Name : Muhid Qaiser

Roll No. : 22i-0472

Section : AI-B

Assignment : 4

Report: CPU vs GPU Convolution Performance

In this Assignment, I try to compare performance of convolution operation on CPU and GPU. I test different GPU configuration to check which one is fastest and how it compare to normal CPU implementation.

Introduction

This report goes through the performance comparison of a 3x3 Sobel X filter applied using CPU, a naive GPU implementation, and an optimized GPU implementation. The main aim was to see how memory access patterns and synchronization affect speed and correctness.

First let's clarify some terms,

Work-Group (WG) : A work-group is a group of threads (work-items) that execute together and can share local memory. It computes a tile (block) of the result matrix in matrix multiplication.

Memory Access Patterns & Synchronization

CPU Memory Access Pattern

The CPU version is pretty basic. It just loops over the image, skips the borders, and for each pixel it reads the 3x3 region and does the sum. This is:

- Row-major access
- Sequential and cache-friendly
- No need for any synchronization because it's serial

Naive GPU Memory Access

The naive kernel (conv_naive) assigns one thread per pixel and uses global memory only.

- Each thread reads from global memory directly for its 3x3 window.
- This leads to redundant memory accesses, multiple threads read overlapping data.
- No shared memory used.

- Synchronization isn't required here either because each thread only writes to its own output pixel.

Problems:

- Global memory is slow (100x slower than shared).
- Threads don’t collaborate or reuse data, which wastes memory bandwidth.

Optimized GPU Memory Access

The optimized kernel (conv_opt) fixes the problem by loading shared memory tiles.

- Each work-group loads a tile of the image into shared memory.
- Threads cooperatively load the region.
- Then they compute convolution using the faster shared memory.

This uses:

- Shared memory for fast access,
- Thread synchronization within the group (barrier) to wait till all data is loaded,
- Less global memory traffic.

Benefits:

- Way less redundant global loads,
- Better memory coalescing,
- Speedup increases as local work-group size increases (to a point).

Limitations:

- Limited number of worker-items per work-group due to device constraints.
- Max item in each dimension is 1024.

- One work-group can have a maximum of 256 threads. Hence why i use filters of size 8x8, 16x16 and 32x8 as 32x32 is more than 256, exceeding maximum number of threads in a work-group.
- And we can compute up-to 1024x1024 result matrices in one kernel launch.
- So each group can have 256 threads where each thread calculates value of a pixel in the resultant matrix. And total number of global threads are 1024x1024.

CPU Implementation

I first run convolution on CPU. The time it take is **173.442 ms**. This is our baseline, because CPU is normally slower for this type of task, but it still important for compare.

Naive GPU Implementation (16x16)

Then I try simple GPU version with block size 16x16. It give us time of **7.0268 ms**, which is **24.6829x** faster than CPU. This result is correct, and already show big improvement from using GPU instead of CPU.

Optimized GPU Implementation

We try three different block size to see which one give better performance.

1. **Block Size 8x8**
   Time = **12.056 ms**
   Speedup vs CPU = **14.3864x**
   Speedup vs naive GPU = **0.582847x**
   This version is slower than naive GPU. Maybe because 8x8 block is too small and not use full power of GPU. It can also be slower due to bad overhead to computation ratio with 8x8 matrix.

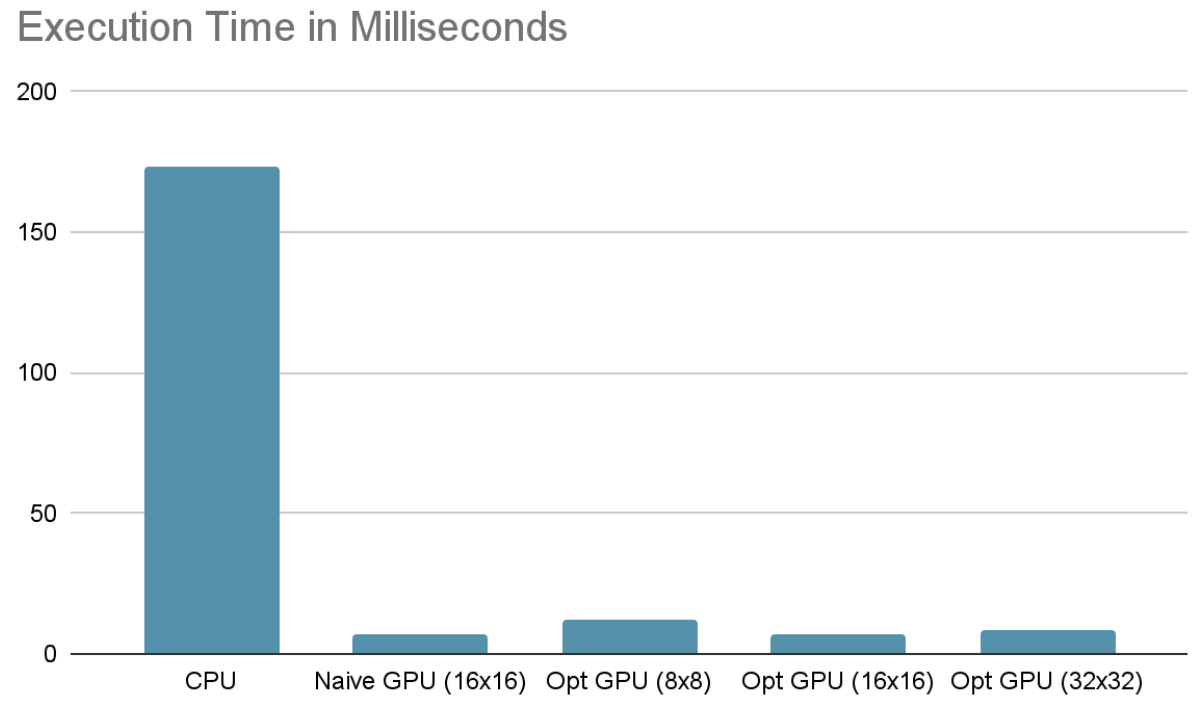
2. **Block Size 16x16**
   Time = **6.7663 ms**
   Speedup vs CPU = **25.6332x**
   Speedup vs naive GPU = **1.0385x**
   This is best version. It run faster than both CPU and naive GPU. It give correct output, and best speed overall.

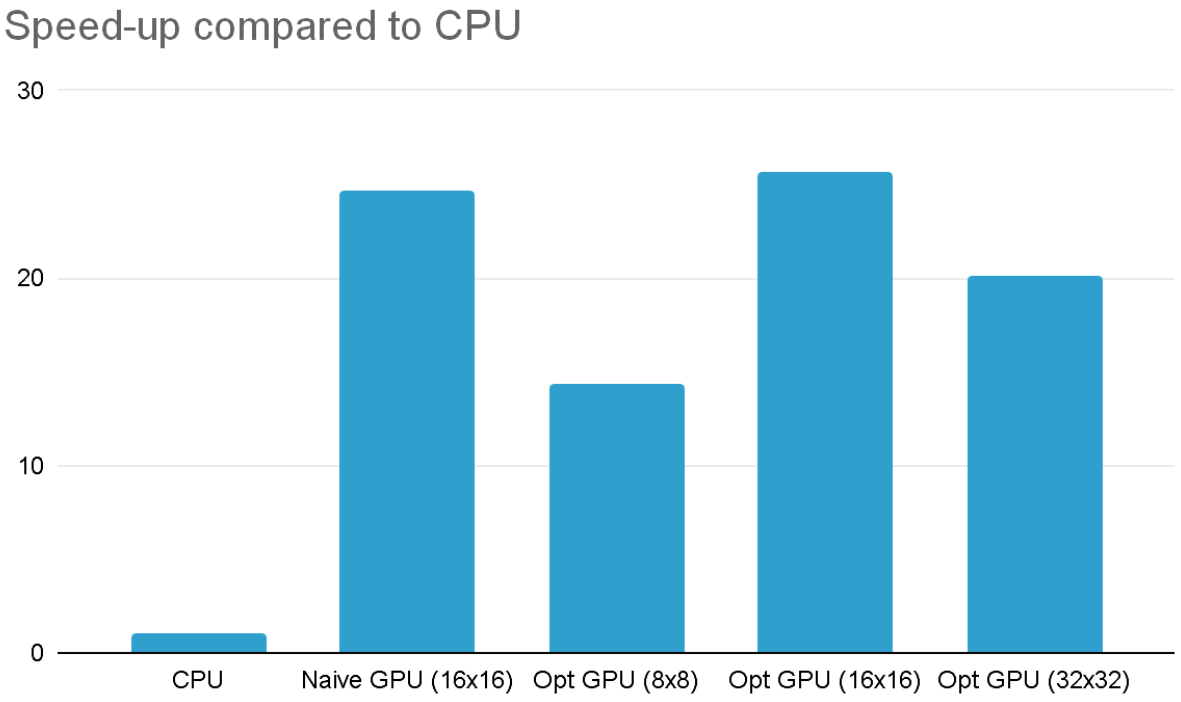
3. **Block Size 32x8**
   Time = **8.6238 ms**
   Speedup vs CPU = **20.112x**
   Speedup vs naive GPU = **0.814815x**
   This version also faster than CPU but not better than 16x16. Maybe because memory access not efficient with this block shape. Overhead could also be at play here since shape is irregular. Unfortunately due to device limits, i couldnt use 32x32 and was forced to use 32x8.

Device Limits

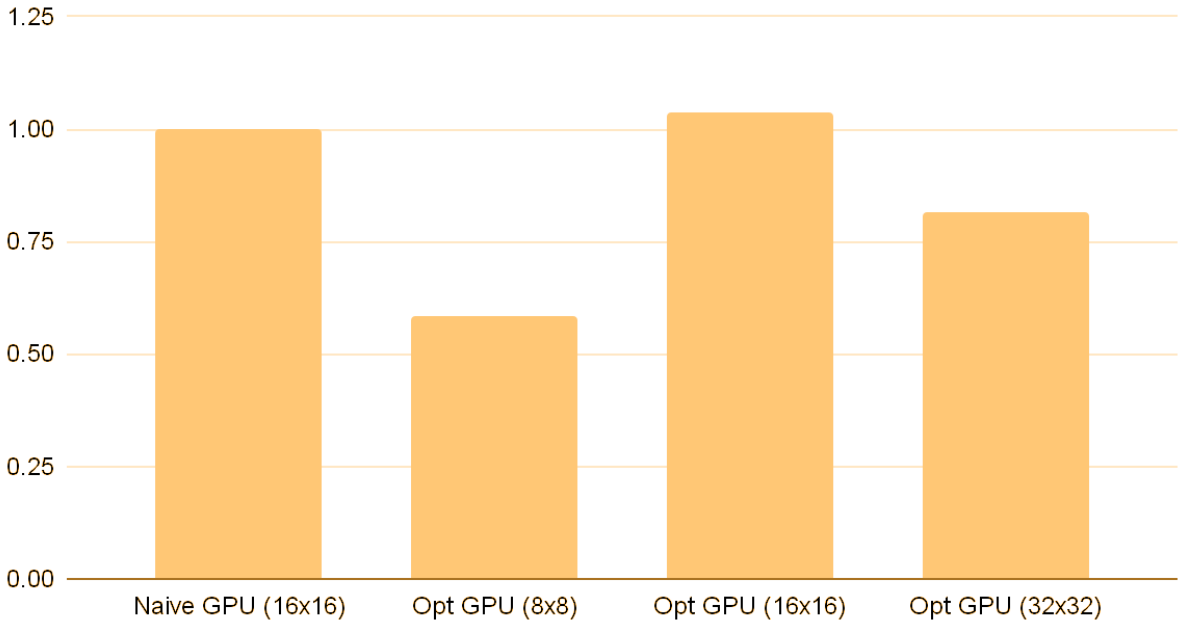
Device support max workgroup size of 256, and max number of items is [1024, 1024]. All of our kernel sizes are inside limit. Hence why i didnt test with 32x32 as it was giving alot of issues. So i did 32x8 which worked and showed improvements as well.

Performance Graphs:

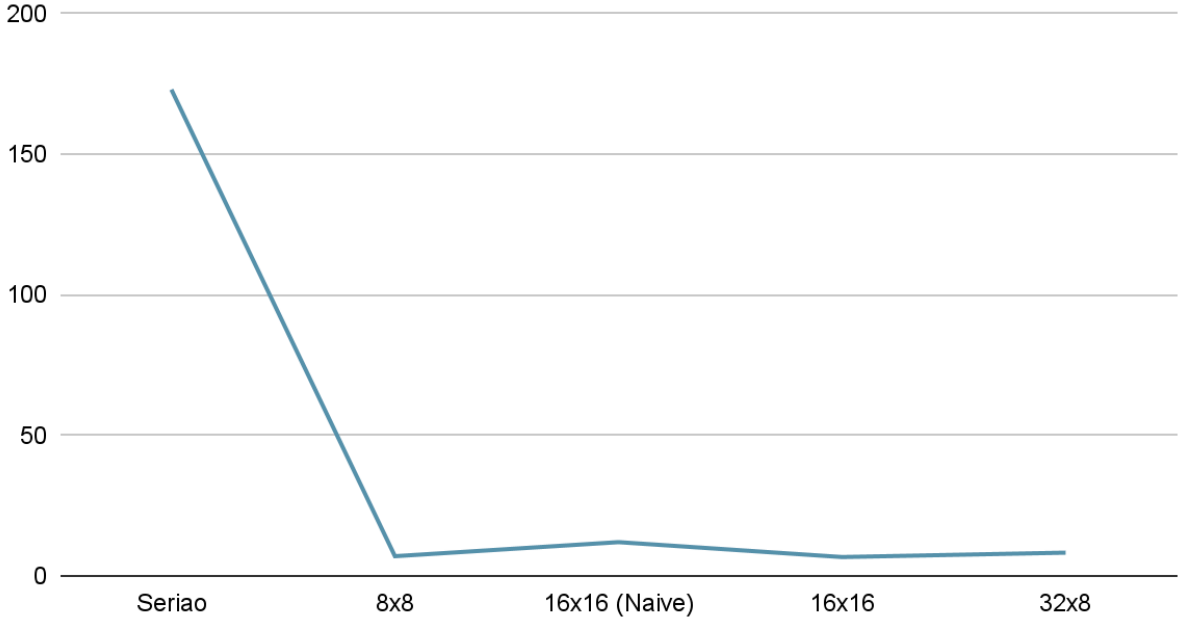

Execution Time in Milliseconds
200
150
100
50
0
CPU
Naive GPU (16x16)
Opt GPU (8x8)
Opt GPU (16x16)
Opt GPU (32x32)

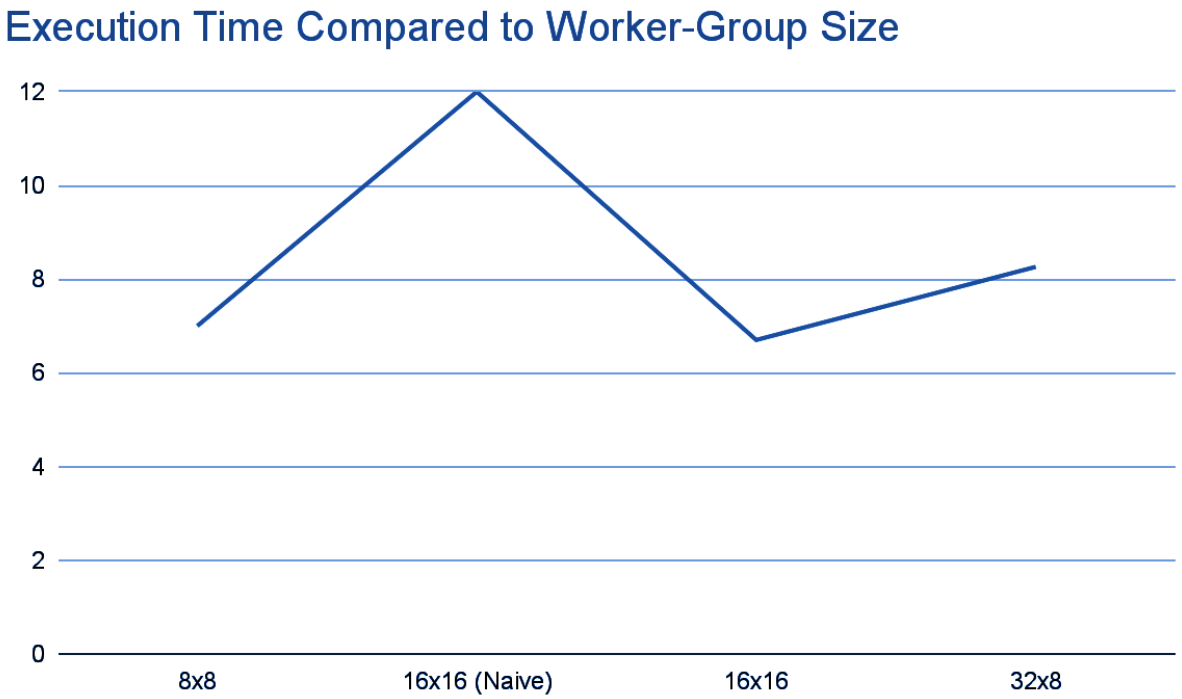

Speed-up compared to CPU
30
20
10
0
CPU
Naive GPU (16x16)
Opt GPU (8x8)
Opt GPU (16x16)
Opt GPU (32x32)

Speed-Up Compared to Naive-GPU



Excution Time Compared





Conclusion

From test we see GPU can do convolution much faster than CPU. Best result we get from optimized GPU with block size **16x16**, which is more than **25x faster** than CPU. Some block sizes not give good speedup, so choosing right configuration is important. All GPU versions give **correct output**, which mean optimization not break correctness.

GPU help very much in speeding up parallel task like convolution.