COMS4040A & COMS7045A Assignment 1 – Report

Anand Patel, 2561034, Coms Hons

7 May 2025

Problem 1 - Parallelizing Brute-Force KNN using OpenMP

1.1 - Brute-Force KNN

The *k-nearest neighbours (KNN)* algorithm is a simple method used in machine learning for classification and regression tasks. In this assignment we implement a brute force version of the KNN classifier. We implement a version where the algorithm classifies a test sample by computing its distance to every point in the training dataset. This method differs from other, more advanced KNN classifier algorithms such as *kd-trees*, *Ball tree* and *Approximate Nearest Neighbour*, whereas the brute force method relies on a purely exhaustive search.

1.2 Overview of Serial Implementation

The serial and parallel implementations follow the same fundamental steps, but differ in how computation is distributed—particularly in distance calculation and sorting. In general both versions (serial and parallel) can be broken down into 4 steps. For each test point, the brute-force KNN algorithm performs the following steps:

1. Compute the Euclidean Distance between the test point and all training points.
2. Sort the computed distances using the quicksort algorithm.
3. Select the *k smallest* distances (*k nearest* neighbours).
4. Assign the test point to the class most frequently represented among the previously selected *k neighbours*.

The Euclidean Distance calculation is the straightest and shortest path between 2 points. Mathematically the Euclidean Distance is

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

This mathematical formula is implemented in C++ (serial version) as follows:

```
for (size_t j = 0; j < NUM_SAMPLES_TRAIN; ++j) {
    double sum = 0.0;
    for (size_t k = 0; k < FEATURE_DIM; ++k) {
        double diff = features_test[i][k] - features_train[j][k];
        sum += diff * diff;
    }
    double eu_dist = sqrt(sum);
    curr_dist.push_back({eu_dist, labels_train[j]});
}
```

Listing 1: Serial Euclidean Distance Calculation in C++

This version calculates the distance sequentially for each training point and stores the result in a vector of **distance_single_sample** structs, which will later be sorted to determine the nearest neighbours.

```
struct distance_single_sample {
    double distance;
    int label;
};
```

Listing 2: distance_single_sample struct

Once all distances are computed and stored in a vector, they need to be sorted in ascending order to identify the k nearest neighbours. This is done using a custom implementation of the quicksort algorithm.

Quicksort is a divide-and-conquer sorting algorithm that works by selecting a *pivot* element and partitioning the array such that elements less than the pivot appear before it and those greater appear after. The process is then recursively applied to the left and right partitions.

The quicksort algorithm can be summarized as follows:

1. Choose a pivot element from the list (usually the last element).
2. Partition the list into two sublists:
    - Elements less than the pivot.
    - Elements greater than or equal to the pivot.
3. Recursively apply steps 1 and 2 to the sublists.

In this assignment, we implemented our own quicksort algorithm instead of using the built-in `std::sort` function. This was due to 1) the brief requires a custom implementation for the parallelization of recursive algorithms and 2) while `std::sort` is optimized for general use, it removes control over recursion and parallelism, making it unusable for fine-tuned performance experiments in a parallel computing context. A manual quicksort implementation allows better control over thread distribution, recursion depth, and memory access. After sorting the distances, the k nearest neighbours are selected based on the smallest k distances. Each of these neighbours has an associated class label. To determine the predicted class for the test point, a majority voting scheme is applied over these k labels.

The logic for majority voting can be described as follows:

1. Initialize a count vector (e.g., of size 10 for CIFAR10) to keep track of votes per class.
2. For the k nearest neighbours:
    - Increment the vote count for the corresponding class label.
3. Select the class label with the highest vote count as the predicted label.

In the case of a tie, the implementation defaults to the class with the lowest numerical label.

This method is efficient, as it only requires a single pass through the top k sorted elements and uses fixed-size arrays for class vote accumulation.

Table 1: Example of majority voting among 5 nearest neighbours

Distance	Class Label
0.42	3
0.55	3
0.60	1
0.70	7
0.74	3

In this example, the class labels of the 5 nearest neighbours are $\{3, 3, 1, 7, 3\}$. Class 3 receives the highest number of votes (3 out of 5), and is therefore selected as the predicted label for the test sample.

1.3 Overview of Parallel Implementation

The parallel implementation of the brute-force KNN algorithm follows the same steps as the serial version, but parallelizes the two most computationally expensive parts: the Euclidean distance computation and the sorting of distances. Each test sample is processed independently, making the algorithm easily parallelisable and well-suited for thread-based parallelism.

1.3.1 Parallelizing Distance Computation with `#pragma omp for`

The outer loop over test samples was parallelized using the `#pragma omp for` directive. This schedules the test samples across threads in a static round-robin fashion. Additionally, the inner loop over the feature dimensions was vectorized using `#pragma omp simd` to take advantage of SIMD-level parallelism during the Euclidean distance calculation.

This approach ensures:

- No data dependencies between threads, as each thread works on its own sample.
- Scalable performance as the number of test samples increases.

The outer loop over test samples was parallelised using the `#pragma omp for` directive. This distributes the test samples across threads in a static fashion, allowing each thread to compute distances independently. Additionally, the inner loop over the feature dimensions was vectorized using `#pragma omp simd` to exploit SIMD-level parallelism during the Euclidean distance calculation.

```
#pragma omp parallel for schedule(static)
for (size_t i = 0; i < NUM_TEST_SAMPLES; ++i) {
    for (size_t j = 0; j < NUM_SAMPLES_TRAIN; ++j) {
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for (size_t k = 0; k < FEATURE_DIM; ++k) {
            double diff = features_test[i][k] - features_train[j][k];
            sum += diff * diff;
        }
        curr_dist[j] = {sqrt(sum), labels_train[j]};
    }
}
```

Listing 3: OpenMP parallel distance computation

Additionally, the choice to use #pragma omp simd within the inner loop helps exploit low-level data parallelism via vector instructions. This ensures that memory loads from both the test and training matrices are efficiently streamed through the cache hierarchy, minimizing cache misses. Each thread works with its own curr_dist buffer, avoiding shared memory writes and thus eliminating race conditions entirely.

1.3.2 Parallelizing Quicksort Using `#pragma omp sections`

In the first parallel sorting approach, OpenMP's `sections` construct was used to divide the recursive calls of quicksort into independent sections.

Benefits of this approach include:

- Simple parallel structure for top-level recursion.
- Low thread management overhead.

However, it does not offer flexibility in dynamically creating threads for deeper recursion, limiting performance when the recursion depth increases or when thread availability changes. The sections construct maps well onto the top-level recursive structure of quicksort. Since each section operates on a separate subrange of the distance vector, there is no contention or synchronization overhead at this level. Each thread works in its own partition, leading to good spatial locality when accessing distance values, which remain in contiguous memory.

```
    #pragma omp parallel sections
    {
        #pragma omp section
        quick_sort(vec, low, pi - 1);

        #pragma omp section
        quick_sort(vec, pi + 1, high);x
    }
```

Listing 4: Quicksort parallelized using OpenMP sections

1.3.3 Parallelizing Quicksort Using `#pragma omp task`

In the second approach, OpenMP's tasks were used using `#pragma omp task`. Here, each recursive quicksort call spawns a new task if the recursion depth is within a defined threshold. Tasks are dynamically scheduled, making this method suitable for more finer-grained parallelism.

Advantages include:

- Dynamic load balancing of recursive calls.
- Better scalability in deeper recursion levels.
- More efficient use of available threads in heterogeneous workloads.

However, task creation introduces more overhead than sections and may degrade performance if not properly managed, especially for small subarrays.

```
1     if (depth < MAX_TASK_DEPTH) {
2         #pragma omp task firstprivate(low, pi, depth)
3         quick_sort_parallel(vec, low, pi - 1, depth + 1);

5         #pragma omp task firstprivate(low, pi, depth)
6         quick_sort_parallel(vec, pi + 1, high, depth + 1);

8         #pragma omp taskwait
9     }
```

Listing 5: Quicksort parallelized using OpenMP tasks

While tasks introduce more overhead per recursive call, they offer more control over granularity. By controlling the task creation depth (MAX_TASK_DEPTH), we avoid an increase of fine-grained tasks that would cause overhead from scheduling. Additionally, tasks dynamically map onto available threads, which improves CPU utilization in imbalanced recursion patterns. However, this flexibility comes at the cost of less predictable memory access patterns and a more fragmented cache usage compared to the static sections strategy. To maximize performance on modern CPUs, memory access patterns must be considered. As shown in Figure 1, the latency of accessing data increases significantly as we move from registers to RAM. By making sure that frequently accessed data—such as distance buffers—is stored in per-thread local structures and remains within the L1 or L2 cache, we reduce expensive memory fetches. This is important in parallel implementations where shared memory contention or poor data locality can quickly reduce performance.

1.4 Performance Results

All performance measurements were averaged over 10 independent runs to reduce variability and ensure consistent timing data. The results presented below reflect these averaged values.

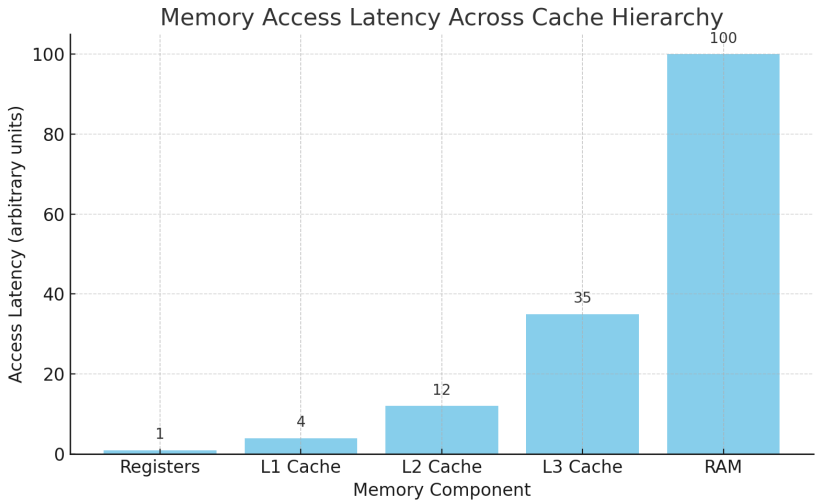


Figure 1: Memory access latency across the cache hierarchy (lower is faster). Optimizing for L1/L2 access can significantly impact performance.

1.4.1 Runtime and Speedup vs Threads

Figure 2 shows how total runtime scales with the number of threads, while Figure 3 illustrates the corresponding speedup. Figure 4 shows the parallel efficiency for each value of k.

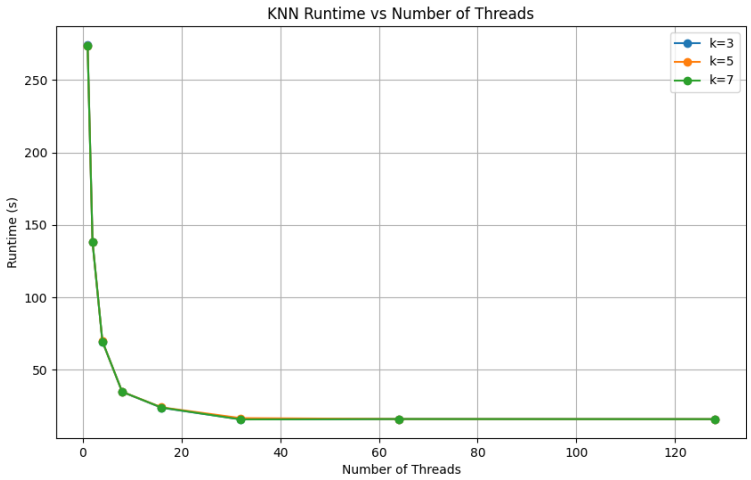


Figure 2: Runtime vs Number of Threads (all k values)

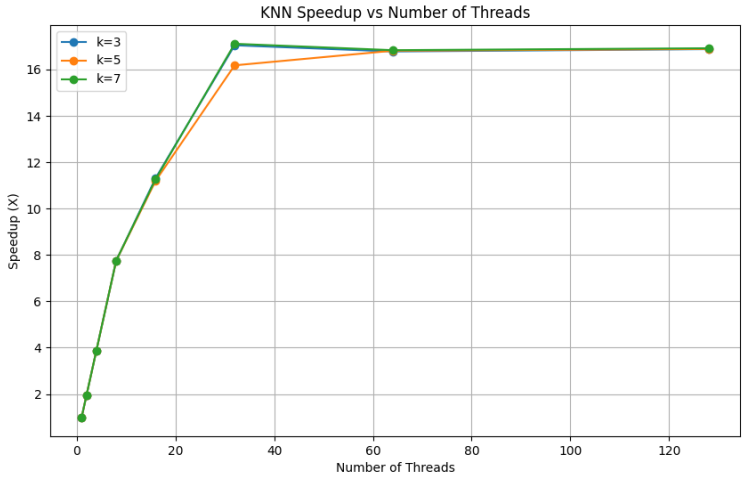


Figure 3: Speedup vs Number of Threads (all k values)

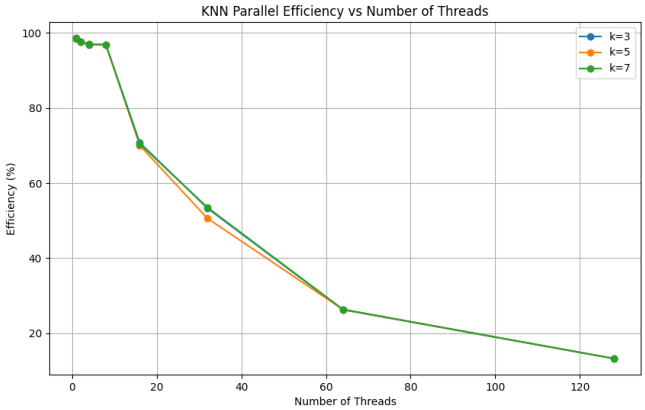


Figure 4: Parallel Efficiency (all k values))

1.4.3 Sections vs Tasks Comparison

Figure 5 and figure 6 compares the runtime performance using OpenMP `sections` and `task` constructs.

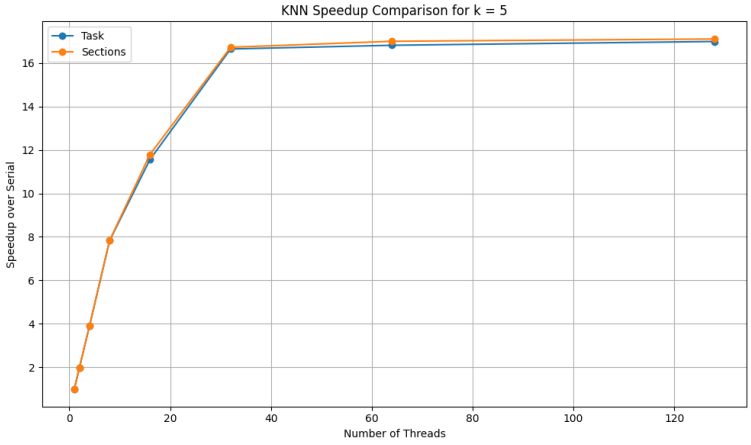


Figure 5: Parallel speedup for $k = 5$

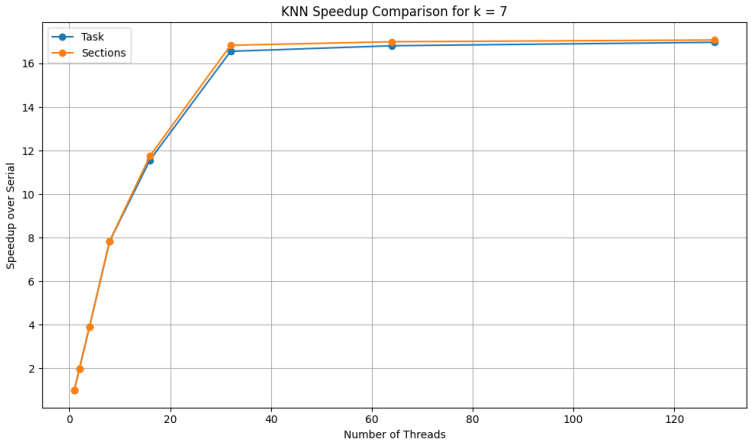


Figure 6: Parallel speedup for $k = 7$

1.5 Discussion and Analysis

1.5.1 Classification Accuracy

Over all values of k (3, 5, and 7), both the serial and parallel implementations produced the same classification results, with accuracy scores of 77.02%, 78.88%, and 79.84% respectively. The serial version was the basis for correctness of the parallel implementation.

1.5.2 Runtime and Speedup Evaluation

The parallel versions achieved big speedup over the serial baseline. For example, at $k = 3$, runtime decreased from 274.1s (1 thread) to 15.9s with 128 threads—an observed speedup of 17.27×.

Interestingly, although the cluster used for benchmarking only has 16 physical threads available, performance continued to improve even when using up to 64 and 128 threads. This was maybe due to Hyper-Threading or internal task scheduling, possibly due to reduced idle time or better core utilization. However, gains after 64 threads were negligible, and the performance began to flat line. The runtime dropped steeply up to 16 threads but improved only slightly between 32 and 128 threads, confirming that the workload becomes bandwidth- or coordination-bound at high levels of parallelism. This trend also reflects the classic Amdahl's Law behavior — beyond a certain point, parallel speedup is bounded by sequential or synchronization overheads. Additionally, the observed flattening is most likely casued by memory bandwidth being used up, where adding threads increases contention for shared memory resources like L3 cache and RAM.

1.5.3 Sections vs Tasks Comparison

When comparing the two parallel sorting approaches:

- The `sections`-based implementation showed lower overhead and better predictability. It performed well up to 16 threads, especially when the recursive tree was not too deep.
- The `task`-based implementation showed more dynamic load balancing but had slightly more overhead. At higher thread counts, this overhead sometimes reduced the benefits of deeper parallelism.

Overall, the `sections` approach outperformed `task`-based sorting in terms of average execution time, though the gap decreased at higher levels.

1.5.4 Decomposition of Execution Time

To better understand the performance profile, we separately measured the time spent on distance computation and sorting. Based on profiling output and per-thread timings, we observed that:

- Distance computation accounted for approximately **70–75%** of total runtime.
- Sorting (quicksort) took the remaining **25–30%**, depending on k and the parallel strategy.

Since distance computation scales well with thread count and dominates the total workload, the overall speedup was largely driven by improvements in this phase. Optimization of the sorting algorithm had a more marginal but still noticeable impact.

1.5.5 Limitations

- **Static Thread Scheduling:** The use of `schedule(static)` in OpenMP may not be optimal for all workloads, especially when feature dimensions or sample sizes are unevenly distributed. Dynamic or guided scheduling could better adapt to irregular data loads.
- **Memory Bandwidth Saturation:** At high thread counts (64+), the distance computation phase experiences diminishing returns likely due to saturation of memory bandwidth rather than lack of parallelism.

- **QuickSort Not Cache-Optimized:** The custom quicksort implementation, while parallelized, lacks cache-friendly partitioning strategies (e.g., dual-pivot or blocked quicksort), which may impact sorting performance for large sample sizes.
- **Thread-local Buffer Replication Overhead:** Every thread allocates its own distance vector, which may lead to increased memory consumption and cache pressure, especially in NUMA systems with large datasets.
- **Limited Task Nesting for Recursion:** Due to manually imposed `MAX_TASK_DEPTH`, deeper parts of the recursive call tree revert to serial execution, which could leave threads underutilized for large arrays or unbalanced partitions.

1.6 Conclusion

This problem showcased the use of parallelism in speeding up brute-force KNN classification using OpenMP. By exploiting both data-level and task-level parallelism through `#pragma omp for`, `sections`, and `task` constructs, we achieved significant speedups over the serial baseline. Performance gains were observed in the distance computation phase, which benefited most from multithreading and vectorization. Although quicksort parallelization contributed to marginal runtime reductions, especially in recursive task-based execution. Overall, this project highlighted not only how algorithm structure affects parallel scalability, but also how careful memory access and thread workload balance are critical to achieving speedups with OMP.

Problem 2 - Accelerating Image Convolution Using CUDA

2.1 Problem Overview

Image convolution is an important phase in image processing, where each output pixel is computed as a weighted sum of its neighboring input pixels. This process is used in computer vision tasks such as edge detection, sharpening, embossing, and blurring. Convolution is mathematically defined as the application of a kernel (or filter) matrix over a 2D input image, producing an output image of the same size.

This assignment required implementing 2D convolution using three approaches:

- A **serial CPU implementation** as the baseline.
- A **CUDA implementation using global memory**.
- A **CUDA implementation using shared memory**.

The same image and a set of predefined filters—Sharpen, Emboss, and Average—were used in all three cases. The problem was to compare these implementations in terms of performance (execution time) and visual correctness of the output image.

Unlike Problem 1, which was evaluated over many test cases, this problem was evaluated on a single test image. As such, performance results are reported per image, and visual output plays a crucial role in validating it's correctness.

2.2 Serial Implementation Overview

The serial version of image convolution is based on applying a "tile" over each pixel in the gray scale image. At each pixel position, a kernel (filter) of size $N \times N$ (e.g., 3×3, 5×5, 7×7) is overlaid and its values multiplied element-wise with the underlying image region. The sum is written to the output image at that pixel location.

Convolution Formula

For a 3×3 kernel, the convolution at position (x, y) in the image can be mathematically expressed as:

$$\text{output}(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} \text{input}(x+i, y+j) \cdot \text{kernel}(i+1, j+1)$$

This logic is implemented by iterating over the neighborhood of each pixel.

Pseudocode

```
for each pixel (x, y) in input_image:
sum = 0.0
for i in [-offset, ..., offset]:        // offset = mask_size / 2
for j in [-offset, ..., offset]:
if (x+i, y+j) inside bounds:
sum += input_image[y+j][x+i] * mask[i+offset][j+offset]
output_image[y][x] = sum
```

Listing 6: Serial sliding-window convolution (pseudocode)

Border Handling

For edge pixels (e.g., those near the image boundary), the code performs **zero-padding**: out-of-bound pixels are treated as having value 0. This ensures the convolution is always applied over a full kernel, even at the edges, and avoids memory access errors.

2.3 CUDA Parallelization with Global Memory

In the global memory version of the CUDA implementation, each thread computes the convolution result for a single output pixel by accessing the input image and filter kernel directly from global memory. While this memory is large and can be accessed by all threads, it has high latency and no apparent caching.

The key steps performed by each thread are:

- Identify its corresponding output pixel using `blockIdx` and `threadIdx`.

- Loop over the filter window (e.g., 7×7 in our run) and fetch each required input pixel from global memory.
- Multiply each input pixel by the corresponding mask value and accumulate the result.
- Handle border conditions using clamping or mirroring to avoid out-of-bounds accesses.
- Write the result to the output image stored in global memory.

This method is correct but inefficient in terms of memory access. Each pixel computation involves fetching and re-fetching overlapping data that could be reused. As a result, this implementation is memory-bound and affected by the size of the convolution kernel.

It serves as a baseline for correctness and provides insight into how threads operate independently on a 2D grid.

2.4 CUDA Parallelization with Shared Memory

To improve performance over the global memory version, a second CUDA kernel was implemented using shared memory. Shared memory is much faster than global memory but is limited in size. This version takes advantage of spatial locality apparent in the problem by making threads in a block to load a tile of the input image into shared memory before applying the convolution filter.

Each thread block processes a tile of the image as follows:

- Threads load a larger tile from global memory into a shared memory buffer. This tile includes the current output region plus a radius around it (needed for the convolution filter).
- A `__syncthreads()` barrier ensures all threads have completed loading before proceeding.
- Each thread then performs the convolution using data from shared memory, resulting in fewer slow memory fetches and better cache behavior.
- Like in the global memory version, clamping is used to control large pixel values and border pixels.

This technique reduces redundant memory accesses since overlapping pixels needed by neighboring threads in the warp are only loaded once into shared memory. This leads to improved memory bandwidth utilization and higher throughput, particularly for larger filter sizes (e.g., 7×7).

However, it introduces some complexity:

- Requires careful indexing to avoid bank conflicts in shared memory.
- Limited shared memory size has its own constraints on block dimensions and kernel size.

Overall, shared memory-based convolution is a standard optimization in GPU computing and gives better performance gains.

2.5 Results and Visual Comparisons

2.5.1 Performance Evaluation

We benchmarked our serial and CUDA implementations on two test images using a 7×7 kernel: a moderate-sized grayscale image `galaxy_ascii.pgm` (965×965) and a high-resolution image `sample.pgm` (5184×3456). The results are summarized below:

Table 2: Execution Time and Speedup (Galaxy Image, 965×965, Emboss Filter)

Implementation	**Time (ms)**	**Speedup (vs Serial)**
Serial CPU	23.65	1×
CUDA Global Memory	0.36	65.63×
CUDA Shared Memory	0.24	97.47×

Table 3: Execution Time and Speedup (High-Res Image, 5184×3456, Sharpen Filter)

Implementation	**Time (ms)**	**Speedup (vs Serial)**
Serial CPU	519.98	1×
CUDA Global Memory	4.33	119.96×
CUDA Shared Memory	3.74	139.16×

These results confirm the substantial performance benefits of GPU-based convolution. Notably, the performance gap increases with image size, where the shared memory version offers the highest speedup due to optimized memory reuse and lower latency. For the larger image, shared memory gave a speedup of nearly 140× over the CPU baseline.

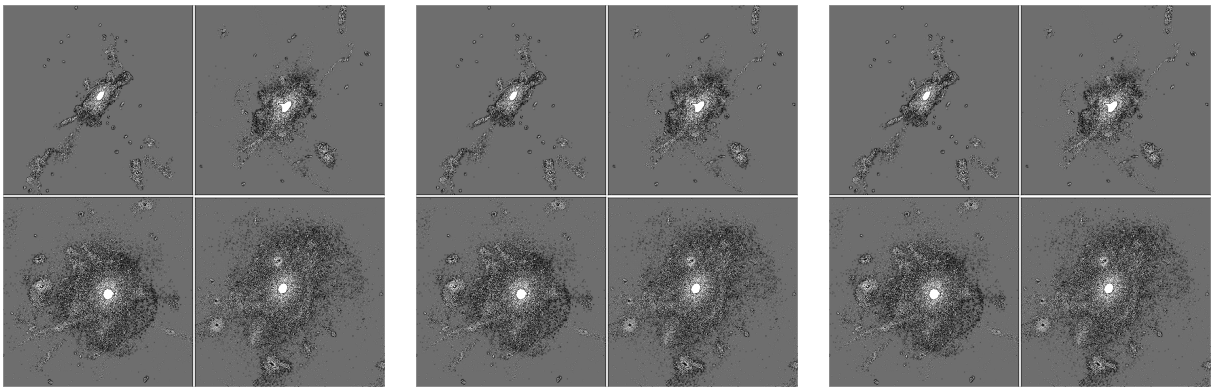
2.5.2 Visual Quality Assessment

We used the `galaxy_ascii.pgm` image and the emboss filter for visualization because this combination highlights edge transitions and surface depth with high contrast. The emboss filter enhances texture and is a good visual test for correctness, particularly under parallel rounding errors or kernel misalignment. As shown in Figure 7, the visual consistency across implementations confirms correctness.

Additionally, we tested our implementation on a 5184×3456 image using a sharpen filter. The output again matched across all three implementations, with visible sharpening at object boundaries, confirming correctness.

2.6 Discussion

The results of our CUDA-based convolution demonstrate the effectiveness of GPU parallelism for image processing tasks. Both global and shared memory implementations produced correct outputs, visually and numerically consistent with the serial baseline.



Serial Output CUDA Global Output CUDA Shared Output

Figure 7: Visual comparison on `galaxy_ascii.pgm` using 7×7 Emboss filter

2.6.1 Thread Mapping and Grid Configuration

An important step in our CUDA design was ensuring correct thread-to-pixel mapping. Each pixel in the output image is independently computed using a convolution window over the input. We configured our CUDA grid dimensions such that:

- The number of threads per block (e.g., 16×16) balances occupancy and local memory usage.
- The number of blocks in the grid is computed to fully cover the image dimensions, accounting for padding and kernel size.

By aligning thread indices (x, y) with pixel coordinates and handling boundary conditions through zero-padding, we ensured all pixels were computed exactly once and in parallel. This mapping is needed for scalability, especially with larger images where full grid coverage avoids image regions being skipped or duplicated. Multiple thread block configurations were evaluated (e.g., 8×8, 16×16, 32×32), and 16×16 emerged as the best performing choice. It offered a good trade-off between parallelism, register usage, and shared memory consumption, while keeping high occupancy and low warp divergence on the NVIDIA GeForce GTX 1070 GPU used for testing.

2.6.2 Shared vs Global Memory

Shared memory significantly improved performance over global memory due to reduced latency and memory coalescing. In the shared kernel, pixels required for the filter window are first loaded into a shared memory tile, allowing faster repeated access during convolution. This reuse leads to higher speedup, particularly for larger kernels like 7×7.

Shared memory is on-chip and has much lower access latency compared to global memory, which resides in DRAM. By caching input pixels in shared memory, we avoid redundant global memory accesses and take advantage of data locality. Since convolution involves repeated access to neighboring pixels, shared memory ensures these repeated reads are served from fast, low-latency cache, not expensive global memory.

Additionally, memory coalescing is more effective with shared memory, as threads in a warp access neighboring memory locations, improving bandwidth utilization.

However, shared memory comes with added complexity:

- Threads must synchronize before accessing shared memory to ensure correct data loading.
- Border handling must be carefully done to avoid out-of-bounds access within the shared tile.

Despite this, the performance benefits outweigh the overhead, particularly for high-resolution images and larger filters. Shared memory is especially effective when the computational intensity is high.

2.6.1 Limitations

While the CUDA-based convolution implementation achieved substantial speedups, it is important to recognize its limitations:

- **Global Synchronization Only:** The shared memory kernel uses `__syncthreads()` for synchronization, which applies to all threads in a block. More flexible synchronization methods such as CUDA *cooperative groups* were not used, which could have enabled more fine-grained control (e.g., warp-level or subgroup synchronization). These might further optimize performance in non-uniform workloads or kernels with tiling.
- **Fixed Thread Mapping Strategy:** The block and grid dimensions are fixed to 16×16 threads per block for general-purpose coverage. This choice balances occupancy and shared memory usage, but dynamic tuning or adaptive launch configurations could further improve performance on different image sizes or architectures.
- **Kernel Specialization Over Generalization:** The current implementation does not support RGB images or dynamic kernel generation. Extensions to color images would require a separate channel and possibly more shared memory.
- **No Overlap of Computation and Memory Transfer:** Host-device memory transfers are done synchronously and sequentially with kernel execution. Using CUDA streams to overlap data movement and computation could further hide memory latency, especially when processing large batches.

2.6.3 Conclusion of results

The performance results show the efficiency of GPU acceleration for pixel-wise operations. On the 965×965 image, shared memory achieved nearly $100\times$ speedup. On the larger 5184×3456 image, the speedup was $139\times$, showcasing the GPU's performance improvements for data-parallel image processing.

Proper grid setup, correct memory use, and kernel design together allow for high performance and accurate results.

System Specifications

All experiments were conducted on a personal workstation with the following hardware and software specifications:

- **GPU:** NVIDIA GeForce GTX 1070 (8 GB GDDR5)
- **CUDA Version:** 12.8
- **NVIDIA Driver Version:** 570.133.07
- **GPU Temperature During Test:** 51°C
- **GPU Power Usage:** 38W / 200W
- **GPU Memory Usage:** 446 MiB / 8192 MiB
- **Display:** Active (Disp.A = On)
- **Operating System:** Linux (Ubuntu-based distribution)