CSC4005 Project 1 Report

How to compile the programs

```
cd /path/to/project1

mkdir build && cd build

cmake ..

make -j4
```

How to execute the programs

```
cd /path/to/project1/build

sbatch ./../src/scripts/sbatch_PartA.sh

sbatch ./../src/scripts/sbatch_PartB.sh
```

How does each parallel programming model do computation in parallel

SIMD (Single Instruction, Multiple Data):

It performs a single operation on multiple data points simultaneously.

MPI (Message Passing Interface):

It uses a message-passing paradigm where individual processes communicate with each other by explicitly sending and receiving messages.

Pthread (POSIX threads):

It allows multiple threads to run concurrently within a single process. Shared memory is used for communication, and synchronization primitives like mutexes are often required.

OpenMP (Open Multi-Processing):

Uses compiler directives to automatically parallelize code. It allows shared-memory parallelism, where threads can be forked to perform tasks concurrently.

CUDA (Compute Unified Device Architecture):

NVIDIA's parallel computing platform and API. CUDA allows developers to utilize NVIDIA GPUs for parallel processing. It uses a hierarchy of thread blocks and grids.

OpenACC (Open Accelerators):

A directive-based programming model designed to simplify the parallelizing of code on heterogeneous systems (like systems that combine CPUs and GPUs).

What are the similarities and differences between each parallel programming model

Similarities:

Performance Objective: All aim to enhance computational performance by executing tasks in parallel.

Concurrency: Each model introduces mechanisms to allow multiple tasks to run at the same time.

Differences:

SIMD: Single instruction operates on multiple data points simultaneously. Emphasizes DLP (Data-Level Parallelism), as it performs the same operation on multiple data simultaneously.

MPI: Message-passing paradigm in distributed systems. Focuses on TLP (Thread-Level Parallelism) across distinct processes.

Pthread: Multi-threading within a single process. Exploits TLP within a shared memory context.

OpenMP: Compiler directives for shared-memory parallelism. Primarily leverages TLP but can also tap into DLP when vectorizing loops.

CUDA: GPU parallelism with a specific hierarchy (thread, block, grid). Exploits both DLP and TLP. Massive DLP due to the simultaneous execution of the same instruction over different data elements in a GPU and TLP because of the concurrently executing threads.

OpenACC: Directive-based for heterogeneous systems. Harnesses both DLP and TLP depending on how directives are used and the target accelerator.

What kinds of optimizations have you tried to speed up your parallel program for PartB, and how does them work?

SIMD:

I padded the original image to aid in applying the filter. In this case the image to filter maintains a width that is a multiple of 8. Then the SIMD operations are perfectly aligned with the image's data layout. This guarantees that there's no "spill" over the boundary when loading 8 adjacent values, ensuring that the SIMD operations are always fully utilized without any need for additional boundary checks or special handling for the image's edge pixels.

Instead of accessing the interleaved RGB values directly, I separated them into three color channels. This ensures better memory access patterns. In this case, I can load 8 continuous red/blue/green pixels, which is significantly efficient for SIMD. In addition, by processing 8 adjacent values at once, the code makes efficient use of spatial locality in memory.

I used AVX2, which can operate on 256-bit wide data, allowing operations on multiple data points simultaneously. For every iteration, 8 continuous values (red/green/blue) are processed, providing a speed-up. In total there are 9 iterations, corresponding to the convolution process. After the last iteration, the value of 8 continuous pixels of the filtered image are computed simultaneously.

MPI:

I've partitioned the convolution operation among different MPI processes, with each process responsible for processing a specific chunk of rows. By segmenting the image into sections and distributing them across multiple processes, I achieve parallelism at a high level, accelerating the overall execution.

I've ensured the rows of the image are divided in a balanced manner among the processes. If there are 11 pixels and 3 tasks, the division is 4, 4, and 3, making sure each process gets roughly an equal workload. This balanced division ensures that no particular process becomes a bottleneck. Guaranteeing that every process is equipped with a roughly equal workload negates situations where certain processes wrap up early and remain idle, leading to inefficiencies.

I replace the original one for loop by two for loops, which can reduce the calculation because in this case not every pixel need to recalculate the position in the image. For each row, we can calculate the row's position, and then the pixel's position can be calculated by adding an offset.

By isolating the red, green, and blue channels into their arrays, I've optimized memory access when applying the filter, which enhances cache performance.

I have removed the inner two nested for loops. I rewrote it in a non-loop format. Every loop carries some overhead – initializing loop counters, branching for loop conditions, and incrementing loop counters. By eliminating the nested loops, I've removed this overhead, making the code potentially faster. In addition, I used arrays to store the positions of the pixels being processed, reducing the number of repetitive calculations needed. By calculating the index only once and reusing them, I've reduced the number of arithmetic operations, improving speed.

Pthread :

I've utilized the Pthreads library, which allows for the creation of multiple threads to operate on the image concurrently. This means that multiple portions of the image can be processed simultaneously, taking full advantage of multi-core processors.

The image data is divided into chunks based on the number of threads. Each thread processes its own chunk of the image. By breaking the task into smaller chunks, I ensure that each thread has an equal amount of work, leading to a balanced load across all threads.

By isolating the red, green, and blue channels into their arrays, I've optimized memory access when applying the filter, which enhances cache performance.

I have removed the inner two nested for loops. I rewrote it in a non-loop format. Every loop carries some overhead – initializing loop counters, branching for loop conditions, and incrementing loop counters. By eliminating the nested loops, I've removed this overhead, making the code potentially faster. In addition, I used arrays to store the positions of the pixels being processed, reducing the number of repetitive calculations needed. By calculating the index only once and reusing them, I've reduced the number of arithmetic operations, improving speed.

OpenMP :

I leveraged the OpenMP library to enable parallel processing. OpenMP provides a simple and flexible way to harness the power of multi-core processors. By incorporating `#pragma` directives, I can easily dictate which sections of the code should run in parallel.

OpenMP allows for easy data partitioning. I divided the image into chunks based on the number of threads specified. Each thread then works on its designated chunk of the image. This ensures that each thread has a similar workload and all threads can run concurrently.

I used the `#pragma omp for` directive to automatically divide the for-loop iterations among the threads. This ensures a balanced distribution of work, where each thread processes roughly an equal number of pixels.

By isolating the red, green, and blue channels into their arrays, I've optimized memory access when applying the filter, which enhances cache performance.

I have removed the inner two nested for loops. I rewrote it in a non-loop format. Every loop carries some overhead – initializing loop counters, branching for loop conditions, and incrementing loop counters. By eliminating the nested loops, I've removed this overhead, making the code potentially faster. In addition, I used arrays to store the positions of the pixels being processed, reducing the number of repetitive calculations needed. By calculating the index only once and reusing them, I've reduced the number of arithmetic operations, improving speed.

CUDA:

I use NVIDIA's CUDA platform, which enables general-purpose parallel programming on GPUs. The ability of GPUs to handle thousands of threads concurrently makes them highly efficient for tasks like image processing where operations on pixels can be parallelized.

The kernel processes the image pixel by pixel, making use of the GPU's ability to execute many threads in parallel. By computing pixel values in parallel, I could achieve a significant speedup compared to sequential execution.

I have removed the inner two nested for loops. I rewrote it in a non-loop format. Every loop carries some overhead – initializing loop counters, branching for loop conditions, and incrementing loop counters. By eliminating the nested loops, I've removed this overhead, making the code potentially faster. In addition, I used arrays to store the positions of the pixels being processed, reducing the number of repetitive calculations needed. By calculating the index only once and reusing them, I've reduced the number of arithmetic operations, improving speed.

OpenACC :

With `#pragma acc parallel` and `#pragma acc loop`, I specified that the loops for image processing are to be executed in parallel on the GPU. OpenACC will handle the decomposition of the loops and distribute the iterations among GPU threads.

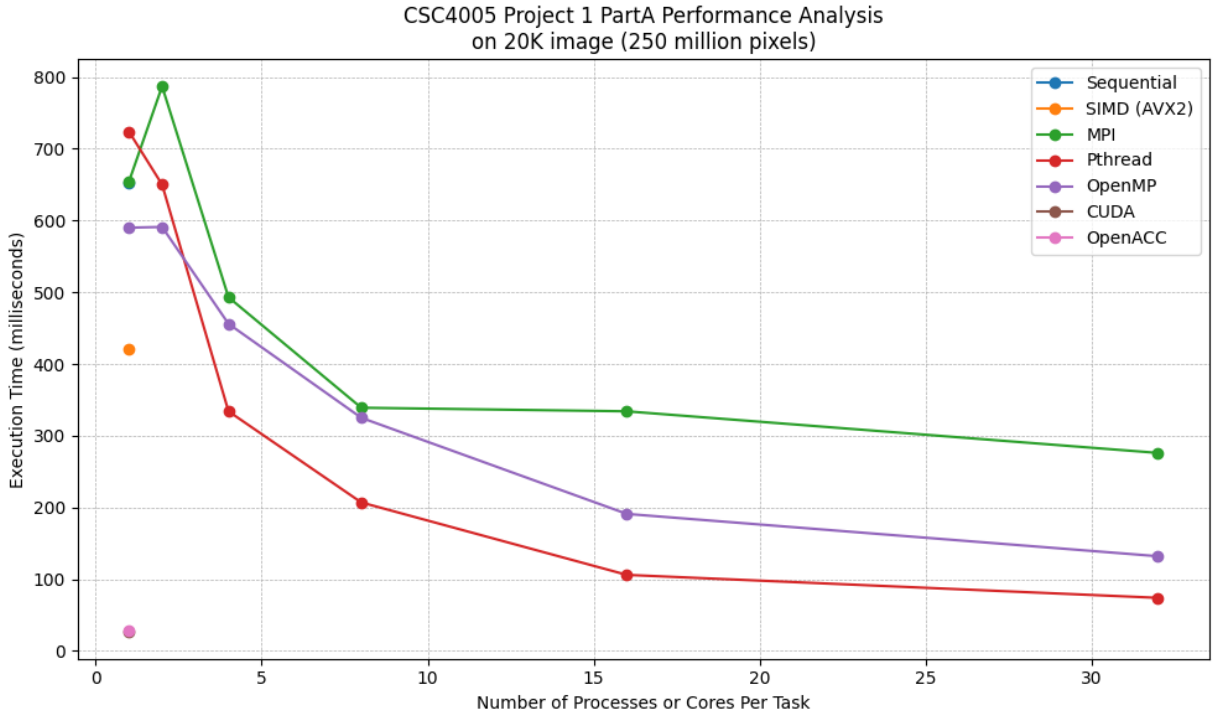
I chose a block size of 1024, allowing each GPU block to have that many threads. The number of gangs (blocks) is calculated based on the image size, ensuring optimal workload distribution across the GPU.

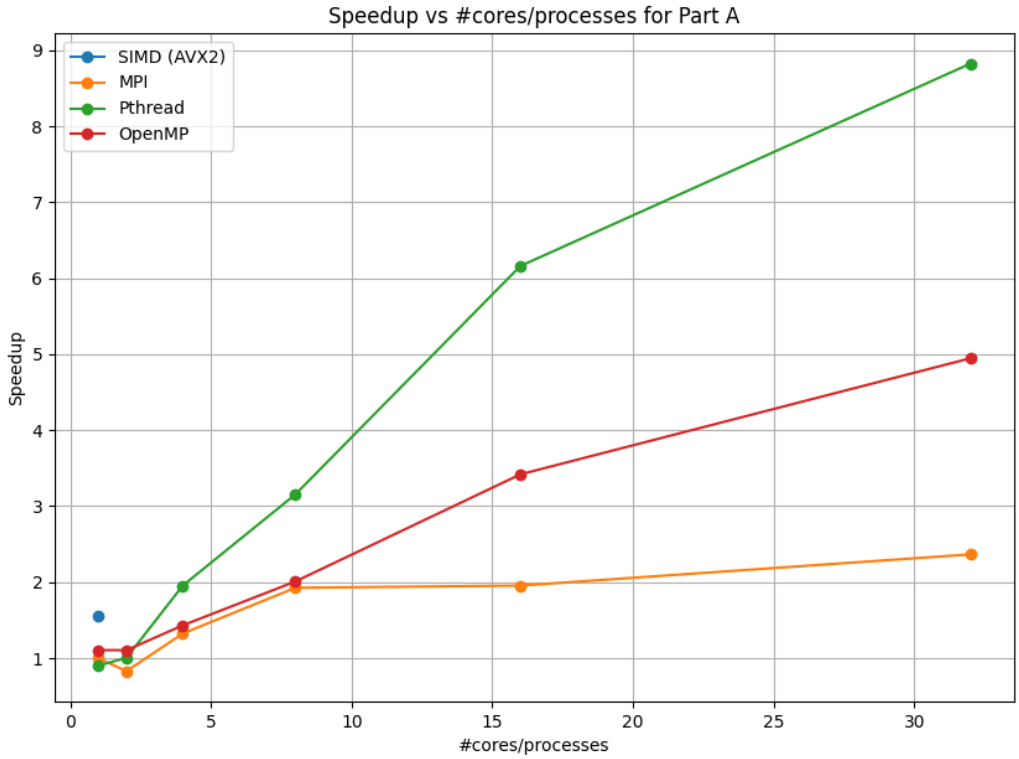
I have removed the inner two nested for loops. I rewrote it in a non-loop format. Every loop carries some overhead – initializing loop counters, branching for loop conditions, and incrementing loop counters. By eliminating the nested loops, I've removed this overhead, making the code potentially faster. In addition, I used arrays to store the positions of the pixels being processed, reducing the number of repetitive calculations needed. By calculating the index only once and reusing them, I've reduced the number of arithmetic operations, improving speed.

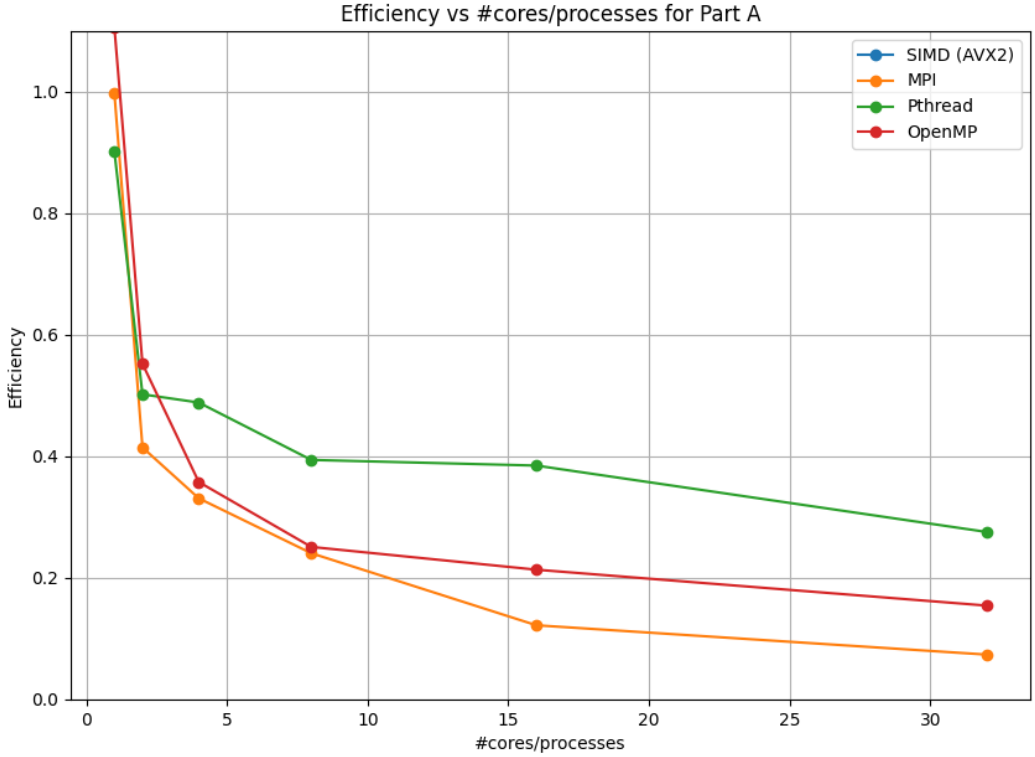
Performance

Part A

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	653	421	654	724	590	27.1173	28
2	N/A	N/A	787	650	591	N/A	N/A
4	N/A	N/A	493	334	456	N/A	N/A
8	N/A	N/A	339	207	325	N/A	N/A
16	N/A	N/A	334	106	191	N/A	N/A
32	N/A	N/A	276	74	132	N/A	N/A

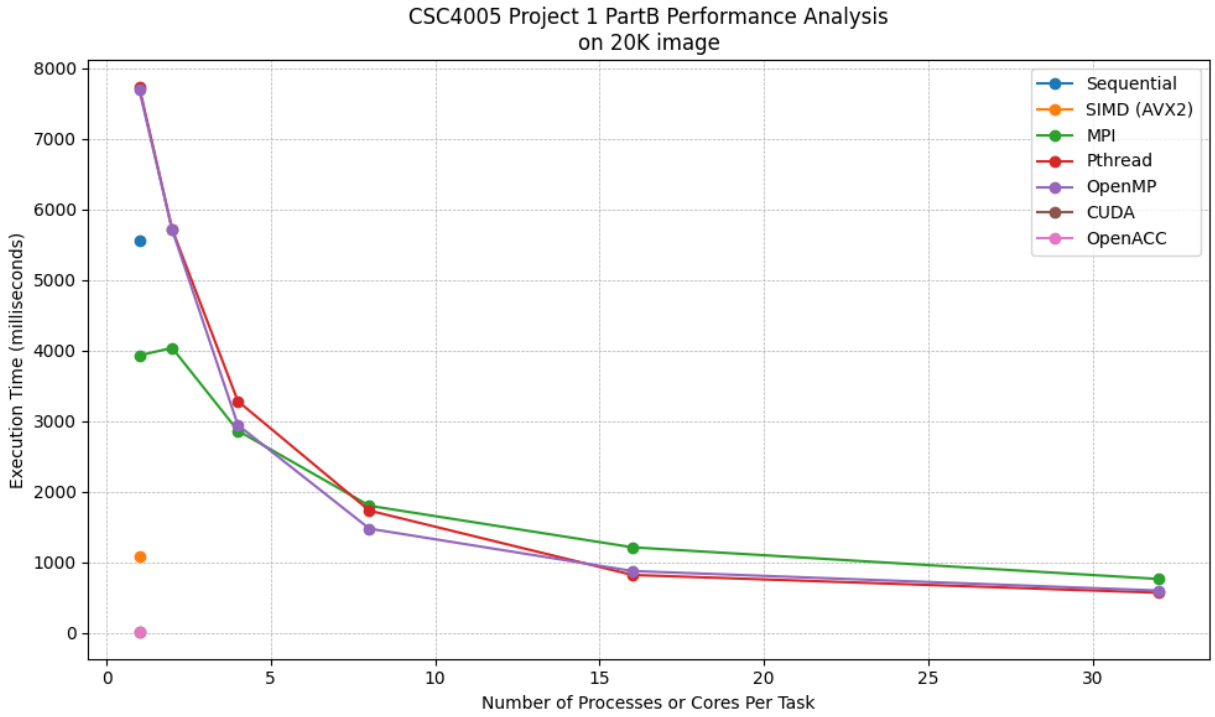


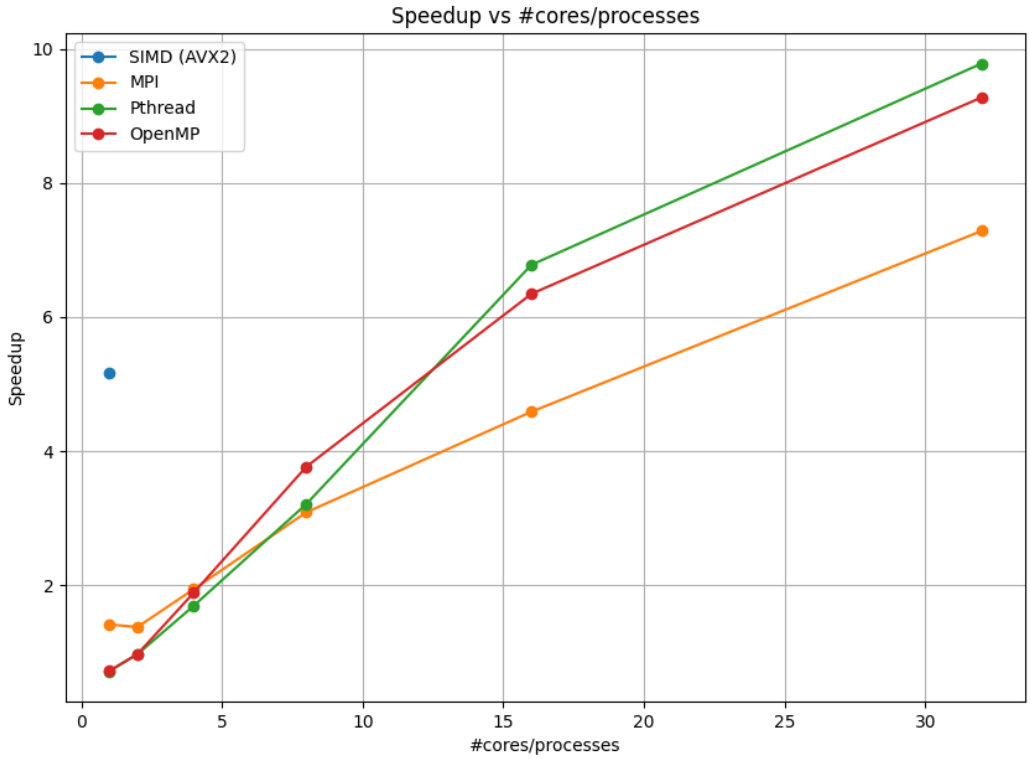

Speedup vs #cores/processes for Part A
SIMD (AVX2)
MPI
Pthread
OpenMP
Speedup
#cores/processes
0
5
10
15
20
25
30
1
2
3
4
5
6
7
8
9

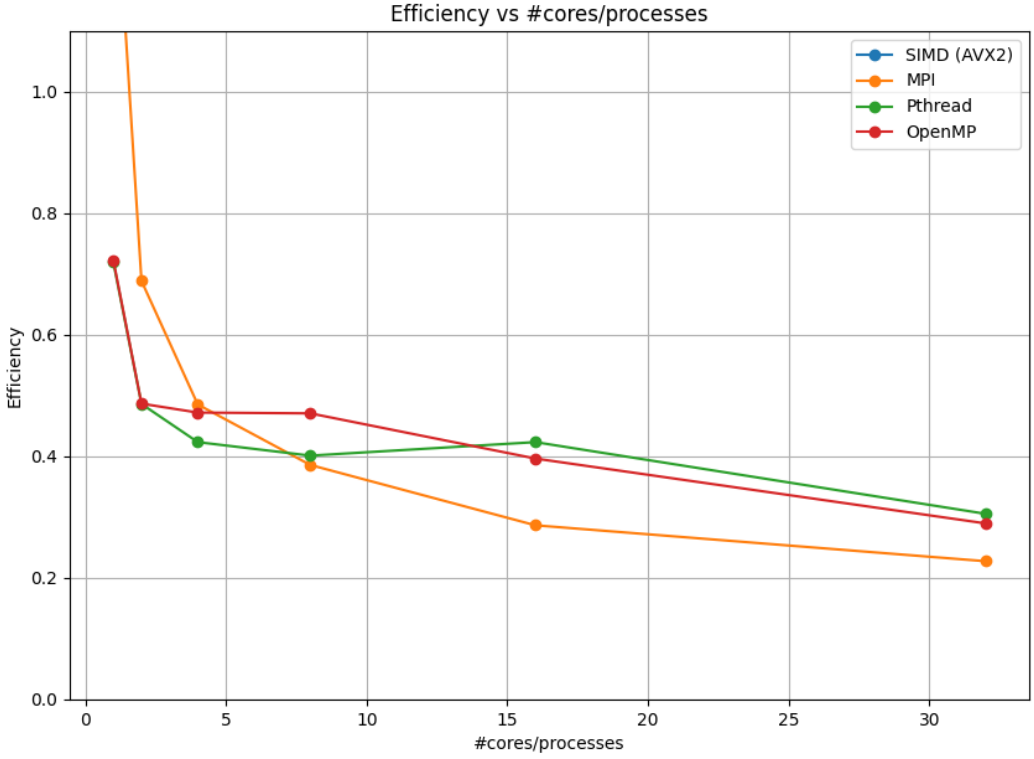

Efficiency vs #cores/processes for Part A
SIMD (AVX2)
MPI
Pthread
OpenMP
Efficiency
#cores/processes
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0

Part B

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	5565	1077	3931	7733	7699	18.2856	19
2	N/A	N/A	4039	5720	5714	N/A	N/A
4	N/A	N/A	2865	3284	2946	N/A	N/A
8	N/A	N/A	1802	1733	1477	N/A	N/A
16	N/A	N/A	1213	821	877	N/A	N/A
32	N/A	N/A	764	569	600	N/A	N/A




Speedup vs #cores/processes
SIMD (AVX2)
MPI
Pthread
OpenMP
Speedup
#cores/processes
0
5
10
15
20
25
30
2
4
6
8
10


Efficiency vs #cores/processes
SIMD (AVX2)
MPI
Pthread
OpenMP
Efficiency
#cores/processes
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0

Observations:

1. **Sequential vs Parallel**:
    - For both Part A and Part B, as expected, the sequential times are significantly higher than most parallel implementations when run on multiple processes/cores.
    - Even for a single process/core, some parallel methods (like SIMD and CUDA) significantly outperform the sequential method, likely due to architecture-specific optimizations.
2. **SIMD (AVX2)**:
    - SIMD shows a significant speed-up compared to the sequential method for both parts, which is expected as SIMD allows multiple data points to be processed simultaneously.
3. **Pthread & OpenMP**:
    - As the number of processes/cores increases, the performance of Pthread and OpenMP improves substantially. This is indicative of effective thread-based parallelization.
    - Pthread seems to have a sharper drop in time than OpenMP, especially as the number of cores increases.
4. **MPI**:
    - The MPI times decrease as the number of processes increases, which is expected as the work is distributed across more nodes.
    - However, for two processes, the time seems anomalously high. This might be due to communication overhead or inefficiencies in splitting the workload.
5. **CUDA & OpenACC**:
    - Both CUDA and OpenACC show significant speed-ups for single-core processing. They utilize the parallel processing capabilities of GPUs effectively.
    - The times for Part B are slightly lower than for Part A, which might indicate that the GPU is better optimized for the filter operation compared to RGB-to-grayscale.

Differences between Part A and Part B:

1. Nature of Operations:
    - RGB-to-grayscale conversion (Part A) is a relatively simpler operation where each RGB pixel value is converted to a single grayscale value.
    - Image filtering (Part B), especially with an equal weight filter, involves multiple pixel values contributing to a single output pixel, which inherently requires more computations.
2. Computational Intensity:
    - The times for Part B are generally higher across all methods, indicating that the soften filter operation is more computationally intensive.
3. Optimization Differences:
    - The smaller difference in times for GPU-based methods (CUDA & OpenACC) between the two operations suggests that the GPU is potentially better optimized for filtering operations compared to simpler conversions.