CSC4005 Project 1 ID:120090712

Part A

I. How to execute the program:

On the clusters, I have followed the instruction in the Github.

```
cd /path/to/project1
mkdir build && cd build

cmake ..
make -j4
cd /path/to/120090712-project1
sbatch ./src/scripts/sbatch_PartA.sh
```

II. Result:

Using Lena jpg as an example:

---------->

Here are results of execution time of the A part programs:

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	640	424	639	714	593	27.1296	28
2	N/A	N/A	790	644	564	N/A	N/A
4	N/A	N/A	503	341	450	N/A	N/A
8	N/A	N/A	366	182	279	N/A	N/A
16	N/A	N/A	289	99	181	N/A	N/A
32	N/A	N/A	264	79	133	N/A	N/A

Table 1. Performance measured as execution time in milliseconds

Here is the line chart of the results of the A part programs:

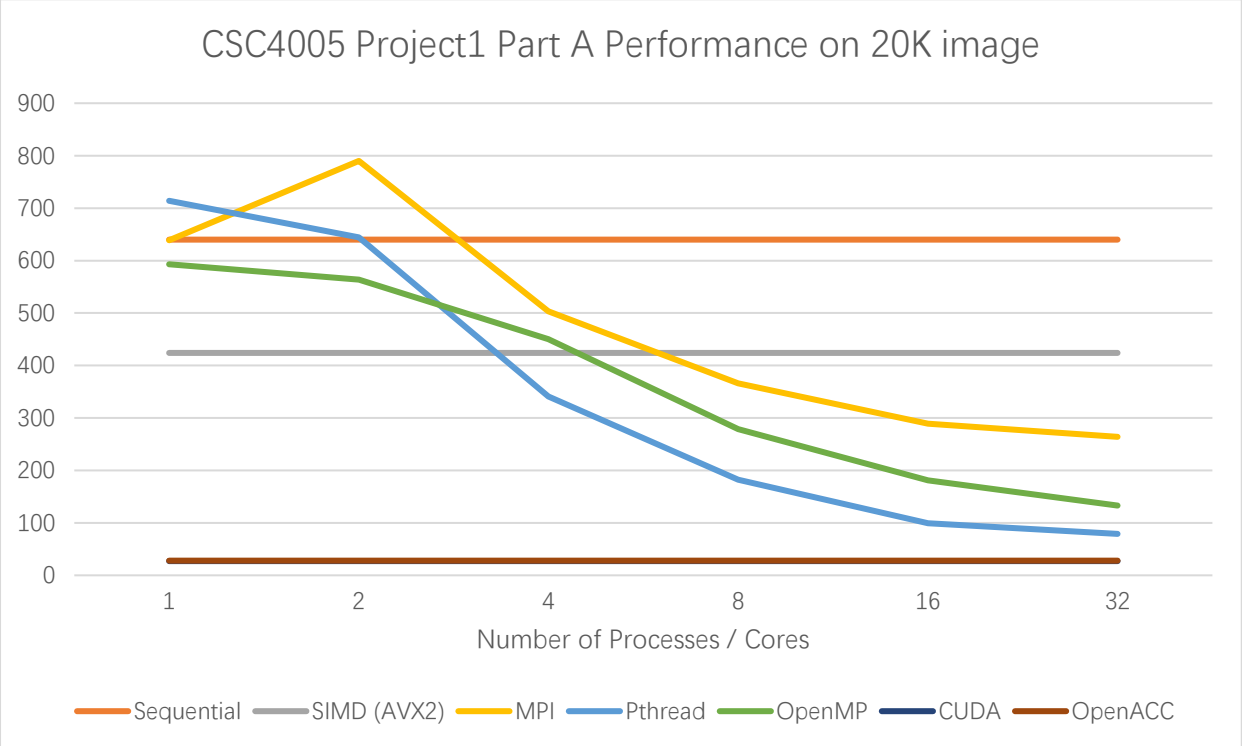


Figure 1. Changes of *performance measured as execution time in milliseconds for Part A*

III. Analysis:

Speedup:

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	1	1	1	1	1	1	1
2	N/A	N/A	0.808861	1.108696	1.051418	N/A	N/A
4	N/A	N/A	1.270378	2.093842	1.317778	N/A	N/A
8	N/A	N/A	1.745902	3.923077	2.125448	N/A	N/A
16	N/A	N/A	2.211073	7.212121	3.276243	N/A	N/A
32	N/A	N/A	2.420455	9.037975	4.458647	N/A	N/A

Table 2. Speedups for Part A

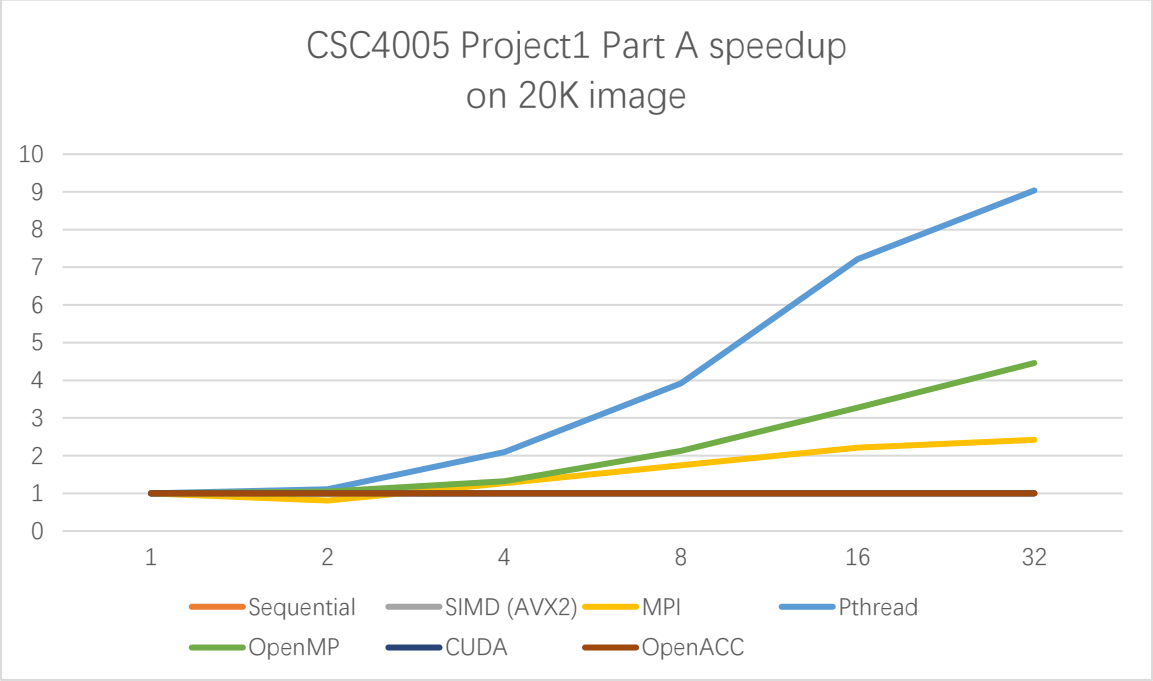


Figure 2. Changes of *Speedups for Part A*

Efficiency:

Number of Processes / Cores	**Sequential**	**SIMD (AVX2)**	**MPI**	**Pthread**	**OpenMP**	**CUDA**	**OpenACC**
1	1	1	1	1	1	1	1
2	N/A	N/A	0.40443	0.554348	0.525709	N/A	N/A
4	N/A	N/A	0.317594	0.52346	0.329444	N/A	N/A
8	N/A	N/A	0.218238	0.490385	0.265681	N/A	N/A
16	N/A	N/A	0.138192	0.450758	0.204765	N/A	N/A
32	N/A	N/A	0.075639	0.282437	0.139333	N/A	N/A

Table 3. Efficiencies for Part A

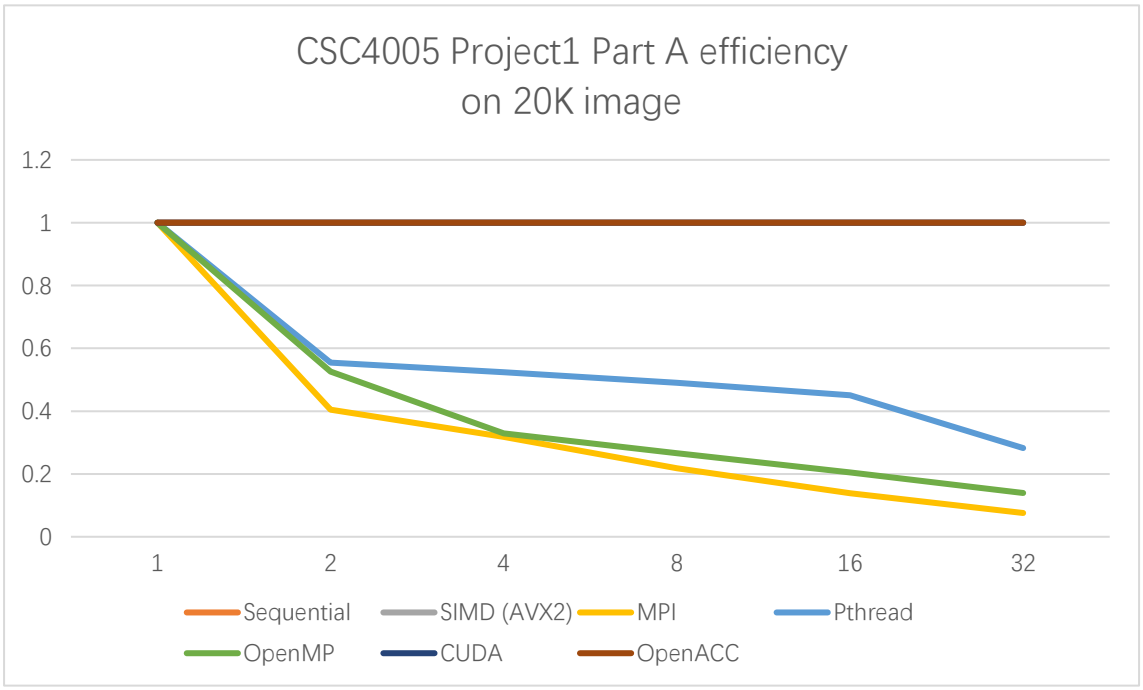


Figure 3. Changes of *Efficiencies for Part A*

Firstly, I have observed that GPUs have significantly greater parallel computing capabilities compared to CPUs. Secondly, for most CPU programs, as the number of processors increases, the execution time decreases. However, this trend may not be evident at the beginning, such as in MPI programs where the time increases when transitioning from one to two processors. This can be attributed to the communication overhead between nodes. The same applies to other CPU parallel computing methods. Additionally, due to the overhead involved in communication and thread creation, the performance gains from adding more processors (efficiency) diminish as the number of processors increases.

Part B

I. How to execute the program (You can find the readme in my source code zip):

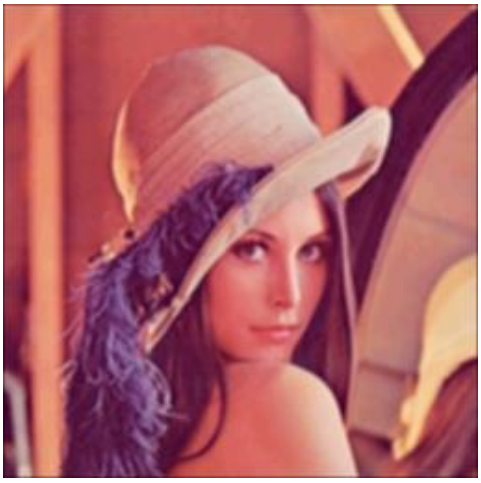
How to rum my programs for PartB on the cluster(similar with PartA):

1. Upload my source code to the cluster.
2. The path of my source code is "120090712-project1". Then go to this path, make sure the "nvcc/pgc++/gcc/g++" commands is availble.
3. Upload the RGB-JPG file in to the "120090712-project1/images", The initial name of the RGB file is 20K-RGB.jpg, make sure the program can find the JPEG.
4. Change the "current path" in the sbatch_PartB.sh, important!!!!!
5. Then, run the following command: I."mkdir build && cd build"; II."cmake .."; III."make -j4"; IV."cd .. && sbatch ./src/scripts/sbatch_PartB.sh".
6. Wait for the result.

II. Results:

Using Lena jpg as an example:

---------->



Here are results of execution time of my B part programs:

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	3841	905	4236	6036	5390	27.7285	25
2	N/A	N/A	4136	5419	5224	N/A	N/A
4	N/A	N/A	2271	2736	2651	N/A	N/A
8	N/A	N/A	1249	1385	1390	N/A	N/A
16	N/A	N/A	843	806	799	N/A	N/A
32	N/A	N/A	641	454	436	N/A	N/A

Table 4. Performance measured as execution time in milliseconds

Here is the line chart of the results of the B part programs:

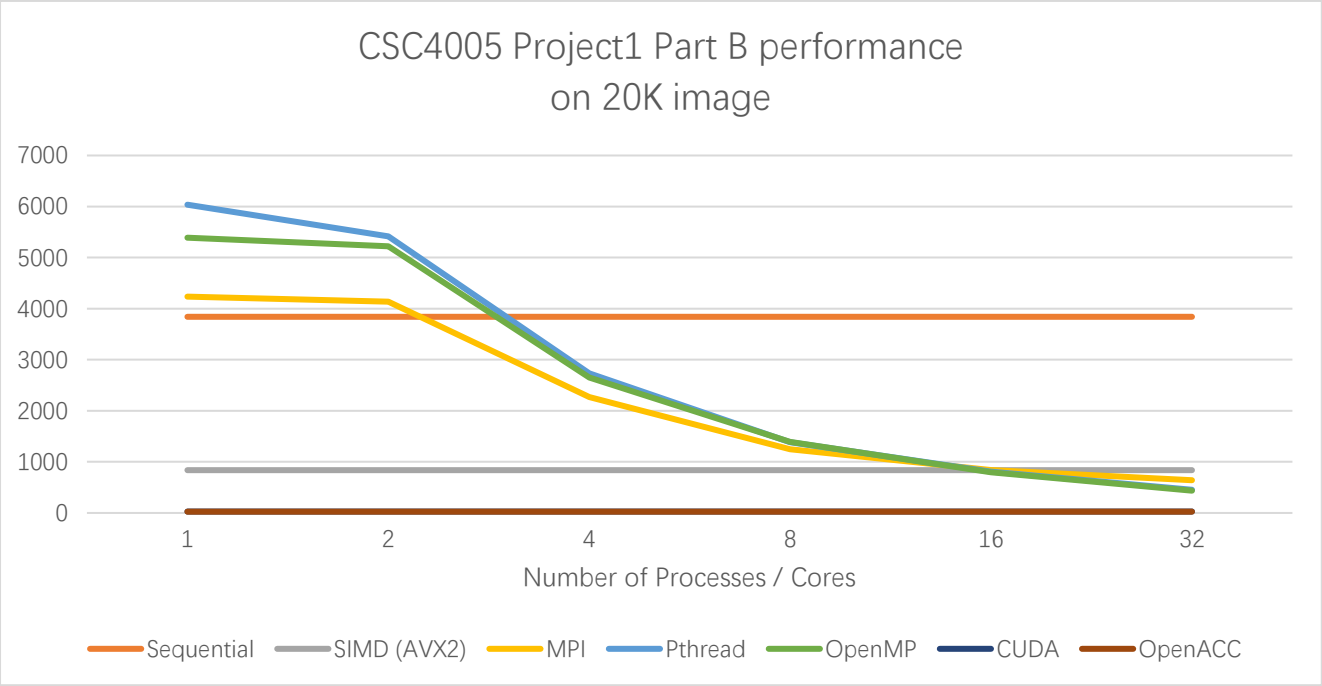


Figure 4. Changes of *performance measured as execution time in milliseconds for Part B programs*

IV. Some basic information of my programs:

1. The CPU programs I wrote is available for modifying the number of the processes/cores. You can modify it at the “sbatch_PartB.sh” file.
2. The filter in every program can be set manually, the definition of the filter is in the previous part of the code, you can change the filter directly.
3. There will be seven new JPG files, the name will be the "20K-Smooth-xxx.jpg", xxx is the six parallel programming implementations and one sequential method.

V. How code solves the task using parallel computing:

(Implemented in MPI, PTHREAD, OPENMP)

When allocating tasks to multiple threads/cores in my code, the input JPEG data is represented as a char array where the index of a pixel can be obtained by multiplying its row number with the column number. Therefore, the pixels in each row are contiguous. As a result, I allocate tasks based on the number of rows, assigning each thread/core to process a specific range of rows (where adjacent rows are also assigned to the

same thread/core). By adopting this allocation approach, I aim to ensure data continuity as much as possible.

(Implemented in SIMD)

In my SIMD code, when iterating through the image, the code reads 8 pixels of data (divided into three color components) at once, which significantly improves efficiency compared to linear processing. However, this method can cause incorrect values for boundary pixels. And I choose to ignore the boundary pixels.

(Implemented in OPENMP)

The code divides the task among multiple processes (executors) to parallelize the computation. The master executor (rank 0) applies the filter to a portion of the image and receives the filtered contents from the slave executors. The slave executors (ranks > 0) also apply the filter to their respective portions of the image and send the filtered contents back to the master executor. Finally, the master executor writes the filtered contents to the output JPEG file.

(Implemented in CUDA)

The image processing is performed on the GPU using CUDA kernels. The main steps involve allocating memory on the host (CPU) and device (GPU), transferring the input image data from the host to the device, applying the 3x3 low-pass filter using a CUDA kernel function, and transferring the output data back from the device to the host. Finally, the filtered image is written to the output JPEG file.

(Implemented in OPENACC)

The image processing task is parallelized using OpenACC loop directives to distribute the computations across multiple cores or accelerators. The code allocates memory for the filtered image and the input buffer, and then uses OpenACC data directives to manage the data transfer between the host and the device. The main computation is performed within a parallel region using OpenACC parallel directives, with nested loop directives to apply the filter to each pixel of the image. Finally, the filtered image is written to the output JPEG file.

VI. Similarities and Differences between these parallel programming:

MPI, OpenMP, and Pthreads enable parallelism at the process or thread level, targeting distributed and shared memory systems respectively.

SIMD enables data-level parallelism by executing the same instruction on multiple data elements simultaneously.

CUDA and OpenACC target accelerators like GPUs, enabling massive data parallelism by offloading computations to the accelerator. Each of these programming models or libraries has its own strengths and is suitable for different types of parallel computing scenarios.

How I optimized my programs to get the better performance than the baseline.

I. For the compelling:

I use O3 instead of O3, O3 aims to improve the efficiency of parallel computing compared to O2. It achieves this by applying advanced optimization techniques specifically targeted at parallel code. These optimizations include automatic vectorization, loop unrolling, and software pipelining. Loop unrolling is a technique where the loop iterations are reduced by executing multiple iterations in a single loop iteration. This reduces the overhead of loop control and improves the efficiency of parallel execution.

Software pipelining is a technique that overlaps the execution of loop iterations to exploit instruction-level parallelism. It reorders and schedules instructions in such a way that the processor's pipeline is fully utilized, minimizing pipeline stalls and improving overall parallel execution efficiency.

By applying these optimizations, O3 can effectively parallelize code, distribute workload across multiple cores or processors, and exploit the capabilities of modern hardware. This results in improved parallel computing efficiency compared to O2, which does not have as extensive optimization techniques targeting parallelism.

Basically, for some CPU parallel code, O3 can improve its efficiency by about 10% than O2.

II. Optimized filter method:

(Implemented in SEQUENTIAL, MPI, PTHREAD, OPENMP)

Initially, the filtering process involved accessing the surrounding 3x3 pixel values for each pixel and calculating the weighted average to obtain the new pixel value. This approach required accessing each pixel's surrounding values nine times, resulting in multiple memory accesses and significant time consumption.

In my new method, I modified the filtering process by extracting the pixel value only once for each pixel. Then, I applied the weighted value to the surrounding pixels that rely on this pixel value. This means that during the traversal of each pixel, it is only necessary to extract the corresponding value from memory once. The only increase in operations occurs during the storage step. However, by storing the processed values in variables that are more efficiently accessed, a considerable amount of time can be saved. As a result, my new method can improve efficiency by approximately 30-40%.

An intuitive display:

Original method:

A B C
D E F → D E F
G H I G H I

For pixel E, the original method is getting the value of the A B C D E F G H I pixels and calculate the weighted Sum, and store it to the E.

My method:

A B C A B C
D E F → D E F
G H I G H I

For pixel E, the original method is getting the value of the E pixel and calculate the weighted value, and store it to the A B C D E F G H I.

III. Less memory access:

To improve the efficiency of my code, I utilize the fact that jpg image data is stored as a char array, with each pixel represented by three consecutive chars. Additionally, the pixels in a row are also consecutive. Accessing each color component individually would trigger multiple memory accesses. Therefore, I employ techniques such as "memcpy" and others to extract multiple required data at once, reducing the number of memory accesses needed.

Analysis (compared with Part A):

Speedup:

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	1	1	1	1	1	1	1
2	1	1	1.024178	1.113859	1.031776	1	1
4	1	1	1.865258	2.20614	2.033195	1	1
8	1	1	3.391513	4.358123	3.877698	1	1
16	1	1	5.024911	7.488834	6.745932	1	1
32	1	1	6.608424	13.29515	12.36239	1	1

Table 5. Speedups for Part B

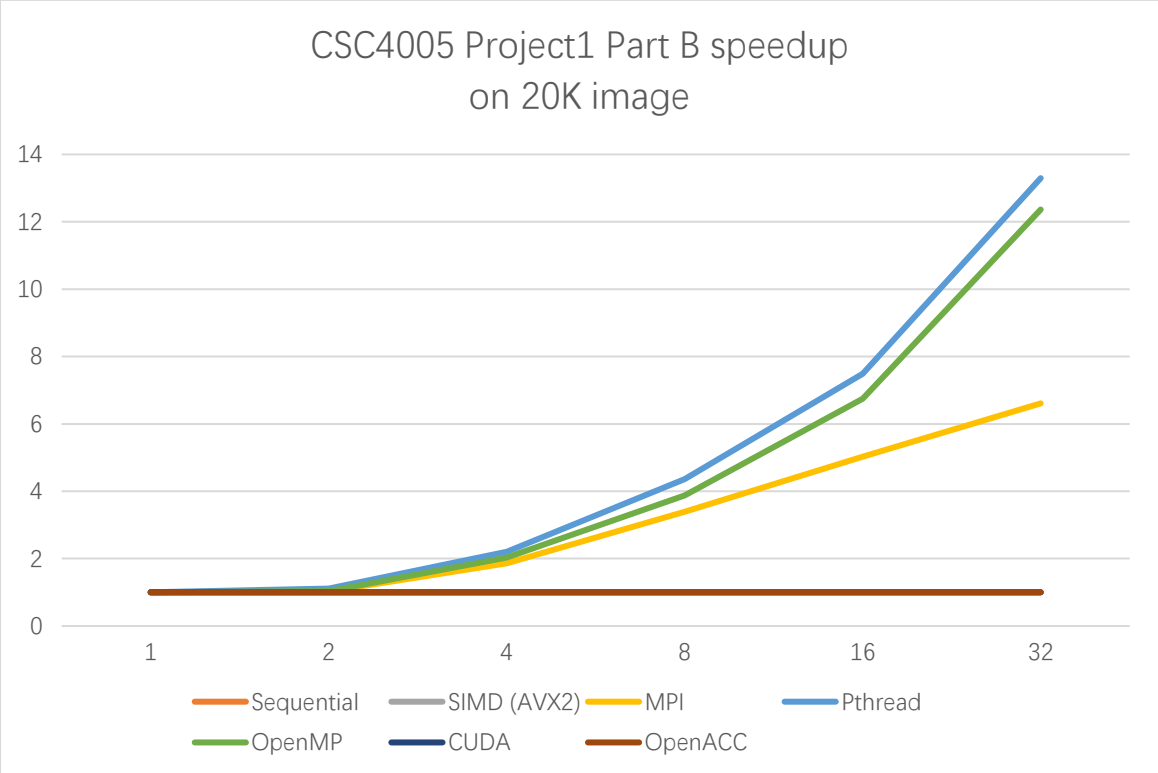


Figure 5. Changes of *Speedups for Part B*

Efficiency:

Number of Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	1	1	1	1	1	1	1
2	1	1	0.512089	0.556929	0.515888	1	1
4	1	1	0.466314	0.551535	0.508299	1	1
8	1	1	0.423939	0.544765	0.484712	1	1
16	1	1	0.314057	0.468052	0.421621	1	1
32	1	1	0.206513	0.415474	0.386325	1	1

Table 6. Efficiencies for Part B

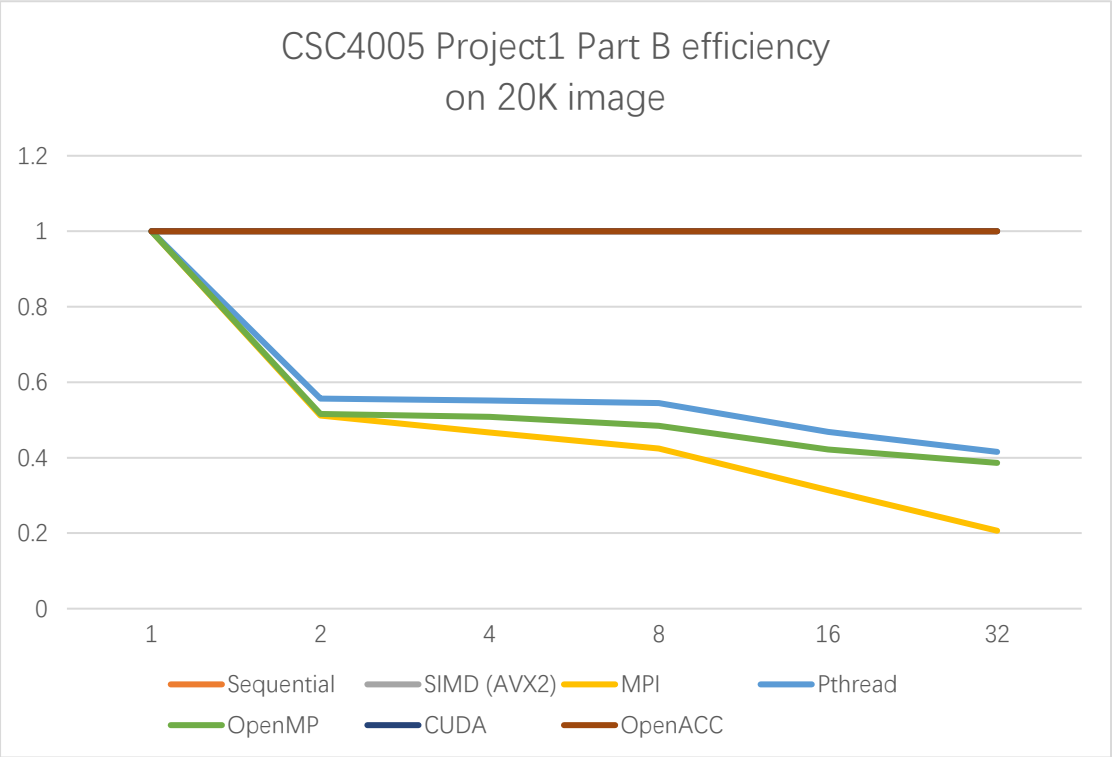


Figure 3. Changes of *Efficiencies for Part B*

Part A Speedup:

The speedup values for SIMD (AVX2), MPI, Pthread, OpenMP are all greater than 1, indicating improved performance compared to the sequential execution.

The speedup increases as the number of processes/cores increases, indicating better parallel scalability.

Part A Efficiency:

The efficiency values for SIMD (AVX2), MPI, Pthread, and OpenMP decrease as the number of processes/cores increases, indicating decreased efficiency with more parallelization.

Part B Speedup:

Similar to Part A, the speedup values for SIMD (AVX2), MPI, Pthread, OpenMP, are all greater than 1, indicating improved performance compared to the sequential execution.

The speedup increases with the number of processes/cores, demonstrating better parallel scalability.

Part B Efficiency:

The efficiency values for SIMD (AVX2), MPI, Pthread, OpenMP decrease as the number of processes/cores increases, indicating decreased efficiency with more parallelization.

Possible reasons for the observed differences:

Memory Overhead: The increased memory access and data movement required with more parallelization in Part A and Part B can introduce overhead, impacting efficiency.

Load Imbalance: Uneven distribution of computational tasks among processes/threads can lead to load