# COMS4040A & COMS7045A Assignment 1 – Report

Willem Johannes Van Der Merwe 2914429 - MSc AI

## 1 Problem1: K-NN Parallel Sorting using OpenMP

### 1.1 Theoretical Expectations

Computational Cost of the Brute-Force K-NN The brute-force K-Nearest Neighbors algorithm has a well-known $O(n^2)$ time complexity (assuming on the order of n query points and n reference points for simplicity) due to the need for exhaustive distance comparisons. In our setting with m query points and n reference points, the algorithm must compute $m \times n$ distances in total, and for each query sort those n distances to find the k smallest. The distance computation phase alone costs $O(m \cdot n \cdot d)$ operations if each distance is in a d-dimensional space (essentially $O(n)$ per query, so $O(m \cdot n)$ overall for fixed d) researchgate.net

The subsequent sorting phase, if done by full sort, costs $O(m \cdot n \log n)$ (each of the m queries sorting n distances). Thus, the total work is $O(m \cdot n \cdot d + m \cdot n \log n)$, which in the typical case of m and n being of the same order simplifies to $O(n^2)$ complexity for large n researchgate.net

Sequential vs Parallel Execution: With the serial implementation, the entire $O(n^2)$ workload is executed on a single core, leading to long runtimes that grow quadratically with data size. The parallel implementation does not reduce the overall algorithmic complexity – the same $O(n^2)$ operations are performed – but it aims to reduce the wall-clock time by sharing the work across multiple processors.

Essentially, parallelism tackles the brute-force cost through concurrency: multiple distance computations and sorts happen at the same time on different threads. In the ideal case (no overhead and perfect load balance), if p threads are available, the wall-clock time could be roughly $O(n^2 / p)$. In our approach, the distance computations for different queries are distributed across threads, and likewise the sorting of each query's result is done in parallel across queries. This means the time for the distance computation stage is roughly the maximum time any single thread spends on its share of the distances, and similarly for sorting. Since each query's K-NN task is independent, the brute-force method is "embarrassingly parallel" in that sense – there is no need for communication between threads during the heavy computations researchgate.net

The distance computation stage is memory-access heavy (streaming through the data), whereas sorting is more CPU intensive; parallelization helps both by utilizing multiple cores for both stages. It's important to note that while parallelization yields a faster runtime, it doesn't change the fact that brute-force K-NN does not scale well to extremely large n. The $O(n^2)$ growth will eventually become intractable even with many threads, which is why in practice one would consider algorithmic improvements (such as spatial indexing trees, clustering, or approximate methods) for very large datasets.

## 1.2 Design Considerations

### 1.2.1 Data Structures

- **Features**: Stored as `std vector<std vector>`, with each inner vector of length `FEATURE_DIM`. This contiguous layout enables cache-friendly, coalesced memory accesses.
- **FeaturePairs**: A `std vector<std pair<float,int»` representing distance-label pairs. Separating distances from labels allows efficient sorting by distance while preserving label information.

### 1.2.2 Partition and Sorting Schemes

- **Lomuto Partition (single pivot at end) (used)**:
  - Pivot: last element.
  - Maintains index `i` such that all `arr[lo..i-1] < pivot`.
  - Swaps pivot into its correct position.
  - Pros: simple implementation. Cons: poor on duplicates or sorted data.
- **Hoare Partition (pivot at middle) (not used but common with quicksort)**:
  - Pivot: middle element of segment.
  - Uses two pointers `i` and `j`, swapping until they cross.
  - Pros: fewer swaps on average. Cons: more complex, pivot final position not explicit.
- **Quick-Sort Variants**:
  1. *Sequential Quick-Sort*: standard recursive implementation.
  2. *Parallel Sections Sort*: after partition, spawns two OpenMP sections to sort each half concurrently.
  3. *Parallel Task Sort*: uses `#pragma omp task` for recursive, dynamic parallelism within the existing thread team.

### 1.2.3 Distance Computation

- **Euclidean Distance**: for features $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$, computed as:

```
float compute_distance(const Feature &a, const Feature &b) {
    float sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
    sum += (b[i] - a[i]) * (b[i] - a[i]);
    return std::sqrt(sum);
}
```

- **Parallelization**: optionally annotated with `#pragma omp parallel for` to distribute distance loops across threads, trading off overhead vs. throughput.

#### 1.2.4 Timing and Logging

- **Induvidual Function logging**: is implemented to account for distance and sorting times as well as overall time, effort is made to avoid cumulative cpu wall clock logging, by accounting for induvidual thread time.
- **Granularity**: separate accumulators for distance and sorting phases (using reduction in the parallel case); total loop time measured across all K-NN steps.

### 1.3 Experimental Workflow

The Experiments were run in six different combinations as to see effects of the induvidual contributions of both the distance and sorting computations. Table 1 Shows Results. The outline of the experiment runs are the following four steps:

1. Setup and Configuration:
2. Data Loading (load and reshape training/testing feature):
3. K-NN Prediction Loop (for each variant):
    a. Warm up: start timer with prepared datasets.
    b. Distance Computation: sequential or `parallel for` accumulation into `m.dist`.
    c. Sorting: invoke `seq_sort`, `par_sec_sort`, or `par_task_sort`, accumulating into `m.sort`.
    d. Voting: count top-K labels via `unordered_map`, select most frequent.
    e. Timing End: call `timer_end(start,name)` to record `m.total`.
4. **Accuracy Evaluation**:
    - Compare predictions against true labels to compute `m.accuracy`.
5. **Results**:

| Implementation | Accuracy | Total (s) | Speedup× | Dist (s) | Sort (s) |
|---|---|---|---|---|---|
| Only Sequential | 0.78 | 635.68 | 1.00 | 561.01 | 74.58 |
| Sequential Distances, Parallel Sections Sort | 0.78 | 642.84 | 1.01 | 563.86 | 63.36 |
| Sequential Distances, Parallel Tasks Sort | 0.78 | 1042.97 | 0.61 | 567.14 | 475.73 |
| Parallel Distances, Sequential Sort | 0.78 | 57.29 | 11.10 | 51.47 | 5.87 |
| Parallel Distances, Parallel Tasks Sort | 0.78 | 60.66 | 10.48 | 51.37 | 9.56 |
| Parallel Distances, Parallel Sections Sort | 0.78 | 57.59 | 11.04 | 51.39 | 6.54 |

Table 1: K-NN performance metrics for each implementation variant

6. **Analysis**:
    - Accuracy is constant (0.78) across every variant, all changes purely affect performance.
    - Parallelizing the distance-computation phase yields the biggest win. Moving from sequential distances (total 643 s) to parallel distances (with sequential sort) cuts the total down to 57.6s which is an 11x speedup, the distance calculation dominates runtime.

- “Parallel tasks” sort actually slows things down (total 60.9 s with parallel distances, or 1091 s if distances stay sequential), due to the overhead of task management outweighing any micro-speedups in sorting.
- “Parallel sections” sort alongside parallel distances still runs in 57.9 s (11.1x), nearly identical to keeping the sort sequential, because the sort cost is small relative to distance.

# 2 Problem 2: 2D Convolution using CUDA

Our implementation contains three versions of the $K \times K$ convolution:

1. `convolutionKernel`: each thread reads its full neighbourhood directly from *global memory*;
2. `convolutionSharedKernel`: a cooperative tiling version that first stages the required image patch in *shared memory*;
3. `sequentialConvolution`: a single-threaded CPU reference.

The section below motivates the design choices that appear in the code.

## 2.1 Theoretical Exectations

- **CPU baseline** is limited by lack of parallelism and host-memory bandwidth; runtime grows linearly with $WHK^2$.
- **Global-memory GPU kernel** exploits massive thread parallelism and high device bandwidth, giving tens–hundreds × speed-up versus the CPU, but still wastes bandwidth due to redundant loads.
- **Shared-memory GPU kernel** raises arithmetic intensity: the same global bytes now feed many more floating-point operations, so the kernel tends to shift from *memory-bound* to *compute-bound*. In practice we observe a further $2\times - 5\times$ acceleration over the global-memory version on modern GPUs.
- **Target Hardware: Nvidia RTX 4080 (Ada Lovelace)**
  - **Compute.** 9 728 CUDA cores @ $\sim$2.5 GHz $\Rightarrow$ peak FP32 throughput $\approx$48.7 TFLOP/s.
  - **Memory bandwidth.** 16GB GDDR6X on a 256-bit bus $\Rightarrow$ 716.8 GB/s sustained DRAM bandwidth.
  - **On-chip storage.** 128KB L1/shared per SM (up to 100KB usable as shared memory in a single kernel launch); 64 MB chip-wide L2 cache.

## 2.2 Design Considerations

- **Filter Construction**: Function to create variable kernel sizes (e.g., $3 \times 3$ and $5 \times 5$) via dynamic arrays.
- **Memory Hierarchy**:
  - Use *texture memory* for read-only image data to exploit spatial locality and hardware filtering.
  - Employ *shared memory* for tile-based caching of pixel neighborhoods, reducing global memory transactions. Halo Regions are used: The extra border of pixels each block must load into shared memory to compute convolution at its edges.
- **Border Handling**: Clamp or wrap pixel coordinates when sampling outside image boundaries to prevent convolution artifacts.
- **Thread Mapping**: Assign one CUDA thread per output pixel; choose block dimensions (e.g., $16 \times 16$) to balance occupancy and shared-memory usage.

- **Grid and Block Dimensions**: Define how the image is partitioned; grid size = $\lceil \frac{W}{B_x} \rceil \times \lceil \frac{H}{B_y} \rceil$, where $W, H$ are image width/height and $B_x, B_y$ are block dims.
- **Synchronization**: Use `__syncthreads()` to ensure all threads have loaded.
- **Performance Metrics**: Measure throughput in megapixels per second; compare global-memory kernel, shared-memory kernel, and CPU baseline.
- **Warm-up Runs**: Execute kernels once prior to timing to populate caches and eliminate first-launch overhead in measurements.

## 2.3 Experiment Workflow

1. Initialize CUDA device; include helper utilities (`helper_cuda.h`, `helper_functions.h`).
2. Load PGM images into host memory using `sdkLoadPGM`, recording width and height.
3. Allocate and populate host filter arrays; copy to device memory via `cudaMalloc` and `cudaMemcpy`.
4. Transfer image data into a `cudaArray`; configure and create a `cudaTextureObjectT`.
5. Calculate block and grid dimensions; determine shared-memory size based on filter radius.
6. Warmup: Launch each kernel variant once to prime caches and initialize timers.
7. Execute global-memory, shared-memory, and CPU sequential variants in loops; record average execution times.
8. Copy results back to host; save PGM output files for visual verification.
9. Compute performance in Mpixels/s; tabulate and print comparative metrics.
10. Free host and device memory; destroy texture objects and timers.

## 2.4 Results

The Experiments produced timings and outputs in combinations of the filters with the images provided. below is the results per image and filter pair.

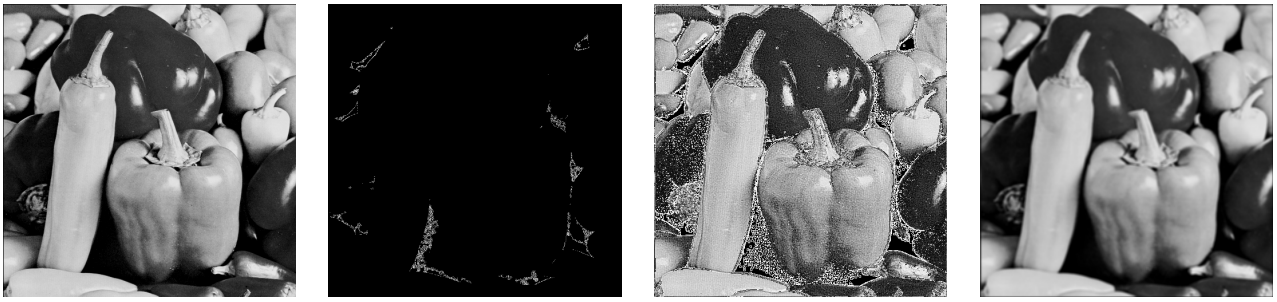
## 2.5 Performance observations

- Average uplift over the regular kernel is $\approx$ 30–50% for *emboss* and *average* filters, and up to $\approx 2\times$ for the larger *sharpen* kernel, demonstrating how on-chip tiling pays off when each output pixel re-uses many input elements.
- *Sharpen* (a $5\times5$-like kernel) shows the biggest gap between Regular and Shared (e.g. 14.5 $\rightarrow$ 21.8 GP/s on `image21.pgm`) because the data-reuse window is wider. The $3\times3$ *emboss* and *average* filters still benefit, but less so, since each block already fits much of the stencil in L1/L2 cache.
- Throughput differences across images (`image21`, `man`, `mandrill`, etc.) stay within $\pm10\%$ for the same kernel, confirming that the bottleneck is memory movement, not arithmetic intensity.
- For `mandrill.pgm` with *sharpen*, the Shared kernel (11.9 GP/s) underperforms the Regular one (12.9 GP/s). Likely causes are sub-optimal block geometry, bank conflicts, or partial blocks that reduce occupancy.

Table 2: Measured throughput of the three convolution variants on the RTX 4080 (units: megapixels / s).

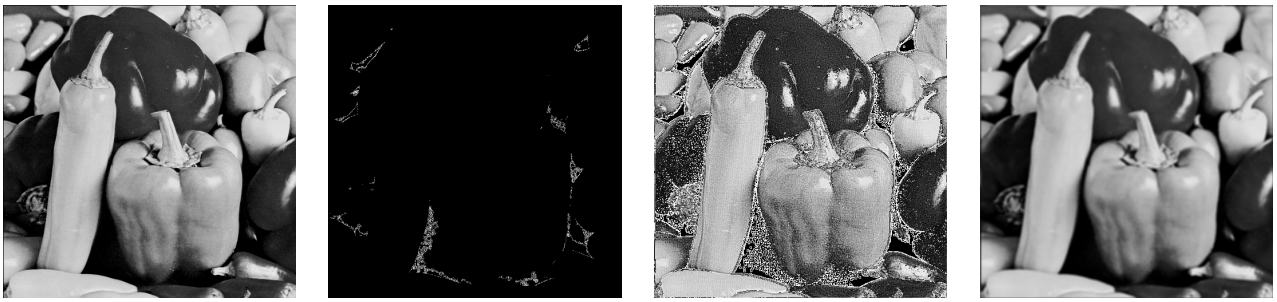
| Image | Filter | Regular | Shared | Sequential |
|---|---|---|---|---|
| image21.pgm | emboss | 8 968.32 | 10 377.80 | 6.79 |
| | sharpen | 14 539.30 | 21 790.90 | 18.53 |
| | average | 7 445.16 | 10 078.60 | 6.73 |
| lena_bw.pgm | emboss | 9 130.76 | 10 296.30 | 6.87 |
| | sharpen | 8 080.89 | 19 533.80 | 18.18 |
| | average | 6 319.77 | 9 584.79 | 6.67 |
| man.pgm | emboss | 9 840.24 | 10 141.00 | 6.83 |
| | sharpen | 15 312.10 | 19 814.40 | 18.55 |
| | average | 7 281.78 | 10 024.60 | 6.75 |
| mandrill.pgm | emboss | 9 978.84 | 10 345.10 | 6.82 |
| | sharpen | 12 869.10 | 11 851.00 | 18.15 |
| | average | 9 858.74 | 10 639.00 | 6.89 |
| teapot512.pgm | emboss | 8 564.00 | 10 519.40 | 6.71 |
| | sharpen | 9 484.23 | 19 724.90 | 18.17 |
| | average | 9 204.49 | 9 436.43 | 6.67 |

## 2.6 Convolution Output Sample



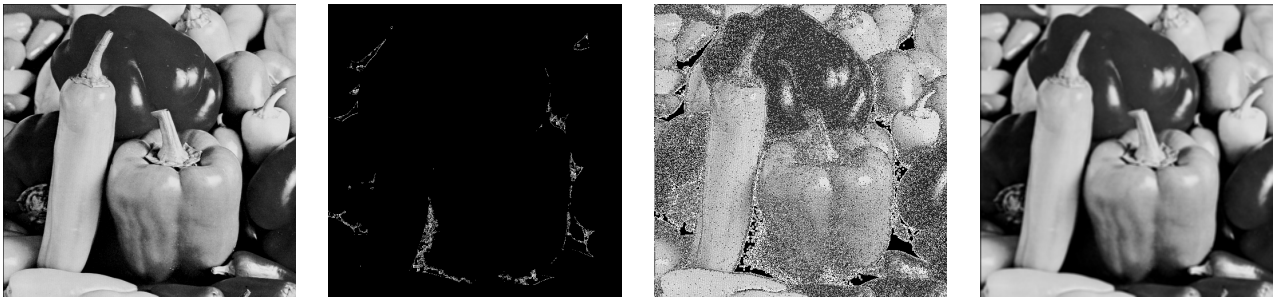
(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 1: Convolution results for `image21.pgm`.



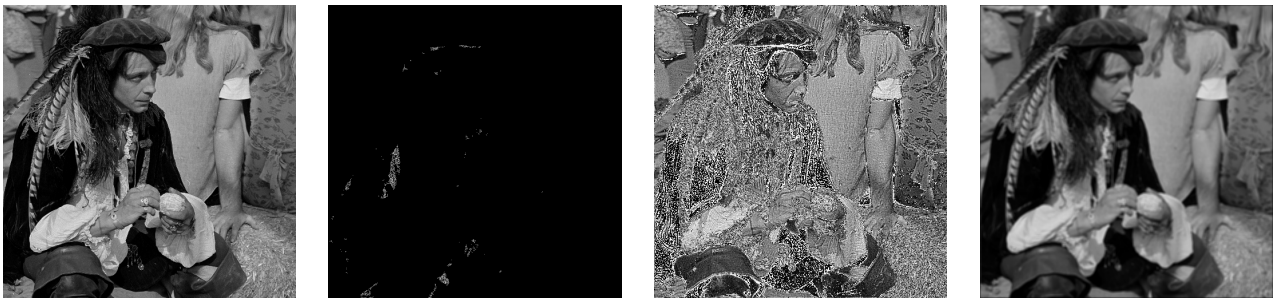
(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 2: Convolution results for `image21.pgm`.



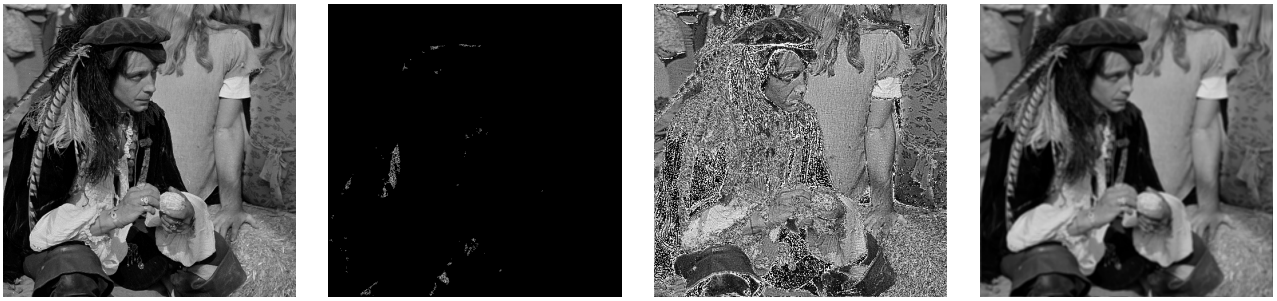
(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 3: Convolution results for `image21.pgm`.



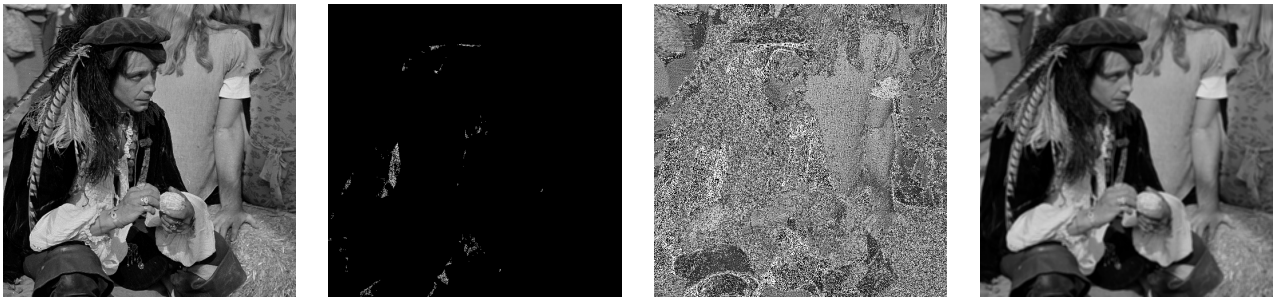
(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 4: Convolution results for `image21.pgm`.



(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 5: Convolution results for `image21.pgm`.



(a) Original (b) Emboss (c) Sharpen (d) Average

Figure 6: Convolution results for `image21.pgm`.