The Chinese University of Hong Kong, Shenzhen

CSC4005 2024 Fall

Parallel Programming

Project 1

# Embarrassingly Parallel Programming

*Author :* Yuzhe Yang
*Student ID:* 121090684

December 5, 2024

# 1 Introduction

This project aims to provide hands-on experience with parallel programming by solving an embarrassingly parallel problem in image processing. It is divided into three parts, each focusing on different aspects of parallel computation across various parallel programming models. The task for each part involves processing images and measuring the performance of different parallel implementations.

### Part A: RGB to Grayscale Conversion

In this part, the task is to convert an RGB image to grayscale using the NTSC formula. The provided source code for different parallel programming models such as SIMD, MPI, Pthread, OpenMP, CUDA, and OpenACC is used to execute this task. This part is designed to introduce the simplest form of parallel computation.

### Part B: Linear Image Filtering (Smoothing)

Here, the task involves applying a $3 \times 3$ equal-weight smoothing filter to the image. This filter considers neighboring pixel values, making the computation slightly more complex but still embarrassingly parallel. The same parallel programming models are used to evaluate performance in this part.

### Part C: Non-Linear Image Filtering (Bilateral Filtering)

This part requires the implementation of a bilateral filter, a non-linear filter that preserves edges while smoothing regions with similar intensity values. Unlike the previous parts, students must implement this parallel program themselves, testing their knowledge of parallelization strategies and optimization techniques.
The bilateral filter[1] computes each output pixel intensity $I'(p)$ as a weighted average of nearby pixels' intensities:

$$I'(p) = \frac{1}{W_p} \sum_{q \in \Omega} I(q) k_r(||I(q) - I(p)||) k_s(||q - p||) \tag{1}$$

where $W_p$ is the normalization term:

$$W_p = \sum_{q \in \Omega} k_r(||I(q) - I(p)||) k_s(||q - p||) \tag{2}$$

Here, $I$ represents the image, $p$ and $q$ are pixel positions, $\Omega$ is the filter kernel, $k_r$ is the range kernel for radiometric differences, and $k_s$ is the spatial kernel for geometric distances. The range kernel is defined as:

$$k_r(||I(q) - I(p)||) = \exp\left(-\frac{||I(q) - I(p)||^2}{2\sigma_r^2}\right) \tag{3}$$

and the spatial kernel as:

$$k_s(||q - p||) = \exp\left(-\frac{||q - p||^2}{2\sigma_s^2}\right) \tag{4}$$

[1] I noticed some typographical errors in the formula provided on the NVIDIA official website `https://docs.nvidia.com/vpi/algo_bilat_filter.html`. For accurate formula details, I referred to `https://en.wikipedia.org/wiki/Bilateral_filter`.

## 2 Experiments

### 2.1 Main Results

The final performance results in Part C, including both my own implementation and a comparison with the baseline provided by the teaching staff, are shown in Table 1 and Table 2, respectively. These tables highlight the execution times for different parallel programming models across various configurations, providing a comprehensive understanding of how different models perform in practice. **All tests were conducted on the Slurm cluster.**

| Processes / Cores | SIMD | MPI | Pthread | OpenMP | CUDA | OpenACC |
|---|---|---|---|---|---|---|
| 1 | **835** (-77.3%) | **1482** (-59.2%) | **2215** (-38.3%) | **2363** (-35.3%) | **5.07** (-47.2%) | **6** (-86.0%) |
| 2 | - | **1385** (-51.2%) | **1743** (-37.8%) | **1204** (-57.0%) | - | - |
| 4 | - | **699** (-51.1%) | **920** (-41.0%) | **1136** (-21.8%) | - | - |
| 8 | - | **354** (-50.8%) | **515** (-30.0%) | **555** (-26.6%) | - | - |
| 16 | - | **186** (-48.5%) | **260** (-34.3%) | **264** (-47.3%) | - | - |
| 32 | - | **105** (-42.3%) | **159** (-26.7%) | **187** (-24.3%) | - | - |

Table 1: My performance and differences related to the baseline in Part C (4K JEPG).

| Processes / Cores | SIMD | MPI | Pthread | OpenMP | CUDA | OpenACC |
|---|---|---|---|---|---|---|
| 1 | 3674 | 3640 | 3661 | 3654 | 9.6 | 43 |
| 2 | - | 2838 | 2804 | 2799 | - | - |
| 4 | - | 1428 | 1560 | 1452 | - | - |
| 8 | - | 719 | 736 | 756 | - | - |
| 16 | - | 361 | 396 | 501 | - | - |
| 32 | - | 182 | 217 | 247 | - | - |

Table 2: Baseline performance in Part C (4K JEPG).

The program is recompiled before each run. To expedite the testing process on the CPU, I directly executed the experiments on the login node:

```bash
CURRENT_DIR=$(pwd)/src/scripts
echo "Current directory: ${CURRENT_DIR}"
echo "OpenMP PartC (Optimized with -O2)"
for num_cores in 1 2 4 8 16 32
do
  echo "Number of cores: $num_cores"
  export OMP_NUM_THREADS=$num_cores
  ${CURRENT_DIR}/../../build/src/cpu/openmp_PartC ${CURRENT_DIR}/../../
    images/4K-RGB.jpg ${CURRENT_DIR}/../../images/4K-Bilateral.jpg $num_cores
  echo ""
done
```

This script tests the OpenMP version of the program optimized with `-O2` by varying the number of cores from 1 to 32. The performance is evaluated on a 4K image file. For the GPU-based programs, I submitted jobs through the Slurm scheduler:

```bash
rm -rf ./profile/*
> ./tmp/gpu.txt
sbatch ./src/scripts/gpu.sh # part of sbatch_PartC.sh
tail -f ./tmp/gpu.txt
```

This script clears the profiling data and submits the GPU jobs using `sbatch`, tracking the output via the `gpu.txt` log file.

The results indicate a significant performance improvement in my implementation compared to the baseline, particularly in the GPU baselines. I will elaborate on my approach in the following section 3.

## 2.2 Other Results

As shown in Tables 3 and 4, we observe the execution times of various implementations for Parts A and B. The results highlight performance differences among sequential, SIMD, MPI, Pthread, OpenMP, CUDA, OpenACC, and Triton implementations.
For Part A (Table 3), which involves converting an RGB image to grayscale using the NTSC formula, the sequential implementation takes 651 ms, while SIMD optimization reduces it to 410 ms. Triton achieves the best performance with 2.68 ms, highlighting the effectiveness of GPU acceleration. CUDA also performs well at 35.322 ms. MPI, Pthread, and OpenMP show speedup with increasing numbers of processes/cores, but diminishing returns appear beyond 16 or 32 units due to overhead.
In Part B (Table 4), the task is to apply a $3 \times 3$ equal-weight smoothing filter to the image, which involves more computation compared to Part A. The SOA version outperforms AOS, with 6614 ms versus 7262 ms. SIMD further reduces the time to 4271 ms. Triton again achieves the best performance with 2.68 ms, while CUDA and OpenACC show good results with 31.1636 ms and 23 ms, respectively. CPU-based approaches like MPI, Pthread, and OpenMP show diminishing returns as more cores are added, mainly due to synchronization overhead.

| **Implementation** | **Configuration** | **Execution Time (ms)** |
|---|---|---|
| Sequential (Optimized with -O2) | - | 651 |
| SIMD (AVX2) (Optimized with -O2) | - | 410 |
| MPI (Optimized with -O2) | 1 Process | 630 |
| | 2 Processes | 774 |
| | 4 Processes | 501 |
| | 8 Processes | 361 |
| | 16 Processes | 288 |
| | 32 Processes | 281 |
| Pthread (Optimized with -O2) | 1 Core | 704 |
| | 2 Cores | 639 |
| | 4 Cores | 342 |
| | 8 Cores | 230 |
| | 16 Cores | 101 |
| | 32 Cores | 79 |
| OpenMP (Optimized with -O2) | 1 Core | 588 |
| | 2 Cores | 588 |
| | 4 Cores | 445 |
| | 8 Cores | 275 |
| | 16 Cores | 178 |
| | 32 Cores | 129 |
| CUDA | - | 35.322 |
| OpenACC | - | 36 |
| Triton | - | 2.68 |

Table 3: Execution Time for Part A.

Multi-core or multi-process implementations generally reduce execution time, but diminishing returns are evident beyond 16 or 32 cores/processes. For example, in MPI (Part A), the execution time drops from 630 ms (1 process) to 288 ms (16 processes), but the improvement becomes less significant beyond that. Similar trends are seen with Pthread and OpenMP, reflecting the impact of thread management and synchronization costs.
Overall, GPU-based methods like Triton and CUDA offer the best performance for these workloads, especially for computationally intensive tasks like image processing. CPU-based methods

| **Implementation** | **Configuration** | **Execution Time (ms)** |
|---|---|---|
| Sequential (AOS) (Optimized with -O2) | - | 7262 |
| Sequential (SOA) (Optimized with -O2) | - | 6614 |
| SIMD (AVX2) (Optimized with -O2) | - | 4271 |
| MPI (Optimized with -O2) | 1 Process | 7281 |
| | 2 Processes | 7102 |
| | 4 Processes | 3740 |
| | 8 Processes | 2086 |
| | 16 Processes | 1059 |
| | 32 Processes | 654 |
| Pthread (Optimized with -O2) | 1 Core | 8082 |
| | 2 Cores | 7231 |
| | 4 Cores | 3838 |
| | 8 Cores | 1846 |
| | 16 Cores | 939 |
| | 32 Cores | 477 |
| OpenMP (Optimized with -O2) | 1 Core | 8547 |
| | 2 Cores | 7313 |
| | 4 Cores | 3870 |
| | 8 Cores | 1851 |
| | 16 Cores | 996 |
| | 32 Cores | 505 |
| CUDA | - | 31.1636 |
| OpenACC | - | 23 |
| Triton | - | 2.68 |

Table 4: Execution Time for Part B.

face limitations with increased cores/processes due to synchronization and management overhead.

## 3 Overview Design

In this section, I will provide a detailed overview of how my Part C program was designed, along with an explanation of the strategies I implemented to enhance the performance of parallel computing.

### 3.1 GPU Part

Both the CUDA and OpenACC implementations of the bilateral filter share several optimization strategies aimed at enhancing the efficiency of parallel processing on the GPU. These common strategies include the precomputation of spatial Gaussian weights, efficient memory access management, and parallelization of the computation across pixels to maximize the use of GPU resources. Both implementations also focus on reducing redundant computations and memory transfers, which can be significant bottlenecks in parallel processing.

#### CUDA

The program uses CUDA to implement a bilateral filter for edge-preserving noise reduction. The bilateral filter is applied in parallel across image pixels using a CUDA kernel. Each pixel and its neighbors are processed concurrently by different threads, allowing efficient workload distribution. The overall strategy is to utilize GPU parallelism to handle many independent pixel operations at the same time, thus significantly speeding up the filtering process. The key optimizations involve minimizing memory access latency by making effective use of shared memory, balancing the workload across threads, and ensuring efficient computation of filter weights. Below, I provide more detailed explanations of the main optimization techniques used.

- **Shared Memory Optimization:** Shared memory is used to store local neighborhoods, which significantly reduces redundant global memory accesses and improves speed. Since global memory access is relatively slow, using shared memory allows threads within the same block to quickly access the required pixel data for computations.

```
__shared__ unsigned char shared_mem[BLOCK_SIZE + 2 * RADIUS][BLOCK_SIZE
    + 2 * RADIUS][3];
```

- **Precomputed Spatial Gaussian Weights:** Spatial weights are precomputed and stored in a lookup table. This optimization reduces the amount of computation that each thread must perform during filtering, as spatial weights only depend on the distance between pixels and do not change across the image.

```
float h_spatial_gaussian[(2 * RADIUS + 1) * (2 * RADIUS + 1)];
for (int dy = -RADIUS; dy <= RADIUS; dy++) {
    for (int dx = -RADIUS; dx <= RADIUS; dx++) {
        h_spatial_gaussian[(dy + RADIUS) * (2 * RADIUS + 1) + (dx +
    RADIUS)] = gaussian(sqrtf(dx * dx + dy * dy), SIGMA_S);
    }
}
```

- **Thread Block and Grid Sizes:** The block size is set to 16x16 to balance the number of threads per block and shared memory usage. This ensures that the GPU's computational resources are fully utilized, while also allowing each thread block to load an appropriate amount of data into shared memory.

```
dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height +
    blockSize.y - 1) / blockSize.y);
```

- **Thread Synchronization:** Synchronization is used within each thread block to ensure that all threads have finished loading their respective pixel data into shared memory before proceeding with the computation. This prevents race conditions and ensures correctness in the computation of filtered pixel values.

```
__syncthreads();
```

## OpenACC

The program also implements a bilateral filter using OpenACC to parallelize the process across the CPU and GPU. By employing OpenACC directives, the computation is offloaded to the GPU, which accelerates the filtering process significantly. Below are the key strategies and optimizations used to maximize performance.

- **Data Management:** The program uses the `acc enter data` and `acc exit data` directives to manage data transfer between the CPU and GPU efficiently. By explicitly controlling when data is copied to the GPU and when it is brought back, unnecessary memory transfers are avoided, which improves performance.

```
#pragma acc enter data copyin(filteredImage[0:buffer_size], buffer[0:
    buffer_size], spatial_gaussian[0:FILTER_DIAMETER*FILTER_DIAMETER])
#pragma acc exit data delete(buffer, filteredImage, spatial_gaussian)
```

- **Parallel Loop Optimizations:** The most computationally expensive part of the algorithm is the nested loops that apply the bilateral filter to each pixel. By using the `acc parallel` and `acc loop collapse(2) independent` directives, the loops are parallelized, allowing the GPU to process multiple pixels in parallel. The collapse(2) directive is used to merge the two nested loops into a single parallel loop, which provides better workload distribution across GPU threads.

```
#pragma acc parallel present(filteredImage[0:buffer_size], buffer[0:
    buffer_size], spatial_gaussian[0:FILTER_DIAMETER*FILTER_DIAMETER])
    num_gangs(1024)
{
    #pragma acc loop collapse(2) independent
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            ...
        }
    }
}
```

- **Precomputed Spatial Gaussian Weights:** Similar to the CUDA version, spatial weights are precomputed and stored in memory before the filtering begins. This reduces redundant computation during the filtering process, as spatial weights do not depend on the image content but only on the pixel positions.

```
float spatial_gaussian[FILTER_DIAMETER * FILTER_DIAMETER];
int index = 0;
for (int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++) {
    for (int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++) {
        spatial_gaussian[index++] = gaussian(sqrtf(dx * dx + dy * dy),
    SIGMA_D);
    }
}
```

- **Efficient Memory Access:** To reduce memory access latency, the program minimizes the number of memory transfers between the CPU and GPU. The image data is copied once to the GPU, processed entirely on the device, and then copied back to the CPU only after the filtering is complete. This reduces the overhead associated with frequent memory transfers, which is a common bottleneck in GPU-accelerated programs.

```
#pragma acc update self(filteredImage[0:buffer_size])
```

- **Gang and Worker Configuration:** The program specifies the number of gangs and workers with the `num_gangs` clause to balance workload distribution and ensure optimal utilization of GPU resources. By experimenting with different configurations, the number of gangs is set to 1024 to achieve a balance between performance and resource usage.

## 3.2 CPU Part

### SIMD

The program implements a bilateral filter using SIMD (AVX2) instructions to accelerate the filtering process on the CPU. By utilizing SIMD, the program performs multiple operations in parallel within a single instruction, improving performance by processing multiple pixels simultaneously. Below are the key strategies and optimizations employed in the SIMD implementation.

- **Spatial Weight Precomputation:** As in the GPU implementations, the spatial weights are precomputed to avoid redundant calculations during the filtering process. The weights are stored in a 2D array and computed using the formula:

```
spatial_weights[(dy + RADIUS) * (2 * RADIUS + 1) + (dx + RADIUS)] = exp
    (-(distance * distance) / (2 * SIGMA_D * SIGMA_D));
```

  where `distance` is the Euclidean distance between pixels.

- **SIMD-based Exponential Function:** To accelerate the computation of the exponential function for the range and spatial weights, a fast approximation of `exp()` is implemented using AVX2 instructions. The function, `__m256 exp_ps(__m256 x)`, approximates the exponential function by clamping input values to prevent overflow, reducing and approximating the exponential using polynomial coefficients. The final result is computed by multiplying the exponentiated values by powers of 2:

```
__m256 exp_ps(__m256 x) {
    // Clamp input values to prevent overflow or underflow
    x = _mm256_min_ps(x, _mm256_set1_ps(88.0f));  // exp(88) is close to
    FLT_MAX
    x = _mm256_max_ps(x, _mm256_set1_ps(-88.0f)); // exp(-88) is very
    close to 0

    // Reduce x by log2(e)
    __m256 t = _mm256_mul_ps(x, _mm256_set1_ps(1.44269504088896341f));
    // x * log2(e)

    // Exponentiation by powers of 2
    __m256 exp2_ipart = _mm256_castsi256_ps(_mm256_slli_epi32(
    _mm256_add_epi32(_mm256_cvtps_epi32(t), _mm256_set1_epi32(127)), 23)
    );

    // Polynomial approximation
    __m256 y = _mm256_add_ps(_mm256_mul_ps(_mm256_set1_ps(0.0001985f), t
    ), _mm256_set1_ps(1.0f));
    return _mm256_mul_ps(exp2_ipart, y);
}
```

- **SIMD Registers for Pixel Processing:** During filtering, neighboring pixel values are loaded into AVX2 registers using `__m256` vectors. For example, the R, G, and B values of neighboring pixels are loaded into SIMD registers:

```
__m256 r_neighbors = _mm256_set_ps(r_values[0], r_values[1], r_values
    [2], r_values[3], r_values[5], r_values[6], r_values[7], r_values
    [8]);
```

- **SIMD Vectorized Distance Calculations:** The squared differences between the center pixel and its neighbors for each channel (R, G, B) are calculated in parallel using SIMD. For example, the range weights for the R channel are computed:

```
__m256 r_diff = _mm256_sub_ps(r_neighbors, _mm256_set1_ps(center_r));
__m256 r_kernel = exp_ps(_mm256_mul_ps(_mm256_set1_ps(-1.0f),
    _mm256_mul_ps(r_diff, r_diff)));
```

- **Summing and Normalizing Results:** After calculating the bilateral filter contributions for each neighboring pixel, SIMD horizontal addition is used to sum the values across the SIMD registers. For example, horizontal addition is performed using:

```
__m256 temp_r = _mm256_hadd_ps(sum_r, sum_r);
__m128 low_r = _mm256_extractf128_ps(temp_r, 0);
__m128 high_r = _mm256_extractf128_ps(temp_r, 1);
low_r = _mm_add_ps(low_r, high_r);
float r_sum_val = _mm_cvtss_f32(low_r);
```

- **Efficient Memory Access:** The program processes each pixel's neighborhood in blocks, loading them into SIMD registers to reduce the number of memory accesses. By processing 8 pixels at once, the program minimizes the time spent accessing memory and improves cache efficiency.

## Other

For the Pthread, MPI, and OpenMP implementations on the CPU, I employed a similar optimization approach across all three models. Each model takes advantage of thread-level parallelism (TLP) to accelerate the bilateral filtering process by dividing the task into smaller sub-tasks. In each case, I precomputed the spatial weights before performing the filtering, reducing redundant calculations and improving efficiency.
While the overall strategy was consistent, each model has distinct features. Pthread and OpenMP are designed for shared memory systems, allowing threads to work directly on shared data. In both implementations, I divided the image into smaller sections, with each thread processing its assigned portion in parallel. In the OpenMP implementation, I used parallel loops to efficiently distribute the workload across multiple cores.
MPI, on the other hand, is designed for distributed memory systems. In this case, I split the image across different nodes, with each node performing the filtering independently on its assigned portion. The results were then gathered and combined at the master node. Although synchronization and communication between tasks were necessary in all models, they were handled differently depending on the underlying architecture of each.
In summary, while the core optimization—precomputing spatial weights and parallelizing the filtering—was shared, the implementations differed based on the memory architecture and parallelization model. This allowed me to tailor the performance improvements to each specific parallel framework.