# 263-0007-00: Advanced Systems Lab

Assignment 4: 120 points
Due Date: April 18th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2024/
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.

The exercises start from the next page.

## Exercises

1. *Stride Access (30 pts)*

   Consider the following code executed on a machine with a cache with blocks of size 32 bytes and a total capacity of 1 KiB. Assume that the only memory accesses are to entries of $O$ and $A$ and occur in the order that they appear (from left to right when in the same line). The cache is initially cold and array $A$ begins at memory address 0, while array $O$ begins immediately after the last element of $A$. You can assume that $A$ and $O$ are of size $n \times n$.

```
 1  double stencil(double* A, int lda){
 2    double acc = A[0];
 3    acc += A[-lda];
 4    acc += A[lda];
 5    return acc;
 6  }
 7
 8  void comp(double* A, double* O, int n, int s){
 9    for(int i = 1; i < n-1; i++){
10      for(int j = 0, jj = 0; j < n; j+=s, jj++){
11         O[(i-1)*n/s+jj] = stencil(&A[i*n+j], n);
12      }
13    }
14  }
```

   Asnswer the following. Justify your answers:

   (a) Consider a direct-mapped cache, determine the miss rate when $s = 1$, $n = 8$.

   **Solution:** All data fits in cache, we have only compulsory misses. Additionally, after the first iteration of loop $i$, we have cache hits on accesses at line 2 and 3. So, miss-rate $= 0.145$.

   (b) Consider a direct-mapped cache, determine the miss rate when $s = 2$, $n = 16$.

   **Solution:** We access 112 values of $O$. This uses 28 rows of the cache (from 0 to 27). $O$ and $A$ start at the same offset, so we have conflicts. $O$ is accessed consecutively, while $A$ is strided. $O$'s miss rate is 25%. There are a total of $(16 - 2) * 16/2 * 4 = 448$ accesses. Of these, there are conflicts at the start and when the stencil does a full loop and catches up to $O$. There are a total of three conflicts, one of which at the very end. There are a total of 107 misses, meaning a miss rate of 0.238 In case we consider the access to $O$ to happen before the one to $A$ we have a total of 106 misses, meaning a miss rate of of 0.236.

   (c) Consider a 2-way associative cache with a LRU replacement policy, determine the miss rate when $s = 2$, $n = 16$.

   **Solution:** There are no more conflicts, therefore we have 25% miss rate on $O$ and $\approx 16\%$ miss rate on $A$ after the first iteration of the $i$ loop. Total number of misses is 92, meaning 20% miss rate.

2. *Cache Mechanics (20 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,i1,j1,x0,y0,res` 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 are of size 10. Assume a cold cache scenario. `sizeof(float)` $= 4$ bytes.

```
 1  struct data_t {
 2    float a;
 3    float b;
 4    float c;
 5    float d;
 6  };
 7
 8  float comp(data_t x[10], data_t y[10]) {
 9    float res = 0;
10    for(int i = 0; i < 2; i++){
11      int i1 = i+1;
12      for(int j = 0; j <= 4; j+=2){
13        int j1 = j+1;
14        float x0 = x[i1*(i1+j1)%4].a;
15        float y0 = y[j1*(i1+j1)%4].b;
16        res += x0 * y0;
17      }
18      // Draw state of cache and write hit miss pattern here
19    }
20
21    for(int i = 0; i < 2; i++){
22      int i1 = i+1;
23      for(int j = 0; j <= 10; j+=5){
24        int j1 = j+1;
25        float x0 = x[i1*(i1+j1)%7].c;
26        float y0 = y[j1*(i1+j1)%7].d;
27        res -= x0 * y0;
28      }
29      // Draw state of cache and write hit miss pattern here
30    }
31    return res;
32  }
```

(a) Considering the cache misses of the computation, do the following two things for each iteration of the two outermost loop:

1. determine the miss/hit pattern for $x$ and $y$ (something like $x$: MMHH..., $y$: MMMH...);
2. draw the state of the cache at the end of each iteration.

Show your work.

i. Miss/hit pattern and state of the cache at line 18, for $i = 0, 1$.

ii. Miss/hit pattern and state of the cache at line 30, for $i = 0, 1$.

**Solution:**

Miss/hit pattern:

| *Pos* | $x$ | $y$ |
|---|---|---|
| First loop | MMMHHH | MMMMHH |
| Second loop | MMMMMM | MMMMHM |

State of the cache:

$i = 0$

| **Set** | 0 |
|---|---|
| 0 | $y_2.a$, $y_2.b$ |
| 1 | |
| 2 | |
| 3 | |
| 4 | $x_2.a$, $x_2.b$ |
| 5 | |
| 6 | |
| 7 | |

$i = 1$

| **Set** | 0 |
|---|---|
| 0 | $y_2.a$, $y_2.b$ |
| 1 | |
| 2 | $y_3.a$, $y_3.b$ |
| 3 | |
| 4 | $x_2.a$, $x_2.b$ |
| 5 | |
| 6 | |
| 7 | |

$i = 0$

| **Set** | 0 |
|---|---|
| 0 | $y_2.a$, $y_2.b$ |
| 1 | $y_6.c$, $y_6.d$ |
| 2 | $y_3.a$, $y_3.b$ |
| 3 | $x_5.c$, $x_5.d$ |
| 4 | $x_2.a$, $x_2.b$ |
| 5 | $y_0.c$, $y_0.d$ |
| 6 | |
| 7 | |

$i = 1$

| **Set** | 0 |
|---|---|
| 0 | $y_2.a$, $y_2.b$ |
| 1 | $y_6.c$, $y_6.d$ |
| 2 | $y_3.a$, $y_3.b$ |
| 3 | $y_3.c$, $3_3.d$ |
| 4 | $x_2.a$, $x_2.b$ |
| 5 | $x_2.c$, $x_2.d$ |
| 6 | |
| 7 | |

(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:

| *Pos* | $x$ | $y$ |
|---|---|---|
| First loop | MMMHHH | MMMMHH |
| Second loop | MMMMMH | MMMMMH |

State of the cache:

$i = 0$

| **Set** | 0 | 1 |
|---|---|---|
| 0 | $y_2.a$, $y_2.b$ | $x_2.a$, $x_2.b$ |
| 1 | | |
| 2 | | |
| 3 | | |

$i = 0$

| **Set** | 0 | 1 |
|---|---|---|
| 0 | $y_2.a$, $y_2.b$ | $x_2.a$, $x_2.b$ |
| 1 | | |
| 2 | $y_3.a$, $y_3.b$ | |
| 3 | | |

$i = 0$

| **Set** | 0 | 1 |
|---|---|---|
| 0 | $y_2.a$, $y_2.b$ | $x_2.a$, $x_2.b$ |
| 1 | $y_6.c$, $y_6.d$ | $y_0.c$, $y_0.d$ |
| 2 | $y_3.a$, $y_3.b$ | |
| 3 | $x_5.c$, $x_5.d$ | |

$i = 0$

| **Set** | 0 | 1 |
|---|---|---|
| 0 | $y_2.a$, $y_2.b$ | $x_2.a$, $x_2.b$ |
| 1 | $y_6.c$, $y_6.d$ | $x_2.c$, $x_2.d$ |
| 2 | $y_3.a$, $y_3.b$ | |
| 3 | $y_3.c$, $y_3.d$ | $x_5.c$, $x_5.d$ |

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

   - SIMD vector length of 256 bits.
   - The following instruction ports that execute integer operations:
     - Port 0 (P0): MAX, ADD, AND, OR
     - Port 1 (P1): ADD, AND, OR
     - Port 5 (P5): AND, OR

     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 30 GB/s.
   - Processor frequency is 2 GHz.
   - Do not consider index computations in your analysis.

   (a) Draw a roofline plot for the machine. Consider only 32-bit integer 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, where all input matrices are of size $n \times n$. 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). 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 all variables stay in registers. Assume you write code that attains these bounds, and add the performance to the roofline plot (there should be two dots).

```c
// A, B, C, M are all of size n * n
void comp1(uint32_t *A, uint32_t *B, uint32_t *C, uint32_t *M, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++){
      A[i*n+j] = A[i*n+j] + max(max(B[i*n+j],M[i*n+j]),C[i*n+j]);
}}}

// A, B, C, M are all of size n * n
void comp2(uint32_t *A, uint32_t *B, uint32_t *C, uint32_t *M, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++){
      A[i*n+j] = A[i*n+j] | (B[i*n+j] & M[i*n+j]) & C[i*n+j];
}}}
```

(c) For each computation, what is the maximum speedup you could achieve by parallelizing it with vector intrinsics? Assume that bitwise operations on 256 bits registers are used, for example _mm256_and_si256(a,b) for *comp2*.

(d) Consider now this new function. Assume that all matrices are of size $n \times m$.

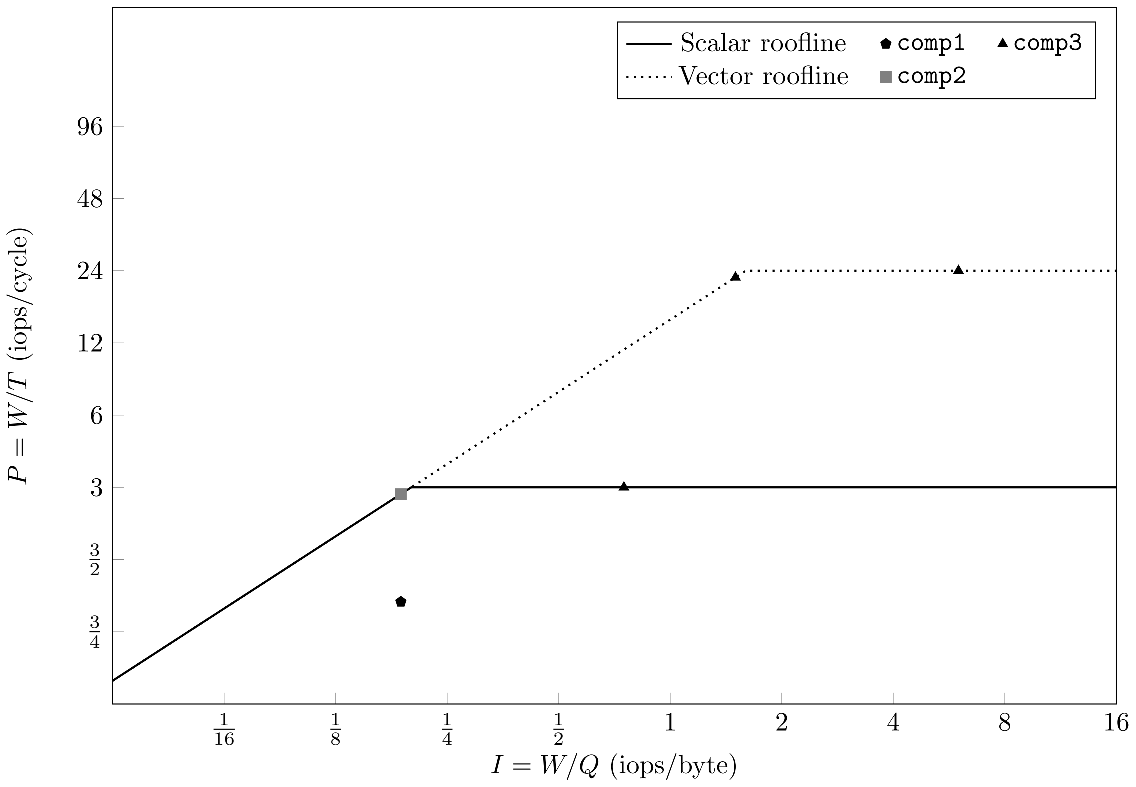
```c
void comp3(uint32_t *A, uint32_t *B, uint32_t *C, uint32_t *M, int n, int m) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++){
      for (int k = 0; k < m; k++){
        A[i*m+j] = A[i*m+j] | (B[i*m+k] & M[i*m+k]) & C[i*m+j];
}}}}
```

Consider only scalar instructions. Derive hard upper bounds on performance and operational intensity based on instruction mix and compulsory misses. Consider only memory reads.

(e) Plot the upper bound to the performance of *comp3* on the roofline plot for $m = 4, 8, 32$. There should be three new dots. Assume that for $m \geq 8$ the code is vectorized.

**Solution:**

Roofline plot:



(a) $\beta = \frac{30}{2} = 15$ bytes/cycle, Max performance is obtained when doing 3 ADDs at the same cycle $\implies$ 3 iops. Memory bound threshold, $\beta I = \pi \implies I = 3/15$, $\beta I = \pi_v \implies I = 24/15$

(b) Considering only compulsory misses.
For comp1: $Q(n) \geq 4 \cdot 4n^2$, $W(n) = 3n^2$, $I(n) = \frac{3n^2}{16n^2} = 3/16$, $T(n) \geq 2n^2$ since it's bottlenecked by the max, $p_1 \leq 1.5$.
For comp2 : $Q(n) \geq 4 \cdot 4n^2$, $W(n) = 3n^2$, $I(n) = \frac{3n^2}{16n^2} = 3/16$, $T(n) \geq 1n^2$, $p_2 \leq 3$. Note that comp2 is memory bound, therefore its performance is 2.81 iops/cycles.

(c) Both *comp1* and *comp2* are memory bound. This means that even though they could achieve a theoretical $8x$ speedup, they can only achieve $1.80\times$ speedup and $1\times$ respectively.

(d) *Comp3* has $W(n,m) = 3nm^2$, and $Q(n,m) \geq 4(4nm) = 16nm$. $I(n,m) \leq 3nm^2/16nm = 3/16m$. $T(n,m) \geq nm^2$ and $p \leq 3$, since the ports can execute ANDs and ORs in parallel. This value is achievable only if $I(n) \geq 3/15 = 0.2$ meaning $m > 1$.

(e) i. $m = 4$, $I = 3/4$, the function is compute bound and $p \leq 3$.
ii. $m = 8$, we can now assume vectoriaztion. $I = 3/2$. The function is memory bound, and maximum performance is $3/2 \cdot 15 = 22.5$.
iii. $m = 32$. $I = 6$, the computation is compute bound and $p \leq 24$.

4. *Cache Miss Analysis (25 pts)*

Consider the following computation that performs a matrix multiplication $C = C + AB$ of a blocked square matrix $A$ of size $n \times n$, and two matrices $B$ and $C$ of size $n \times b$ using a tiled $j$-$i$-$p$-$k$ loop. Note that we iterate only on the nonzero blocks of $A$. Each block $A_i$ of $A$ is of size $b \times b$.

$$\left(\begin{array}{ccc|c} A_0 & 0 & 0 & 0 \\ 0 & A_1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ \hline 0 & 0 & 0 & A_{n/b} \end{array}\right) \left(\begin{array}{c} B_0 \\ B_1 \\ \vdots \\ \hline B_{n/b} \end{array}\right) = \left(\begin{array}{c} A_0 B_0 \\ A_1 B_1 \\ \vdots \\ \hline A_{n/b} B_{n/b} \end{array}\right)$$

```c
void BGEMM(double* A, double* B, double* C, int n, int b){
  for(int j = 0; j < b; j++){
    for(int i = 0; i < n; i+=b){
      for(int p = 0; p < b; p++){
        for(int k = 0; k < b; k++){
          C[(i+p)*b + j] += A[(i+p)*n + i+k] * B[(i+k)*b+j];
        }
      }
    }
  }
}
```

Assume that the code is executed on a machine with a write-back/write-allocate fully-associative cache with blocks of size 32 bytes, a total capacity of $\gamma$ doubles and with a LRU replacement policy. Assume that $n$ is divisible by $b$, $b > 4$, cold caches, and that all matrices are cache-aligned. Justify all your answers.

(a) Assume $\gamma << n$ and $\gamma > 12b$, determine, as precise as possible, the total number of cache misses that the computation has. The result is parametric in $b$ and $n$. For each of the matrices ($A$, $B$ and $C$), state also the kind(s) of locality it benefits from to reduce misses.

**Solution:** Locality type:

- $A$: spatial locality
- $B$: temporal locality
- $C$: temporal locality

$\lceil b/4 \rceil nb$ for $A$, $nb$ for $B$ and $nb$ for $C$.

(b) Determine the minimum value of $\gamma$ such that the computation only has compulsory misses.

**Solution:** In order to have compulsory misses only, the cache should be able to fit the $nnz$ elements of $A$, 4 columns of $B$, and 4 columns of $C$. This gives a total size of $\lceil b/4 \rceil 4n + 8n$ doubles.

(c) Repeat subtask b after you switch loops i and j. Meaning the new order is *i*-*j*-*p*-*k*. Assume that the body of the computation stays the same.

**Solution:** Now we need to store a tile $b \times b$ of A, $b$ blocks of $B$, and 1 block of $C$. This gives a size of $\lceil b/4 \rceil 4b + 4b + 4b$ doubles.

(d) Assume we avoid storing the zero elements of $A$, and we store the blocks $A_0, A_1, \ldots, A_m$ contiguously in memory. Give two performance benefits of this approach.

**Solution:**

i. Fewer TLB misses
ii. Fewer Cache misses