ECE 759
High Performance Computing for Engineering Applications
Assignment 2
Due Friday 02/17/2025 at 23:59 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment2.pdf (choose one of the formats). Submit all plots (if any) on Canvas. Do not zip your Canvas submission.

All *source files* should be submitted in the HW02 subdirectory on the main branch of your git repo. Please use the name HW02 exactly as shown here (both in terms of capitalization & name). Other names like hw2, hw02, HW2 will not be recognized by the grading scripts. The HW02 subdirectory should have no subdirectories. For this assignment, your HW02 folder should contain task1.cpp, scan.cpp, task2.cpp, convolution.cpp, task3.cpp, and matmul.cpp.

All commands or code must work on *Euler* with no modules loaded unless specified otherwise (post a question on Piazza if the term *module* is confusing). The commands may behave differently on your computer, so be sure to test on *Euler* before you submit.

Please submit clean code. Consider using a formatter like clang-format.

IMPORTANT: Before you begin, copy any provided files from Assignments/HW02 directory of the ECE 759 repo.

Specify your GitHub link here: **https://github.com/FusionSandwich/repo759/HW02**

Note that your link should be of this format: https://github.com/YourGitHubName/repo759/HW02

---

1. a) Implement the scan function in a file called scan.cpp with signature defined as in scan.h. You should write an inclusive scan[1] on your own and not use any library scan functions.

   b) Write a file task1.cpp with a main function which (in this order)

   i) Creates an array of n random float numbers between -1.0 and 1.0. n should be read as the first command line argument as below. ii) Scans the array using your scan function. iii) Prints out the time taken by your scan function in *milliseconds*[2]. iv) Prints the first element of the output scanned array.
   v) Prints the last element of the output scanned array. vi)
   Deallocates memory when necessary.

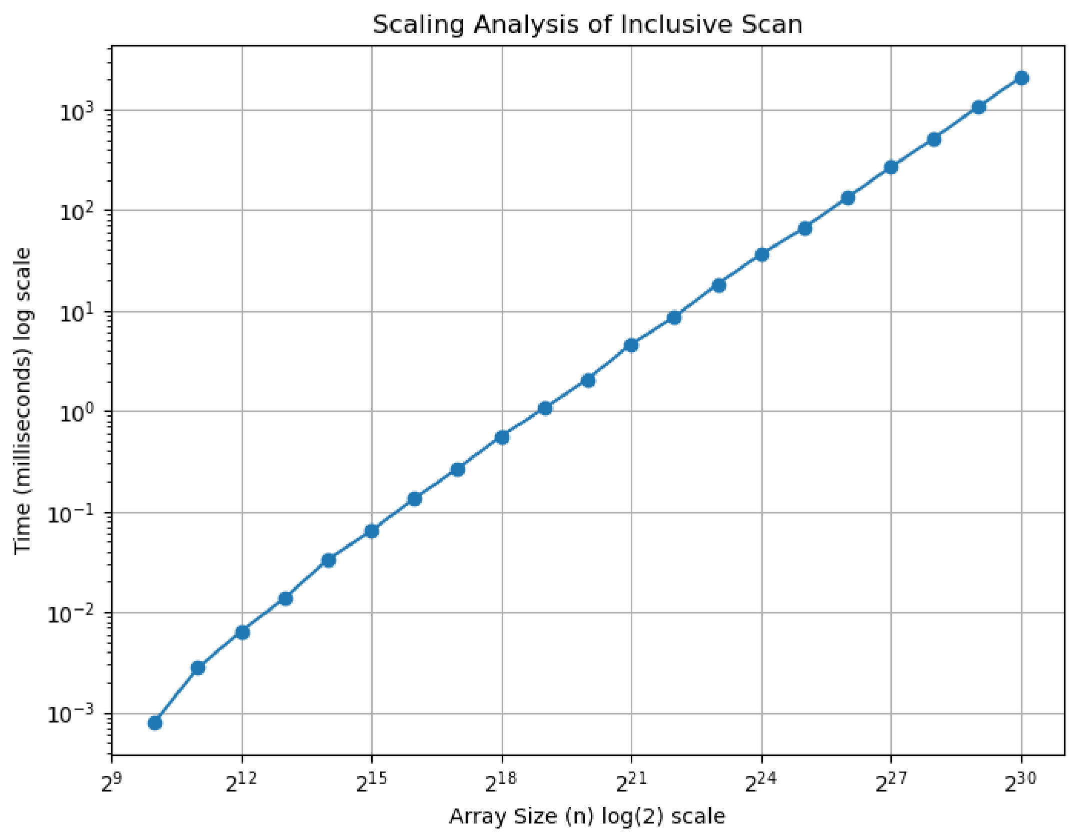
   - Compile command: `g++ scan.cpp task1.cpp -Wall -O3 -std=c++17 -o task1`
   - Run command (n is a positive integer; keep in mind: use Slurm, do not run on head node): `./task1 n`
   - Example expected output (followed by a newline):
     ```
     0.06
     0.65
     87.3
     ```

   c) On an *Euler* compute node (using a Slurm Job), run task1 for each value n = $2^{10}, 2^{11}, \cdots, 2^{30}$ and generate a plot task1.pdf (with axis labels) which plots the time taken by your algorithm as a function of n. This is called a scaling analysis.

---

[1] Given an array $[a_0, a_1, \cdots, a_{n-1}]$, the inclusive scan function produces the array $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \cdots, a_0 + a_1 + \cdots + a_{n-1}]$. (In general, a scan can involve any other associative operation, not only +, like in this task.) [2]Recall the document timing.md.

Feel free to post your scaling analysis plot in case you want to help other colleagues get an idea of what they should obtain at the end of this exercise.
As seen below it's roughly linear with both time as log and array as log(2)



2. Convolutions[2] appear very prominently in image processing and in other fields like the numerical solution of partial differential equations, machine learning, etc.

   a) Implement the convolve function in a file called convolution.cpp with signature as provided in convolution.h.
   The operation that needs to be implemented is as follows:

$$g[x,y] = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \omega[i,\, j] f[x+i-\frac{m-1}{2},\, y+j-\frac{m-1}{2}],$$

   where $x,y = 0,\cdots,n-1$; f is the original image, ω is the mask, and g is the result of the convolution; m is the dimension of the square matrix ω, where $m \geq 1$ is assumed to be an odd number. There are several ways of dealing with the boundaries, but here we will assume that $f[i,j] = 1$ if one of the following two conditions is not satisfied, and $f[i,j] = 0$ if neither of the following two conditions is satisfied:

$$\begin{cases} 0 \leq i < n, \\ 0 \leq j < n. \end{cases}$$

   In other words, we will pad zeros for corners, and pad ones for edges (excluding the corners).

   Example (uses $n = 4$ and $m = 3$):

[2] See here for more on convolutions in image processing, just note that we use a slightly different formulation.

$$f = \begin{bmatrix} 1 & 3 & 4 & 81 & 3 & 4 & 8 \\ 6 & 5 & 2 & 46 & 5 & 2 & 4 \\ 3 & 4 & 6 & 83 & 4 & 6 & 8 \\ 1 & 4 & 5 & 21 & 4 & 5 & 2 \end{bmatrix}$$

$$\omega = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

3 10 10 10

g = 109127 14 11

14 14

5 11 14 4

b) Write a file task2.cpp with a main function which (in this order)

i) Creates an n×nimage matrix (stored in 1D in row-major order) of random float numbers between -10.0 and 10.0. The value of n should be read as the first command line argument.

ii) Creates an m×mmask matrix (stored in 1D in row-major order) of random float numbers between -1.0 and 1.0. The value of m should be read as the second command line argument.

iii) Applies the mask to image using your convolve function. iv) Prints out the time taken by your convolve function in *milliseconds*.

v) Prints the first element of the resulting convolved array. vi) Prints the last element of the resulting convolved array. vii) Deallocates memory when necessary via the delete function.

- Compile command: `g++ convolution.cpp task2.cpp -Wall -O3 -std=c++17 -o task2`
- Run command (where n and m are positive integers and m is an odd number; do not run on the head node): `./task2 n m`
- Example expected output (followed by a newline):

```
0.1
1.5
52.36
```

3. Implement in a file called matmul.cpp the four functions with signatures and descriptions as in matmul.h to produce the matrix product $C = AB$. Pay attention to the argument types defined in matmul.h. For all of the cases, the array C that stores the matrix C should be reported in row-major order.

a) mmul1 should have three for loops: the outer loop sweeps index i through the rows of C, the middle loop sweeps index j through the columns of C, and the innermost loop sweeps index k through; i.e., to carry out, the dot product of the i*th* row A with the j*th* column of B. Inside the innermost loop, you should have a single line of code which increments C_{ij}. Assume that A and B are 1D arrays storing the matrices in row-major order.

b) mmul2 should also have three for loops, but the two innermost loops should be swapped relative to mmul1 (such that, if your original iterators are from outer to inner (i,j,k), then they now become (i,k,j)). That is the only difference between mmul1 and mmul2.

c) mmul3 should also have three for loops, but the outermost loop in mmul1 should become the innermost loop in mmul3, and the other 2 loops do not change their relative positions (such that, if your original iterators are from outer to inner (i,j,k), then they now become (j,k,i)). That is the only difference between mmul1 and mmul3.

d) mmul4 should have the for loops ordered as in mmul1, but this time around A and B are stored as std::vector<double>. That is the only difference between mmul1 and mmul4.

e) Write a program task3.cpp that accomplishes the following:

- generates square matrices A and B of dimension at least 1000×1000 stored in row-major order.
- computes the matrix product $C = AB$ using each of your functions (note that you may have to prepare A and B in different data types so they comply with the function argument types). *Your result stored in matrix C should be the same no matter which function defined at* **??** *through* **??** *above you call*.
- prints the number of rows of your input matrices, and for each mmul function in ascending order, prints the amount of time taken in *milliseconds* and the last element of the resulting C. There should be nine values printed, one per line

▪ Compile command: `g++ task3.cpp matmul.cpp -Wall -O3 -std=c++17 -o task3`

▪ Run command: `./task3`

▪ Sample expected output:

```
1024
1.23
2.365
1.23
2.365
1.23
2.365
1.23
2.365
```

f) In a couple sentences, explain the difference that you see in the times for mmul1, mmul2 and mmul3 *when running on an Euler compute node*. What would explain the performance results you report? Be as specific as possible. Also comment on the difference or similarity you see between mmul1 and mmul4.

The differences in execution times for mmul1, mmul2, and mmul3 stem from how their loop ordering affects memory access patterns, cache utilization, and data localization. In mmul1 (i, j, k), the innermost loop iterates over k, meaning each iteration accesses a new row of B (B[k * n + j]), leading to frequent cache misses since B is stored in row-major order. mmul2 (i, k, j), on the other hand, improves cache efficiency because the innermost loop iterates over j, meaning B's elements are accessed in a more sequential manner (B[k * n + j] varies continuously), reducing cache misses. In contrast, mmul3 (j, k, i) performs the worst because the outermost loop iterates over j, meaning writes to C (C[i * n + j]) are scattered across different rows rather than staying within a cache-friendly block. mmul1 and mmul4 exhibit nearly identical performance since they use the same loop order, with mmul4 relying on std::vector<double>, which remains contiguous in memory. However, mmul1 is likely marginally more memory efficient as it directly uses raw arrays, avoiding the slight overhead associated with std::vector's dynamic memory management.