Repeat the previous task assuming that we interchange the i-loop and the k-loop, i.e., we have now a j-k-i configuration. Assume that the body of the computation stays the same.

Solution: In this case, the accesses to matrix A produce $n^3/2$ misses and benefit from temporal locality in the body of the computation only. Accesses to B produce $n^2/8$ misses and benefit from both temporal and spacial locality. Finally, accesses to C produce $n^3/2$ misses and only benefit from the trivial temporal locality in the loop body when reading and writing to the same memory location. The total number of misses is therefore $n^3 + n^2/8$.

(c) Using the j-k-i configuration of the previous task, determine the minimum value for γ, as precise as possible, such that the computation only has compulsory misses. For this, assume that LRU replacement is not used and, instead, cache blocks are replaced as effectively as possible to minimize misses.

Solution: In order to have compulsory misses only, the cache should be able to fit the complete A matrix, 1 block of B, and 1 block-column (i.e. 8 columns) of C. Thus, $\gamma \geq n^2 + 8n + 8$ doubles.