263-0007-00: Advanced Systems Lab
Assignment 4: 120 points
Due Date: Th, April 9th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2020/
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 https://moodle-app2.let.ethz.ch/course/view.php?id=12308.
- (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. *Cache Mechanics (40 pts)*

   Consider the following functions, executed on a machine with a write-back/write-allocate 2-way set associative cache with blocks of size 64 bytes, a total capacity of 4096 bytes and with a LRU replacement policy. These functions take a matrix A with $m = 2112$ columns and n rows. The largest element in each column of A is stored in an array v of size m. Memory accesses occur in exactly the order that they appear. The variables `i,j,m` and `n` remain in registers and do not cause cache misses. Array v is aligned to the cache block and the first element of A is immediately after the last element of v in memory, i.e., `A = v + m`. For this and the following exercises, assume a cold cache scenario.

```
1 void calculate1(float *A, float* v, int m, int n) {
2     for (int i = 0; i < n; ++i) {
3         for (int j = 0; j < m; ++j) {
4             v[j] = max(v[j], A[i * m + j]);
5         }
6     }
7 }
```

```
1 void calculate2(float *A, float* v, int m, int n) {
2     for (int j = 0; j < m; ++j) {
3         for (int i = 0; i < n; ++i) {
4             v[j] = max(v[j], A[i * m + j]);
5         }
6     }
7 }
```

Counting cache misses from both **reads and writes**, answer the following (assume $m = 2112$):

(a) What is the cache miss rate if $n = 8$ for both functions?

(b) What is the cache miss rate if $n = 16$ for both functions?

(c) What is the minimum value of n in which function `calculate2` achieves its highest miss rate?

(d) Suggest an improvement over `calculate2` that achieves a lower miss rate than `calculate1` for the n that you calculated in (c).

2. *Stride access (20)*

Consider the following function, executed on the same machine as in the previous exercise. The array v is aligned with the cache block size. Memory accesses occur in exactly the order that they appear in the code. Thus, no optimizations are performed that reduce the memory accesses or reorder computations. The variables `i,j,n,sum` and `stride` remain in registers and do not cause cache misses.

```
1  double calculate(double* v, int n, int stride)
2  {
3      double sum = 0;
4      for (int j = 0; j < 2; ++j) {
5          for (int i = 0; i < n; ++i) {
6              sum += v[i*stride];
7          }
8      }
9      return sum;
10 }
```

Counting cache misses answer the following:

(a) Assuming $stride = 4$, $n > 1$ and n even, what is the maximum value of n that achieves the highest hit rate?

(b) Under the same assumptions, what is the minimum value of n that achieves the lowest hit rate.

(c) Repeat parts (a) and (b), now assuming $stride = 32$.

(d) Assumming $stride = 16$ and $n = 40$. Determine the hit rate of the function.

3. *Rooflines (40 pts)*

You are given a computer with the following parameters:

- It has a SIMD vector length of 4 double-precision floats.
- It has three execution ports that can execute floating point operations (P0, P1 and P2). Behind P0 and P1 there is one execution unit that executes multiplications, one that executes additions, and one that executes FMAs. In P2 there is only one execution unit that executes additions. The mapping of execution units to ports is as follows:
  - **P0**: FMA, ADD, MUL.
  - **P1**: FMA, ADD, MUL.
  - **P2**: ADD.
- All execution units have a throughput of 1 operation/cycle for both scalar and vector operations.
- Each instruction has a latency of 1 cycle.
- One cache with 64-byte cache block size.
- It has a read bandwidth from main memory of 32 bytes/cycle.

(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). For `computation3`, FMAs are used to fuse an addition with a multiplication. Both arrays (x and y) are cache-aligned (first element goes into first cache block). Assume you write code that attains these bounds, and add the performance to the roofline plot (there should be three dots).

```
1 void computation1(double *x, double *y, double c1, double c2, double c3, int n) {
2     for(int i = 0; i < n; i++) {
3         x[i] = x[i] + c1 + y[i] + c2 + y[i+4] + c3;
4     }
5 }
```

```
1 void computation2(double *x, double *y, double c1, double c2, double c3, int n) {
2     for(int i = 0; i < n; i++) {
3         x[i] = x[i] * c1 * y[i] * c2 * y[i+4] * c3;
4     }
5 }
```

```
1 void computation3(double *x, double *y, double c1, double c2, double c3, int n) {
2     for(int i = 0; i < n; i++) {
3         x[i] = x[i] + c1 * y[i] + c2 * y[i+4] + c3;
4     }
5 }
```

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

(d) For each of the three functions, consider the following modification in the memory access pattern (strided access). We only show `computation1`, but assume the same modification in `computation2` and `computation3`. Array y has an according larger size.

```
1 void computation1(double *x, double *y, double c1, double c2, double c3, int n) {
2     for(int i = 0; i < n; i++) {
3         x[i] = x[i] + c1 + y[7*i] + c2 + y[7*i+4] + c3;
4     }
5 }
```

Follow the same assumptions as in part (b) and derive hard upper bounds on the operational intensity and performance of each function considering this new access pattern (without vectorization). Add the new performance of each function to the roofline plot (three additional dots).

4. *Skinny matrix-matrix multiplication (15 pts)*

Let A be a $m \times m$ square matrix, and B, C be $m \times n$ rectangular matrices. The matrix-matrix multiplication (MMM) operation $C := AB + C$ takes $2m^2n$ flops. Consider the case of a "skinny" matrix where m is large and $n \leq \sqrt{M}$, where M is the number of elements that can fit in cache. Assume that the elements of the matrices are double-precision floating point numbers. Consider also an algorithm that implements MMM for skinny matrices using a one-dimensional partitioning of A and C as follows:

$$C \to \begin{pmatrix} C_1 \\ \hline \vdots \\ \hline C_\alpha \end{pmatrix}, \quad A \to \begin{pmatrix} A_1 \\ \hline \vdots \\ \hline A_\alpha \end{pmatrix}$$

where each C_i is $m_c \times n$ and fits in the cache, each A_i is $m_c \times m$, and α is the number of partitions such as $\alpha = \frac{m}{m_c}$. You can assume that m is multiple of m_c. Then, the algorithm computes each suboperation $C_i \mathrel{+}= A_iB$ by a sequence of outer products using one column of A_i and one row of B at a time. For completeness, we show the pseudo-code to compute one suboperation $C_i \mathrel{+}= A_iB$:

```
void suboperation_Ci(Ci, Ai, B):
    for h = 0 to m-1
        for j = 0 to m_c-1
            for k = 0 to n-1
                Ci[j][k] += Ai[j][h] * B[h][k]
```

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 its execution. Assume a machine with a single cache and answer the following.

(a) Determine a tight lower bound of the I/O cost (in bytes) for the skinny MMM algorithm considering only compulsory misses, i.e., assuming that everything fits in the cache (no capacity misses) and fully associative cache (no conflict misses). Consider reads and writes. Write the lower-bound as a function of m and n.

(b) Assume now that the size of the cache is such that it can only store (and fit exactly) one partition C_i, one row of B (n elements) and one element of A at a time. Determine a tighter lower bound of the I/O cost as a function of m, n and M, considering compulsory and capacity misses. Assume a fully associative cache (no conflict misses).
Hint: C is read and written once to main memory, and each element of A is read once. The only matrix that is read multiple times from main memory due to capacity misses is B.