Project 1

Introduction

In this project, we are asked to implement the embarrassing parallel forms of the bilateral filter, which is a filter frequently utilized in Computer Vision. The weights are influenced by both the Euclidean distance between the current pixel and its neighbors, as well as the radiometric differences.

Compilation and Execution

To compile the program, please do the following steps:

```
cd project1
mkdir build && cd build
cmake ..
make -j4
```

After compilation, in order to batch process the project in order to get the execution time, you can simply `sbatch` at the project root directory:

```
cd ..
sbatch ./src/scripts/sbatch_PartC.sh
```

The result is stored in `Project1-PartC-Results.txt`. You can use `vim` to open it:

```
vim Project1-PartC-Results.txt
```

To get the expected output image for each program individually, you can run the compiled file under `build` directory separately:

```
cd build
# Sequential
./src/cpu/sequential_PartC_soa ../images/4K-RGB.jpg ../output.jpg
./src/cpu/sequential_PartC_aos ../images/4K-RGB.jpg ../output.jpg
# SIMD
./src/cpu/simd_PartC ../images/4K-RGB.jpg ../output.jpg
# MPI
mpirun -np {Num of Processes} ./src/cpu/mpi_PartC ../images/4K-RGB.jpg
../output.jpg
# Pthread
./src/cpu/pthread_PartC ../images/4K-RGB.jpg ../output.jpg {Num of Threads}
# OpenMP
./src/cpu/openmp_PartC ../images/4K-RGB.jpg ../output.jpg
# CUDA
./src/gpu/cuda_PartC ../images/4K-RGB.jpg ../output.jpg
# OpenACC
./src/gpu/openacc_PartC ../images/4K-RGB.jpg ../output.jpg
```

Parallel Principle and Similarities/Differences

For all the methods except SIMD, they all exploit SPMD paradigm and DLP, since they execute the same program but with different data elements. The difference are:

1. CUDA and OpenACC utilize GPU instead of CPU.
2. MPI is multi-processing while the rest CPU parallelisms are multi-threading.

For SIMD, it is a Parallel Processing Computer Type which nowadays most of our CPU cores support. It focuses on the parallelism on a single operation taking effect in registers of the CPU, since each register is able to contain more than one integer, floating point, etc.

Optimizations

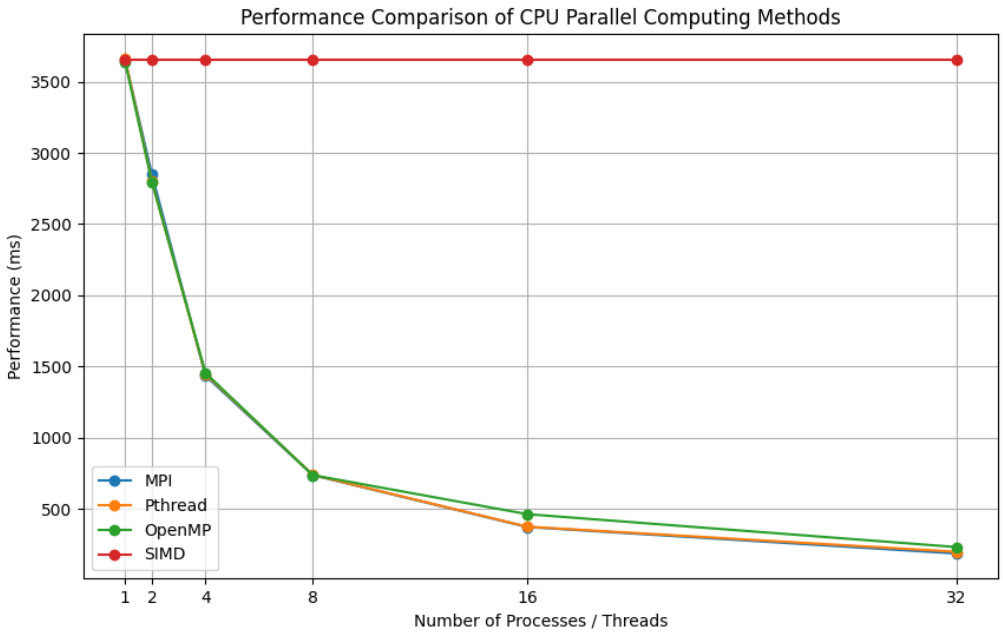
1. For SIMD, I do parallelism on multiplication of spatial and intensity weights, and sum reduction utilizing SSE horizontal adding operations. I also use SOA for better spatial locality.
2. For pthread and OpenMP, I share the data to different threads and use SOA for spatial locality.
3. For MPI, since it needs message passing, I use AOS for the output image structure. It sends and receives as a whole rather than separately as what SOA does, which can speed up the process. For image processing for input image, SOA structure is still used since it better utilizes the spatial locality and is faster than AOS.
4. For CUDA, I do grid search on block size, and find 32 is the optimized size, and the size larger than 32 is not appropriate for the server GPU. I also implement AOS than SOA and it has a 0.5ms benefits, since the memory copy operation cost less time.
5. For OpenACC, I use AOS for both input and output since the memory copy operation cost less time. It copies once but SOA copies three times for each channel. And the result is also much faster than the baseline. **(For bonus)**
6. For Triton, I use torch tensor to store and process the image. **(For bonus)**

Result

The result is shown below in ms:

Number of Processes / Cores	SIMD (AVX2, -O2)	MPI (-O2)	Pthread (-O2)	OpenMP (-O2)	CUDA	OpenACC	Triton
1	3652	3653	3662	3640	9.6	5	1604.46
2		2849	2802	2797			
4		1436	1447	1455			
8		738	740	738			
16		374	376	464			
32		188	200	233			

The following figure shows the performance comparison for CPU speedup:



The speedup factors are shown as followed, using $S(p) = \frac{t_s}{t_p}$ and Sequential -O2 (3745ms) as a baseline. All MPI, Pthread and OpenMP has similar speedup performance, while OpenACC is faster than CUDA and Triton using GPU.

Number of Processes / Cores	SIMD (AVX2, -O2)	MPI (-O2)	Pthread (-O2)	OpenMP (-O2)	CUDA	OpenACC	Triton
1	1.025	1.025	1.023	1.029	390	749	2.334
2		1.314	1.337	1.339			
4		2.608	2.588	2.574			
8		5.075	5.061	5.075			
16		10.013	9.960	8.071			
32		19.920	18.725	16.073			

The efficiencies for MPI, pthread and OpenMP are shown below. It has a decreasing trend when the core number becomes larger.

Number of Processes / Cores	MPI (-O2)	Pthread (-O2)	OpenMP (-O2)
1	102.5%	102.3%	102.9%
2	65.7%	66.9%	70.0%
4	65.2%	64.7%	64.4%
8	63.4%	63.3%	63.4%
16	62.6%	62.3%	50.4%
32	62.3%	58.5%	50.2%

Result Analysis and Comparison to Part A&B

1. The parallel implementations significantly outperform the sequential version. For instance, while the sequential implementation takes around 3745 ms, CUDA and OpenACC reduce the time to only 9.6 and 5ms. The multi-threading methods and MPI also demonstrate considerable speedup, especially as the number of processes or threads increases.
2. For scalability, MPI, Pthread and OpenMP all exhibit diminishing returns as more cores or processes are used, indicating potential overhead from thread management or communication costs. This is particularly evident in the OpenMP results, where efficiency drops around 50% at 16 and 32 cores.
3. Comparing to Part A&B, we can find that the SIMD performance improves not that much in Part C. This is because both A&B have more operations for registers to do parallelism, while the largest overhead in Part C (exponential function) is hard to be parallelized, therefore resulting in a performance shortage. The rest performance seems to be similar to the other parts.