MPCS HPC WINTER 2021 – HOMEWORK 1

General Instructions

Due date is 2021-01-22 at 5:00pm CST.

Your solution must be pushed to your GitHub Classroom repo by the due date. Alternately, if you're experiencing technical issues with GitHub, you may send a compressed file (.zip or .tar.gz) of your solution to rahaman@cs.uchicago.edu by the due date.

See the rubric on Canvas for point totals. If a solution does not compile, you will be penalized an additional 20% off the total points possible for the respective problem.

PROBLEM 1 – MATRIX-VECTOR MULTIPLICATION

OVERVIEW

In many situations, it's useful to interpret matrix-vector multiplication differently depending on whether the vector is the left or right operand. In this problem, you will implement matrix-vector multiplication such that:

- If the vector is the **left operand**, it is treated as a **row vector**
- If the vector is the **right operand**, it is treated as a **column vector**

That is, you will implement an $\times$ operator that acts like examples below:[1]

$$\boldsymbol{x} = [1 \quad 2 \quad 3]$$
$$\boldsymbol{A} = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix}$$
$$\boldsymbol{x} \times \boldsymbol{A} = [66 \quad 72 \quad 78]$$
$$\boldsymbol{A} \times \boldsymbol{x} = [50 \quad 68 \quad 86]$$

You will then explore how the compiler detects and applies SIMD optimizations in these two cases.

SPECIFICATIONS

In the problem1/ directory, you must have these source files

1. matvec_mul.c (or mat_mul.cpp) and matvec_mul.h
    a. Implementations of your matrix-vector multiplication functions.
    b. If your functions are header-only (such as templates), you may omit the .cpp file.
2. mul_test.c (or mul_test.cpp)
    a. An executable program that runs and verifies a test case for $\boldsymbol{x} \times \boldsymbol{A}$ and $\boldsymbol{A} \times \boldsymbol{x}$.
    b. You may hardcode the values of the matrices and vectors. The values from the Overview section are sufficient.
3. mul_bench.c (or mul_bench.cpp)
    a. An executable program that runs the multiplications and reports timings.
4. CMakeLists.txt
    a. Compiles the executable targets mul_test and mul_bench
5. A report (report.pdf, report.docx, etc.) about your results
6. Any other source or header files for your solution.

Your matrix-vector multiplication functions must be implemented with these features:

1. They must handle 32-bit floats, but they can handle others if you like
2. The result vector must be passed by reference and is assumed to be already allocated by the functions' callers. Do not allocate and return a new vector in the functions.

[1] Incidentally, NumPy implements @, the matrix-multiplication infix operator, to behave like this.

The benchmark program, mul_bench, must do the following:

1. Take one command line argument for n
    a. All should contain 32-bit floats.
2. Compute and measure the wall times of $\boldsymbol{b} = \boldsymbol{x} \times \boldsymbol{A}$, where $\boldsymbol{x}$ and $\boldsymbol{b}$ has size n; and $\boldsymbol{A}$ has size (n, n). All contain 32-float floats.
    a. **Repeat** this computation 4 times and report the **minimum** time out of the 4 trials.
    b. Do not include the time to malloc or free the data structures or print output
    c. For these benchmarks, the values of the vector and matrix elements don't matter. In fact, you can get perfectly valid benchmarks from using all 0's.
    d. If your program does not reference $\boldsymbol{b}$ at all, the compiler might completely omit the matrix-vector multiplications as part of its optimizations. To avoid this, you can simply print a single element from $\boldsymbol{b}$ as a dummy value at the end of your program
3. Repeat (3) for $\boldsymbol{b} = \boldsymbol{A} \times \boldsymbol{x}$

In the report, you must discuss the following. For each question, about 1 – 5 sentences will suffice.

1. Examine and discuss the optimization reports from the compiler. Which significant loops were vectorized and why? Which significant loops were not vectorized and why not?
2. Present a graph of the runtimes from an **optimized** build of mul_bench with n = 256, 512, 1024, 2048, and 4096. Plot the matrix-vector and vector-matrix functions with separate lines on the same graph.

PROBLEM 2 – PARTICLE MOTION

OVERVIEW

In this problem, you will simulate a group of moving particles in 2 dimensions with some simple assumptions: (a) each particle has a different, constant velocity; (b) the particles exert no force on each other. You will explore how different data representations will affect the compiler's ability to vectorize code.

Many applications with multi-dimensional data can be implemented with either an **array-of-structs** or a **struct-of-arrays** representation of the data. For example, to represent an array of particle coordinates, we could use the array-of-structs approach:

```
typedef struct {
  float x, y;
} Coord;

Coord *positions = malloc(n * sizeof(Coord));
```

This is intuitive, but the compiler can sometimes infer SIMD operations more effectively with a struct-of-arrays:

```
typedef struct {
  float *x, *y;
} Coords;

Coords positions;
positions.x = malloc(n * sizeof(float));
positions.y = malloc(n * sizeof(float));
```

The optimal choice is dependent on the particular algorithm and even the capabilities of different compilers.

ALGORITHM

Below is our algorithm for the motion of one particle in 2D. It has the following variables:

- A 2D position $\vec{\boldsymbol{r}}_i$ at time i. The algorithm does not assume units. For example, it could be in meters (m).
- A 2D velocity $\vec{\boldsymbol{u}}$. This could be in meters per second (m/s)
- A discrete time interval dt. This could be in seconds (s).

The values of $\vec{\boldsymbol{r}}_o$, $\vec{\boldsymbol{u}}$, and dt are arbitrary, but the values shown in the algorithm will make nice-looking images. Otherwise, the "algorithm" at each timestep i is simply $\vec{\boldsymbol{r}}_i = \vec{\boldsymbol{r}}_{i-1} + \vec{\boldsymbol{u}}dt$. Note that for multiple particles, you will need additional loop(s) to perform the initialization and timestepping for the all the particles.

ALGORITHM: For 1 particle in 2 dimensions

Let p and q be random numbers in the interval $[0, \pi)$
Let the particle's initial position vector $\overrightarrow{\boldsymbol{r}_0} = \langle p, q \rangle$
Let the particle's velocity vector $\vec{\boldsymbol{u}} = \langle \sin p \cos q\ , -\cos p \sin q \rangle$
Let the time interval $dt = 0.025$

for each timestep i:
$\vec{\boldsymbol{r}}_i = \vec{\boldsymbol{r}}_{i-1} + \vec{\boldsymbol{u}}dt$

WRITING IMAGES

You will also be writing bitmap (.bmp) files[2] to visualize the particles. The functions you need are already provided in problem2/bitmap.c and problem2/bitmap.h; and they are demonstrated in problem2/bitmap_demo.c. Below is some further explanation, but you are not required to know any internals of the file format itself.

In one common bitmap format (which we use here), each pixel is represented with three intensity values: red, green, and blue. Each intensity is an integer in the range [0, 256).[3]

In memory, one pixel can be represented by a struct of three unsigned char. To force byte-alignment in an array of pixels, the struct can be padded with an additional, unused unsigned char:[4]

```
typedef struct {
  unsigned char blue;
  unsigned char green;
  unsigned char red;
  unsigned char reserved;
} RgbQuad;
```

In the bitmap file itself, first there is a header of metadata. Then there is a continuous buffer of RGB values, which is interpreted as a row-major matrix of pixels that map to the image. The metadata is already implemented in the save_bitmap function from problem02/bitmap.h and problem02/bitmap.c. So to use save_bitmap to create a bitmap file, all you need is:

- A pointer to an array of RgbQuad, representing a row-major matrix of pixels
- The dimensions of the matrix
- A filename for the output file

Some hints for working with bitmaps:

- It's convenient to encapsulate your RgbQuad array in a matrix data structure.
- You can find reference tables for RGB color codes in many places. For example: https://www.w3schools.com/colors/colors_picker.asp

SPECIFICATIONS

In the problem2/ directory, you must have the following files:

1. particle_aos.c (or particle_aos.cpp)
    a. Runs and times a particle simulation using an array-of-structs data representation
2. particle_soa.c (or particle_soa.cpp)
    a. Runs and times a particle simulation using a struct-of-arrays data representation

---

[2] See https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-storage

[3] Together, these can represent the same color range as a standard hex color code.

[4] For some compilers, high optimization levels will byte-align structs in arrays. If so, this would implicitly insert the same padding in a struct with only three unsigned char. Using a struct with explicit padding provides more consistent (if sometimes redundant) behavior.

3. CMakeLists.txt
    a. Compiles the executable targets particle_soa and particle_aos
4. The report (report.docx, report.pdf, etc.) about your results
5. Any other source or header files you need

The particle_soa and particle_aos programs must both do the following

1. Takes command line arguments of the following forms
    a. No arguments: uses 512 particles and 128 timesteps
    b. 2 arguments: uses the specified number of particles and number of timesteps, in that order
    c. Any other number of arguments: print a warning message and exit
2. Print a bitmap image of the particles at every timestep.
    a. Put the timestep index somewhere in the filename: for example, particle_aos_000.bmp, particle_aos_001.bmp, particle_aos_002.bmp, etc.
    b. The image size is arbitrary, as it won't affect our performance measurements.
    c. The given values for $\vec{r}_0$ and $\vec{u}$ produce nice images when you plot the ranges $0 < x < \pi$ and $0 < y < \pi$.
    d. This does not have to be performance optimized; we'll exclude from our timings.
3. At the end of the simulation, report the total wall time of the simulation, **excluding** the time it takes to malloc data, free data, write to stdout, and write to file.

The report should discuss the following. For each question, about 1 – 5 sentences will suffice.

1. Compare the optimization reports for the struct-of-arrays and array-of-structs implementations. Were the significant loops vectorized in both cases? Did this match your expectations?[5]
2. Graph the execution times for both implementations using 512 timesteps and 64, 128, 256, and 512 particles. Show the two implementations as separate lines on the same graph.

BONUS (OPTIONAL, NO CREDIT)

Using your bitmap images, you can easily make an animated GIF of your simulation. There are many ways to do it, including online converters. My preferred tool is ImageMagick (https://imagemagick.org). Assuming you have bitmap files name particle_soa_000.bmp, particle_soa_001.bmp, etc., you can use ImageMagick's convert tool in the terminal like this:

```
$ convert -delay 5 -loop 0 particle_soa*.bmp particle_soa.gif
```

Using the given values of $\vec{\boldsymbol{r}}_o$, $\vec{\boldsymbol{u}}$, and dt; and the ranges $0 < x < \pi$ and $0 < y < \pi$, you can observe the particles moving in a counter-clockwise circle around the center of the image. For example:

https://www.dropbox.com/s/lkmzr31exvfwbrx/particle_soa.gif?dl=0

[5] This varies greatly by compiler. For example, on my Mac, GCC was able to vectorize both implementations, but Apple Clang only vectorized one version. Whatever your compiler does, report it.)