CSC4005 2023 Fall Project 1 Report

Tianci Hou 121090184

Please refer to the appendix for the generated images.

1 Part-A: RGB to Grayscale

1.1 Overall performance

Processes or Cores	Sequential	SIMD	MPI	Pthread	OpenMP	CUDA	OpenACC
1	638	419	699	715	587	27.1295	28
2	N/A	N/A	778	643	589	N/A	N/A
4	N/A	N/A	492	339	450	N/A	N/A
8	N/A	N/A	376	233	317	N/A	N/A
16	N/A	N/A	286	136	168	N/A	N/A
32	N/A	N/A	253	67	163	N/A	N/A

Table 1: Performance in Part A (millisecond)

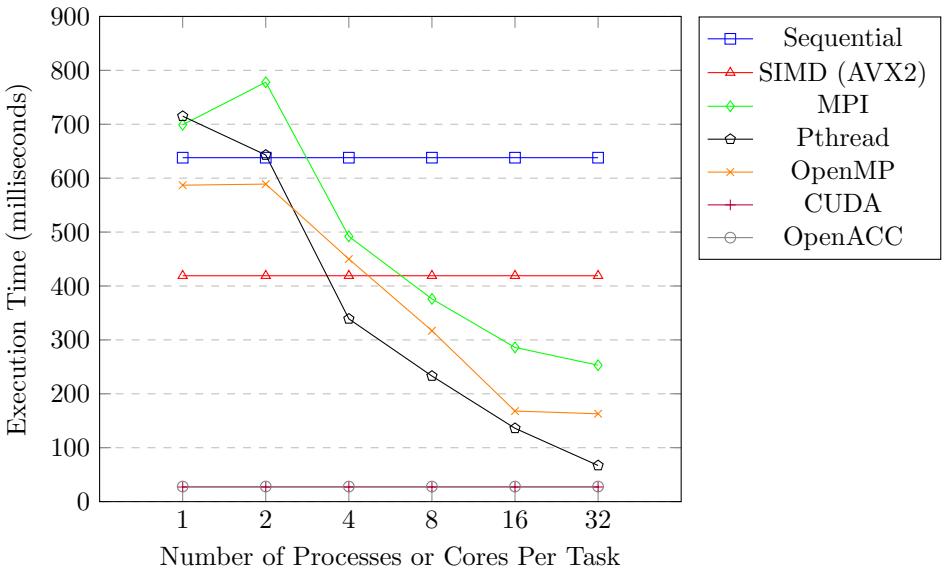


Figure 1: Execution time in Part A

1.2 Speed up and efficiency

Processes or Cores	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	52.27%	-8.73%	-10.77%	8.69%	2251.68%	2178.57
2	N/A	-17.99%	-0.78%	8.32%	N/A	N/A
3	N/A	29.67%	88.20%	41.78%	N/A	N/A
4	N/A	69.68%	173.82%	101.26%	N/A	N/A
5	N/A	123.08%	369.12%	279.76%	N/A	N/A
6	N/A	152.17%	852.24%	291.41%	N/A	N/A

Table 2: Speed up compared with sequential in Part A

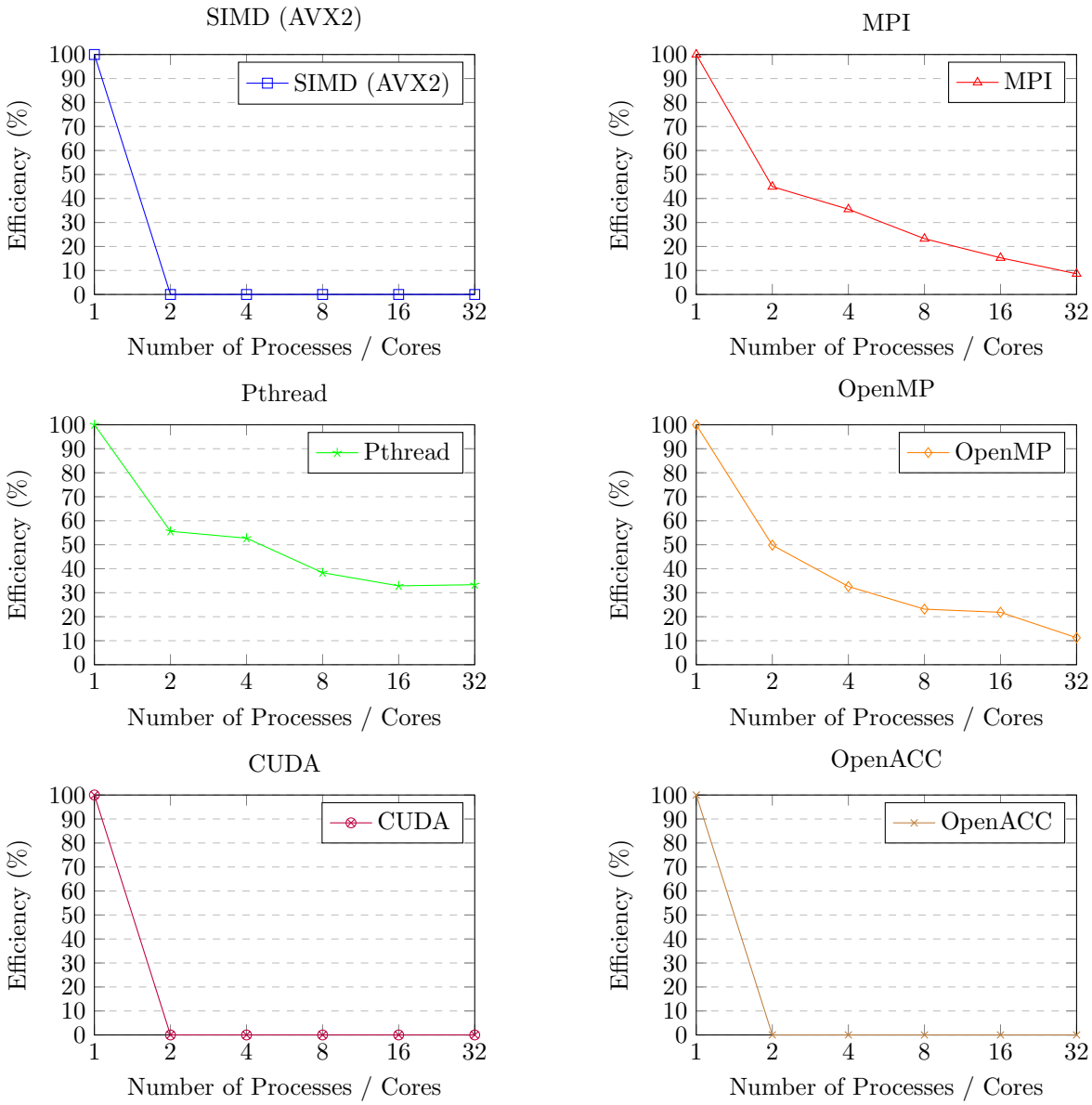


Figure 2: Efficiency in Part A

2 Part-B: Image Filtering (Soften with Equal Weight Filter)

2.1 Common techniques, data types and math expressions for Part B

Common techniques:

- Unrolled the nested for two loops (see it in the math expressions).

Data types:

- Use `float` as the basic data type during the transformation.
- Use `unsigned char` as the final values in the filtered image.

Math expressions:

$$Filtered\_image_{x,y} = \lfloor \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+1} Filter_{i-x-1,j-y-1} Orignial\_image_{i,j} \rfloor$$

2.2 Set up

Environment: Refer to course GitHub page.

Type	Compilation flags
Sequential	`-O2`
Simd	`-O2 -mavx2`
MPI	`-O2`
Pthread	`-O2`
OpenMP	`-O2 -fopenmp`
CUDA	default
OpenACC	`-acc`
OpenMP with SIMD	`-O2 -mavx2 -fopenmp`

(a) Compilation Flags

```
$ cd path/to/project
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ cd ..
# Running on cluster
$ sbatch src/script/sbatch_PartB.sh
# Running on PC
# Run at build/src/cpu or gpu
```

(b) Build Commands

Figure 3: Compilation and Build Information

2.3 Overall performance

Only includes basic performance required in this assignment.

The extra performance see 2.9.

Processes or Cores	Sequential	SIMD	MPI	Pthread	OpenMP	CUDA	OpenACC
1	4596	1104	4989	5043	5500	23	23
2	N/A	N/A	4744	4986	5468	N/A	N/A
4	N/A	N/A	2596	2546	2793	N/A	N/A
8	N/A	N/A	1518	1365	1415	N/A	N/A
16	N/A	N/A	989	720	837	N/A	N/A
32	N/A	N/A	640	435	430	N/A	N/A

Table 3: Performance in Part B

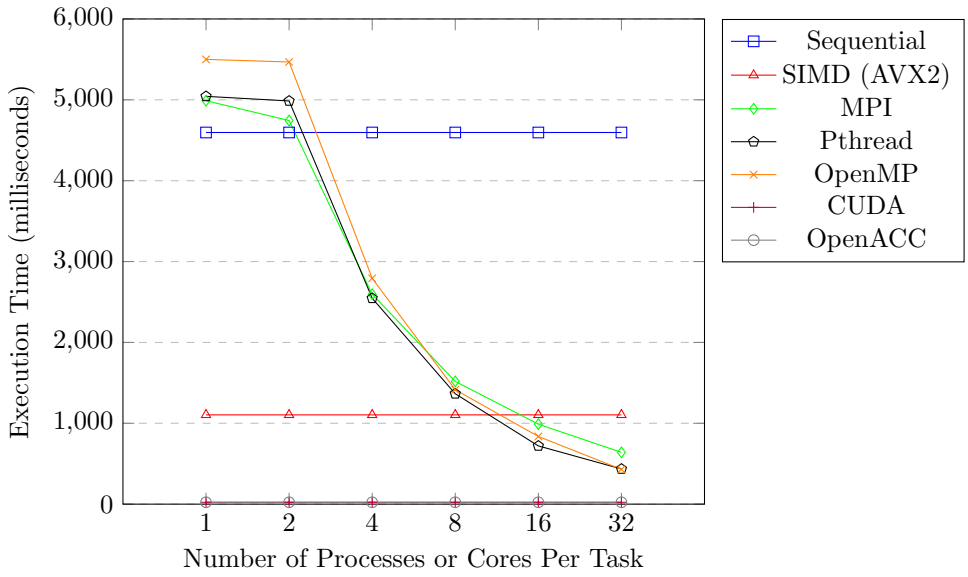


Figure 4: Execution time in Part B

2.4 Speed up and efficiency

Processes or Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	57.68%	292.66%	46.80%	59.94%	55.31%	39.13%	0.00%
2	N/A	N/A	50.38%	44.99%	33.49%	N/A	N/A
4	N/A	N/A	44.99%	50.67%	39.13%	N/A	N/A
8	N/A	N/A	37.88%	34.43%	31.59%	N/A	N/A
16	N/A	N/A	9.50%	28.33%	30.11%	N/A	N/A
32	N/A	N/A	8.44%	22.99%	40.70%	N/A	N/A

Table 4: Speed up compared with baseline in Part B

Processes or Cores	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	316.30%	-7.88%	-8.86%	-16.44%	19882.61%	19882.61%
2	N/A	-3.12%	-7.82%	-15.95%	N/A	N/A
4	N/A	77.04%	80.52%	64.55%	N/A	N/A
8	N/A	202.77%	236.70%	224.81%	N/A	N/A
16	N/A	364.71%	538.33%	449.10%	N/A	N/A
32	N/A	618.13%	956.55%	968.84%	N/A	N/A

Table 5: Speed up compared with sequential in Part B

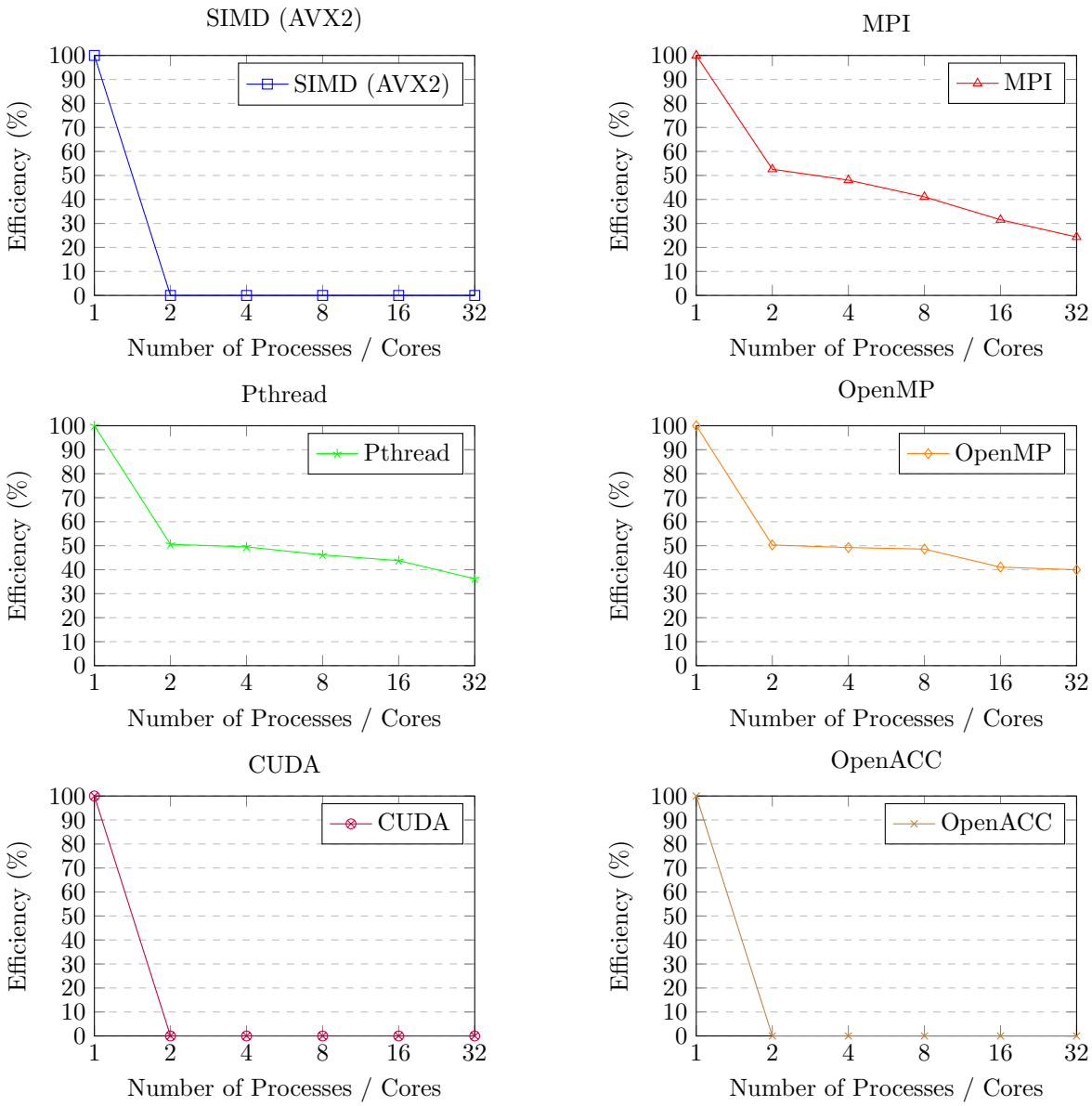


Figure 5: Efficiency in Part B

2.5 Sequential

2.5.1 Techniques

In the naive approach, it has a nested `for` loops, which is time consuming.

So I unrolled the nested loops, and this technique is used in other programs.

2.5.2 Sub experiment: std::round(x), x+0.5, x

In the naive approach, there is a typo that `sum_r` is declared as `int`, and when store the values into the final image, using `static_cast<unsigned char>(std::round(x))` to do the job. So I wonder the difference among `std::round`, `x+0.5` and `x`.

We can find that `std::round` is really time consuming since it has a `if` statement in itself. And since

Code	Time (millisecond)
std::round	8408
x+0.5	5260
x	4582

Table 6: Sub experiment std::round(x) x+0.5 x

we can guarantee that when doing the filter operation, `x+0.5` will not overflow, so we can use it to completely replace it.

2.6 SIMD (AVX2)

2.6.1 Techniques

When we do the filtering in the sequential way, we first find the target pixel and do the convolution within the 9 pixels. In the SIMD, we find 8 target pixels and do the convolution.

- Memory efficiency: we load the 8 pixels in the row. They occupy consecutive addresses in memory, and if there are less than 8, only the remaining ones are loaded.
- Instruction efficiency: when we want to get out the values. `_mm256_extract_epi32` is a good choice. It will translate to few instructions. Ref: Stack Overflow

2.6.2 Sub experiment: _mm256_fmadd_ps, _mm256_add_ps and _mm256_add_mul

When implementing the SIMD code, I found there is instruction called `_mm256_fmadd_ps` which can perform accumulation of vector multiplication. And we cannot directly use this one since gcc will throw the error "error: inlining failed in call to always_inline" . So I need to add "-march=native" to make it pass the compilation.

Code	Time (millisecond)
_mm256_fmadd_ps	1241
_mm256_add_ps and _mm256_add_mul	1104

Table 7: Sub experiment _mm256_fmadd_ps, _mm256_add_ps and _mm256_add_mul

The experiment shows that the is not significant differences between them, but `fmadd` is a bit slower. In addition, I found a post on the Stack Overflow. Although I haven't thoroughly understood the internal reasons yet, I think this is an interesting topic.

2.7 MPI, Pthread, OpenMP

2.7.1 Techniques

The overall idea is to cut the image into several regions and distribute them to child processes/threads.

- Memory efficiency 1: We cut by rows, as mentioned earlier, the values in the row occupy consecutive addresses in memory. This operation can also reduce false sharing.
- Memory efficiency 2: We can first separate the values of the three channels and create a new array, so that they are also continuous.
- Small subtask size: We try to make the area processed by each child process/thread as close as possible

2.7.2 Sub experiment: cut by rows, cut by pixels

Dividing by each pixel is definitely more uniform, so what is the time difference between them.

Code	Time (millisecond)
cut by rows	989
cut by pixels	1267

Table 8: Sub experiment cut by row, cut by pixels, MPI 16 cores

Maybe because I used `if` statement to determine whether it is an edge pixel, overall it is not as good as cutting by rows.

2.8 CUDA, OpenACC

2.8.1 Techniques

Because the number of cores used in the GPU is much larger than that of the CPU, I parallelize each pixel in the GPU.

The conclusion is that GPU is very suitable for image processing, even when processing 10K sized images, it can be completed in much less time than the CPU, and does not require complex optimization

2.9 Extra: OpemMP with SIMD (AVX2)

I tried to combine OpenMP and SIMD (AVX2) and do some experiments.

Processes or Cores	Sequential	SIMD	OpenMP	OpenMP with SIMD
1	4596	1104	5500	1492
2	N/A	N/A	5468	1193
4	N/A	N/A	2793	637
8	N/A	N/A	1415	333
16	N/A	N/A	837	199
32	N/A	N/A	430	118

Table 9: Extra Performance in Part B

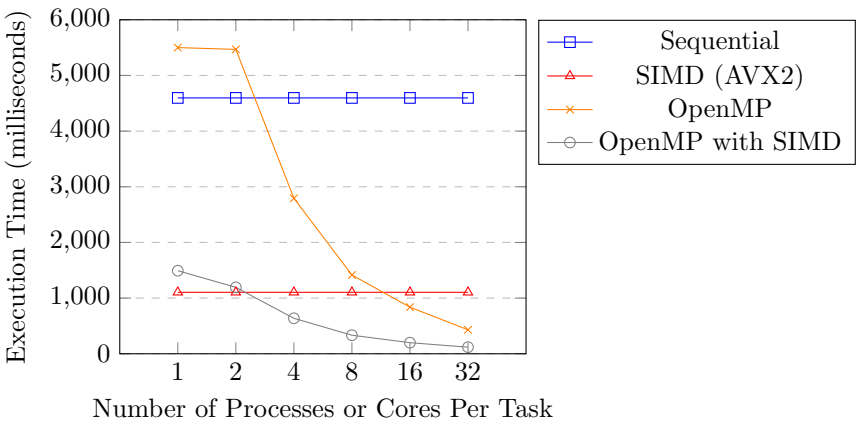


Figure 6: Extra Execution time in Part B

2.10 Conclusion

The similarities between these models are that they all aim to achieve parallelism in order to improve performance. They all have different ways of achieving parallelism, such as SIMD for data-parallel applications, MPI for distributed memory systems, and Pthreads and OpenMP for shared memory systems. They also have different levels of parallelism, such as instruction-level parallelism (ILP), data-level parallelism (DLP), and task-level parallelism (TLP).

The differences between these models are mainly in their target architectures and programming paradigms. For example, CUDA is designed specifically for NVIDIA GPUs, while OpenACC is designed to work with a variety of accelerators (also can be used in GPU). Pthreads and OpenMP use threads for parallelism, while MPI uses message passing.

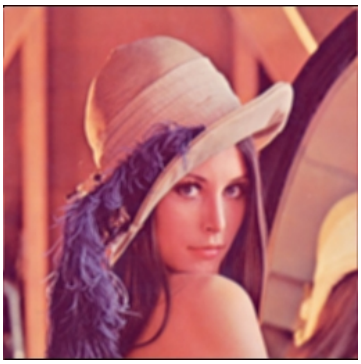
From the results of experiments, we can find the efficiency drops when the number of cores or processes increases. And manage data accessing in a efficient way is the most important key for both Part A and Part B.

Appendix

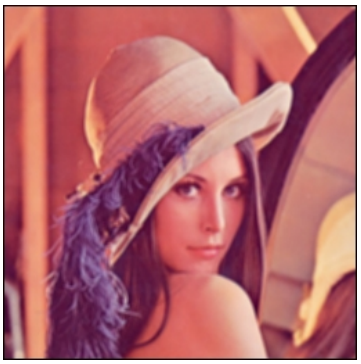
(a) Lena-RGB

(b) Part A: Gray

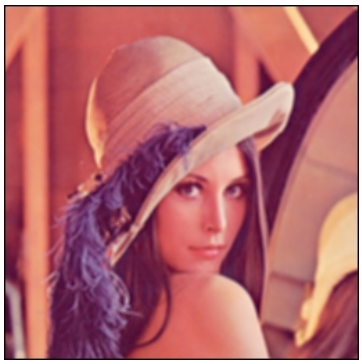
(c) Part B: Sequential



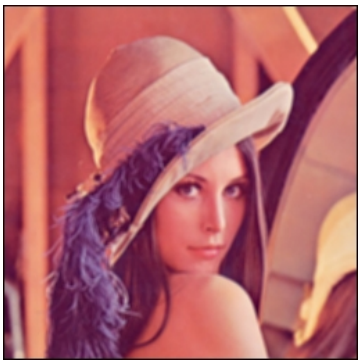
(d) Part B: SIMD



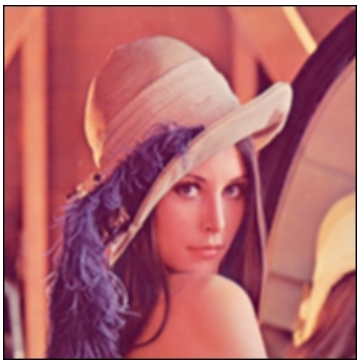
(e) Part B: MPI



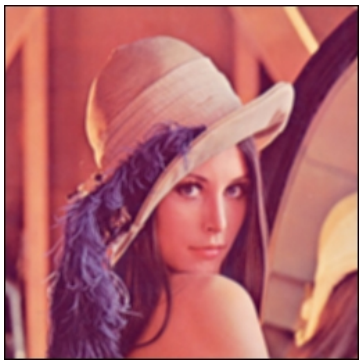
(f) Part B: Pthread



(g) Part B: OpenMP



(h) Part B: CUDA



(i) Part B: OpenACC