Student Name: Marios G. Angelis
Student ID: 22-953-186
Spring Semester 2023

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab

Homework 4

---

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  }
```

   Answer the following. Justify your answers:

   (a) Determine the miss rate when s1 = 1 and s2 = 1.
   Solution:
   The miss rate is 25%. We have 2 misses per 4 iterations. For example, there are two misses when i=0 and j=64 for elements x[0] and x[64] respectively. For each miss, a cache block that contains 4 doubles is loaded from the main memory. More specifically, vector elements x[0]-x[3] are loaded in the first set of the cache, and elements x[64]-x[67] are loaded in the 16th set of the cache. Thus, when i=1, i=2, and i=3 we have only hits for elements x[1]-x[3] and x[65]-x[67]. In total, there are 32 misses for 128 accesses.

   (b) Determine the miss rate when s1 = 2 and s2 = 2.
   Solution:
   The miss rate is 100%. We have 2 misses per iteration. For example, there are two misses when i=0 and j=64 for elements x[0] and x[128] respectively. For each miss, a cache block that contains 4 doubles is loaded from the main memory. More specifically, vector elements x[0]-x[3] are loaded in the first set of the cache, and elements x[128]-x[131] are loaded again in the 1st set of the cache replacing x[0]-x[3]. Then, when i=1 and j=65, we have a miss for element x[2] since elements x[128]-x[131] are located in the first set of the cache. Thus, vector elements x[0]-x[3] are loaded in the first set of the cache replacing x[128]-x[131]. Then, we have another miss for the element x[130], and elements x[128]-x[131] are loaded again in the 1st set of the cache replacing elements x[0]-x[3]. In total, there are 128 misses for 128 accesses.

   (c) Determine the miss rate when s1 = 2 and s2 = 1.
   The miss rate is 25%. For iterations 0-31, we have 3 misses per 4 iterations. More specifically, when i=0 and j=64, we have 2 misses for elements x[0] and x[64]. Vector elements x[0]-x[3] are loaded in the first set of the cache, and elements x[64]-x[67] are loaded in the 16th set of

the cache. Then, when i=1 and j=65, we have 2 hits for elements x[2] and x[65]. Next, when i=2 and j=66, we have a miss for x[4] and a hit for x[66] so elements x[4]-x[7] are loaded in the second set of the cache. Finally, when i=3 and j=67, we have 2 hits for elements x[6] and x[67]. This pattern is repeated for the first 32 iterations of i generating 24 misses.

For iterations 32-63, we have 1 miss per 4 iterations. More specifically, when i=32 and j=96, we have a hit for element x[64] since elements x[64]-x[67] were loaded in the 16th set of the cache from the first iteration (i=0), and a miss for element x[96]. Thus, elements x[96]-x[99] are loaded in the 24th set of the cache. This pattern is repeated for the last 32 iterations of i generating 8 misses. In total, we have 32 misses for 128 accesses (24 misses from the first 32 iterations and 8 misses from the rest iterations).

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

Solution:

Since the cache is 2-way associative, we have 16 sets. The miss rate is 50%. We have 2 misses per 2 iterations. For example, there are two misses when i=0 and j=64 for elements x[0] and x[128] respectively. For each miss, a cache block that contains 4 doubles is loaded from the main memory. More specifically, vector elements x[0]-x[3] are loaded in the first set of cache way 0, and elements x[128]-x[131] are loaded in the first set of cache way 1. When i=1, we have 2 hits for elements x[2] and x[130]. This pattern is repeated for the rest iterations of i generating 64 misses for 128 accesses in total.

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. $sizeof(float) = sizeof(int32\_t) = 4bytes$.

```
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:

The cache has 8 sets and each set can fit 2 floats. The first iteration (i=0) produces 3 hits and 9 misses with the following hit/miss pattern: MMHMMMMMHHMM. The second iteration (i=1) produces 3 hits and 9 misses with the following hit/miss pattern: MMMMMHHMHMMM. The third iteration (i=2) produces 6 hits and 6 misses with the following hit/miss pattern: MMHHMMHHHHMM.

The state of the cache at the end of the first, second, and third iteration are respectively:

Set	0
0	x[3].b, x[3].u0
1	
2	
3	
4	
5	
6	y[3].b, y[3].u0
7	x[2].u2, x[3].a

Set	0
0	x[3].b, x[3].u0
1	
2	x[0].u2, x[1].a
3	y[2].a, y[2].b
4	
5	x[2].a, x[2].b
6	y[3].b, y[3].u0
7	x[2].u2, x[3].a

Set	0
0	x[3].b, x[3].u0
1	y[1].b, y[1].u0
2	x[0].u2, x[1].a
3	x[1].b, x[1].u0
4	
5	x[2].a, x[2].b
6	y[3].b, y[3].u0
7	x[2].u2, x[3].a

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

Solution:

The cache is 2-way set associative and it has 4 sets. The first iteration (i=0) produces 7 hits and 5 misses with the following hit/miss pattern: MMHMMMHHHHHH. The second iteration (i=1) produces 3 hits and 9 misses with the following hit/miss pattern: MMMMMHHMHMMM. The third iteration (i=2) produces 8 hits and 4 misses with the following hit/miss pattern: HMHHHHHHHMMM.

The state of the cache at the end of the first, second, and third iteration are respectively:

Set	0	1
0	x[0].a, x[0].b	x[3].b, x[3].u0
1		
2	y[0].a, y[0].b	y[3].b,y[3].u0
3	x[2].u2, x[3].a	

Set	0	1
0	x[3].b, x[3].u0	x[0].a, x[0].b
1	x[2].a, x[2].b	
2	x[0].u2, x[1].a	y[3].b,y[3].u0
3	x[2].u2, x[3].a	y[2].a,y[2].b

Set	0	1
0	x[3].b, x[3].u0	x[0].a, x[0].b
1	x[2].a, x[2].b	y[1].b,y[1].u0
2	x[0].u2, x[1].a	y[3].b,y[3].u0
3	x[1].b, x[1].u0	y[2].a,y[2].b

3. Rooflines (40 pt) Consider a processor with the following hardware parameters:

   - 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 }
```

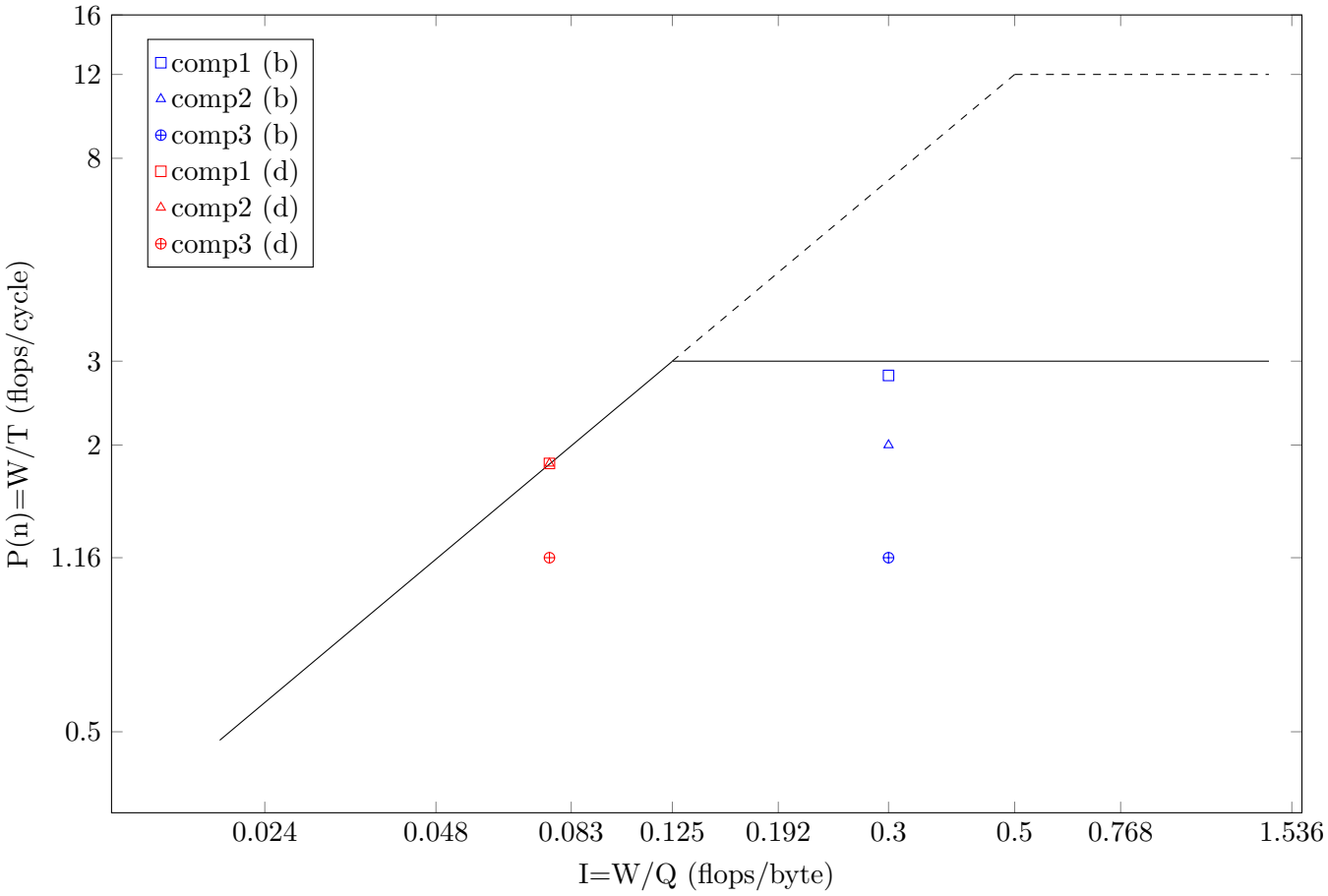
```
1 void comp3(double *x, double *y, double *z, double a, double b, double c, int n) {
2   for (int i = 0; i < n; i++) {
3     z[i] = a * x[i] * y[i] * z[i] * max(x[i] * b, y[i] * c);
4   }
5 }
```

   (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:



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

(b) Considering only reads and compulsory misses, a lower bound on the number of bytes transferred is $Q(n) \geq 8 * (3n) = 24n$ for each of the functions. Also, the number of floating point operations $W(n) = 7n$ for each of the functions. Thus, the operational intensity is $I(n) \leq \frac{7n}{24n} = 0.29$ flops/byte. This means that all the computations are compute-bound and their performance is upper bounded by 3 flops/cycle. Now, based only on the instruction mix, comp1 performs 2n ADDs, n MAX operations and 2n FMAs. Thus, its runtime will be $T1 \geq \frac{5n}{2} = 2.5n$ cycles and its performance will be $P1 \leq \frac{7n}{2.5n} = 2.8$ flops/cycle. comp2 performs 6n ADDs and n MAX operations. Thus, its runtime will be $T2 \geq \frac{7n}{2} = 3.5n$ cycles and its performance will be $P2 \leq \frac{7n}{3.5n} = 2$ flops/cycle. Finally, comp3 performs 6n MULTs and n MAX operations. Thus, its runtime will be $T3 \geq 6n$ cycles and its performance will be

$P3 \leq \frac{7n}{6n} = 1.16$ flops/cycle.

(c) The operational intensity is the same as in (b),$I(n) \leq 0.29$ but since we use vector instructions, the computations are now memory bound and performance will be upper bounded by $P(n) \leq I(n) * \beta = 7$ flops/cycle. Thus, we have:

$$speedup_1 \leq \frac{min(4 * oldP, 7)}{oldP} = \frac{7}{2.8} = 2.5x$$
$$speedup_2 \leq \frac{min(4 * oldP, 7)}{oldP} = \frac{7}{2} = 3.5x$$
$$speedup_3 \leq \frac{min(4 * oldP, 7)}{oldP} = \frac{4.64}{1.16} = 4x$$

(d) With this access pattern and considering that the block size is 64 bytes, the new operational intensity, in this case, is $I(n) \leq \frac{7n}{8*11.5n} = 0.076$ flops/byte. $Q(n) \geq 8*11.5n$ since 2n blocks will be loaded for vector y, n blocks will be loaded for vector z, 8n blocks will be loaded for vector x from the access pattern x[16i] and 0.5n blocks will be loaded for vector x from the access pattern x[i]. This happens since access pattern x[i] will fetch 1 block per 16 iterations instead of 2 blocks per 16 iterations because the other block will have been already fetched by access pattern x[16i]. With this intensity, the computations become memory bound and the performance is bounded by $P(n) \leq I(n) * \beta = 1.83$ flops/cycle. Thus, $P1(n) = 1.83, P2(n) = 1.83, P3(n) = 1.16$ flops/cycle.

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 × n using a j-i-k loop and:

```
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 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 benefit from temporal locality only and produce n^2 misses. Thus, the total number of misses is $n^2 + n^3/4$.

(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:
Accesses to matrix A benefit from spacial locality only and produce $n^3/2$ misses. Accesses to matrix B benefit from both temporal and spatial locality only and produce $n^2/8$ misses. Accesses to matrix C do not benefit from any kind of locality and produce $n^3/2$ misses. Thus, the total number of misses is $n^3 + n^2/8$.

(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 cache block of B and 8 columns of C. Thus, γ should at least be $n^2 + 8(n + 1)$.