Parallelization of Edge Detection using OpenMP and CUDA

[Mustafa Yılmaz]

[Ramazan Yel]

May 20 , 2024

1 Introduction

This report details the implementation and performance comparison of edge detection using the Sobel filter in a parallel computing environment. The parallel algorithms are implemented using OpenMP for CPU and CUDA for GPU. The results are analyzed in terms of execution time, speed-up, and efficiency with varying parameters such as the number of threads and scheduling methods for OpenMP, and the number of blocks and threads for CUDA.

2 Pseudocode for Parallel Algorithm

```
function sobelFilterCUDA(gray_img, edge_img, width, height)
  allocate d_gray_img on device
  allocate d_edge_img on device
  copy gray_img to d_gray_img
  launch sobelFilter kernel
  copy d_edge_img to edge_img
  free d_gray_img and d_edge_img

function sobelFilterOpenMP(gray_img, edge_img, width, height, num_threads)
  set number of threads
  parallel for each pixel (y, x) in gray_img
    compute gx, gy using Sobel operator
    compute edge magnitude and store in edge_img
```

3 Foster's Parallel Algorithm Design Steps

Parallel programming permeates various aspects of daily digital interactions. Below are three notable examples:

3.1 Partitioning:

- Data Parallelism: We divided the image data among multiple threads/blocks for concurrent processing. Each pixel's edge detection is computed independently based on its neighboring pixels.

3.2 Communication:

- Minimal communication is required as each thread/block works on a separate portion of the image. Boundary pixels need special handling but are processed independently.

3.3 Agglomeration:

- The algorithm is refined by grouping pixel operations together to reduce the overhead of managing too many threads/blocks. This also helps in balancing the workload among the available processing units.

3.4 Mapping:

- For OpenMP, the work is distributed across CPU cores using thread parallelism. We set the number of threads and use dynamic scheduling to handle load imbalance.
- For CUDA, the work is mapped onto GPU threads organized into blocks. We choose the grid and block sizes to maximize GPU utilization.

4 Parallelism Type

Data Parallelism: Each pixel's edge detection is independent of others, making it suitable for data parallelism. Both OpenMP and CUDA implementations focus on processing different portions of the image simultaneously.

5 Experiment Results

5.1 Timing Results

Num Threads	Default (s)	Static 1 (s)	Static 100 (s)	Dynamic 1 (s)	Dynamic 100 (s)	Guided 100 (s)	Guided 1000 (s)
1	0.3124	0.3884	0.3029	0.6993	0.2993	0.2918	0.2913
2	0.1568	0.1983	0.1537	0.9180	0.1609	0.1457	0.1461
4	0.0785	0.1034	0.0794	0.9045	0.0821	0.0729	0.0736
8	0.0830	0.0928	0.0929	0.7817	0.0580	0.0487	0.0529

5.2 Speed-Up

Num Threads	Default	Static 1	Static 100	Dynamic 1	Dynamic 100	Guided 100	Guided 1000
1	0.9348	0.7511	0.9631	0.4173	0.9741	1.0000	1.0017
2	1.8617	1.4715	1.8988	0.3180	1.8135	2.0027	1.9966
4	3.7166	2.8212	3.6749	0.3226	3.5545	4.0027	3.9630
8	3.5157	3.1444	3.1405	0.3733	5.0293	5.9926	5.5161

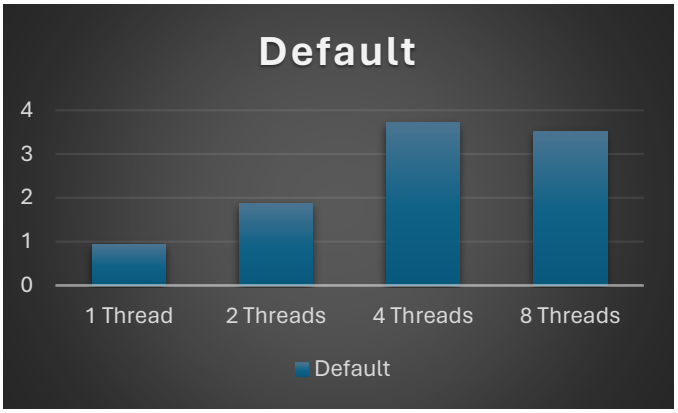
5.3 Efficiency

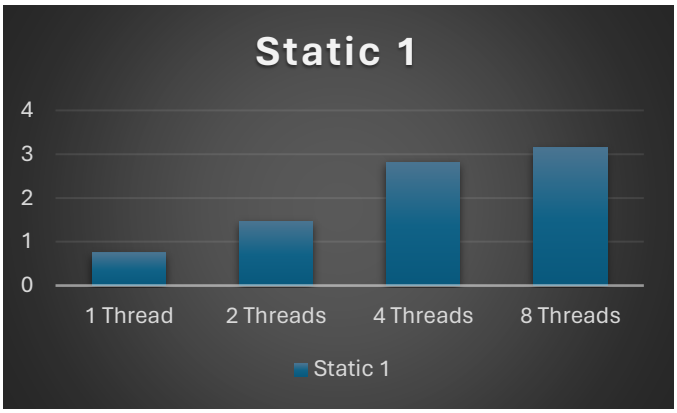
Num Threads	Default	Static 1	Static 100	Dynamic 1	Dynamic 100	Guided 100	Guided 1000
1	0.9348	0.7511	0.9631	0.4173	0.9741	1.0000	1.0017
2	0.9309	0.7357	0.9494	0.1590	0.9067	1.0014	0.9983
4	0.9291	0.7053	0.9187	0.0807	0.8886	1.0007	0.9908
8	0.4395	0.3930	0.3926	0.0467	0.6287	0.7491	0.6895

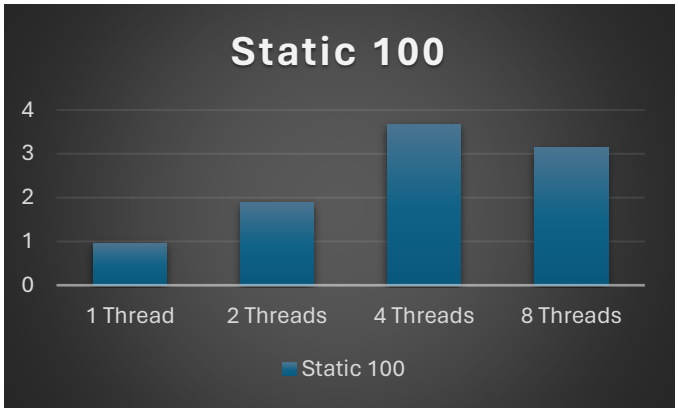
5.4 CUDA Results

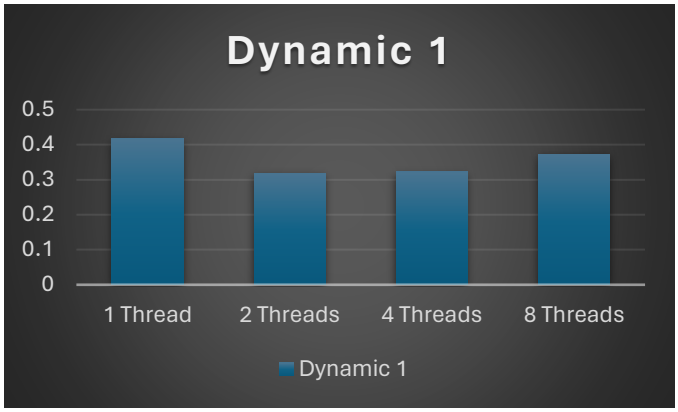
Blocks x Threads		Timing (ms)	Speed-Up	Efficiency
8 x 16		0.48	607.08	4.74
8 x 64		0.16	1823.75	3.56
16 x 16		0.5086	573.82	2.24
16 x 64		0.1830	1594.54	1.56
32 x 64		0.1152	2532.64	1.24
64 x 64		0.101504	2874.48	0.70
128 x 128		0.091808	3178.13	0.19

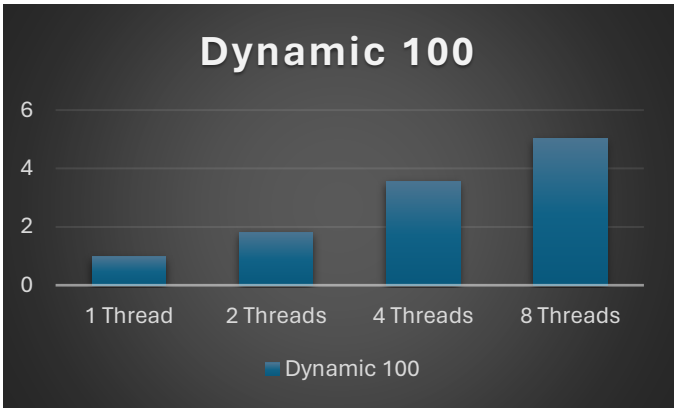
6 Speed-Up Charts

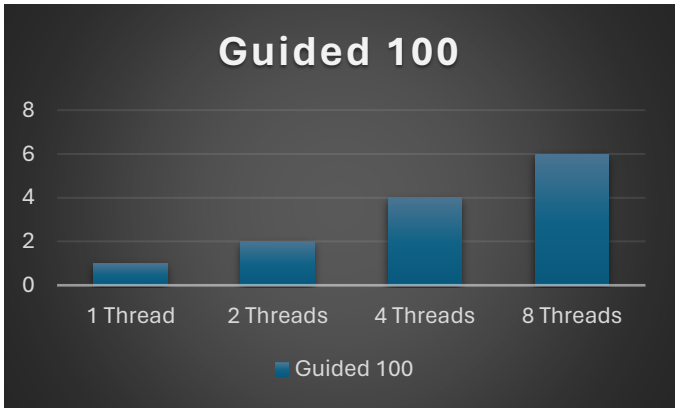


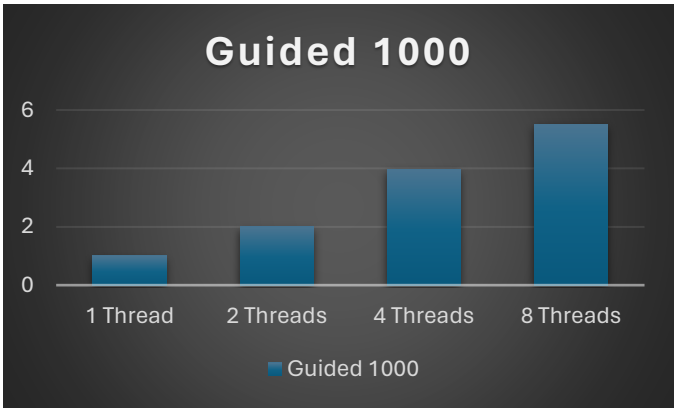












7 Checksum

md5sum seq_out.jpg → 778a7dbb7b882526ac93d796c16f9294

md5sum cuda_out_omp.jpg → 778a7dbb7b882526ac93d796c16f9294

md5sum cuda_out.jpg → 778a7dbb7b882526ac93d796c16f9294

8 Conclusion

When comparing the performance across both OpenMP and CUDA implementations:

For OpenMP, the Guided 100 Schedule was the most effective, providing the best speed-up of 5.9926 with 8 threads.
For CUDA, the 128 x 128 blocks x threads configuration was the most efficient, achieving a remarkable speed-up of 3178.13.

These results demonstrate that the appropriate scheduling strategy or configuration significantly impacts parallel computation performance. While OpenMP's Guided 100 schedule was the best among CPU-based parallel strategies, CUDA's 128 x 128 configuration vastly outperformed all OpenMP schedules due to its ability to leverage the massive parallelism available in modern GPUs.