CMSC416: Assignment 4 Report

Krisnajit Rajeshkhanna

November 2024

1 Overview of Implementation

In this implementation of the convolution operation using CUDA, video frames are processed to apply image kernels such as blur, edge detection, sharpening, and identity transformations. The host (CPU) reads the input video file and extracts individual frames, which are stored in a 1D array in BGR (Blue, Green, Red) format for efficient memory access. To leverage the GPU's parallel processing capabilities, the host allocates memory on the device (GPU) for both the input and output frames and copies the frame data and convolution kernel to the device memory using `cudaMemcpy`. By transferring all necessary data to the GPU before computation, we minimize data movement overhead and take full advantage of the GPU's high-throughput architecture.

The core of the implementation is the CUDA kernel function `convolveGPU`. Each thread within this kernel is responsible for computing the convolution operation on one or more pixels of the video frame. Global thread indices are calculated using block and thread indices, and grid-stride loops are employed to ensure that all pixels are processed, even when the frame dimensions exceed the total number of available threads. By adjusting the block size via the `blockDimSize` variable, we can optimize thread utilization and manage the trade-off between parallelism and resource constraints.

Boundary conditions are handled by skipping border pixels where the convolution kernel would extend beyond the frame boundaries, thus preventing out-of-bounds memory access. The convolution is applied independently to each color channel—blue, green, and red—by accumulating the weighted sums of the neighboring pixels as defined by the kernel matrix. This approach maximizes parallelism and ensures efficient memory access patterns, as each thread accesses contiguous memory locations corresponding to the color channels. By fine-tuning the block size and utilizing grid-stride loops, the implementation achieves efficient utilization of the GPU's computational resources, leading to significant performance improvements over a CPU-only implementation.

2 Performance Analysis

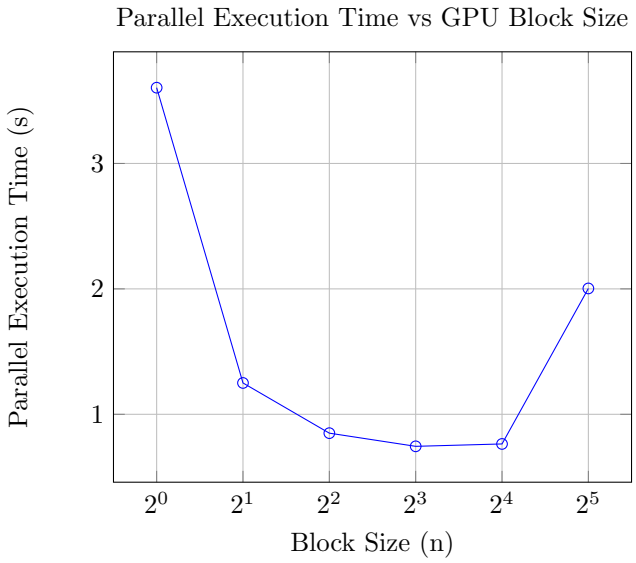

Figure 1: Parallel Times vs Block Size over 512x512 Grid

The graph illustrates the execution time of the GPU implementation as a function of the block size used during kernel launches. The block size varies from 1 to 32, and a fixed grid size of 512 x 512 is used. Several key observations can be made from the plot:

- **Decreasing Execution Time with Increasing Block Size (Up to a Point):** The execution time decreases significantly as the block size increases from 1 to 8. Specifically, the execution time drops from 3.60418 seconds at a block size of 1 to 0.744599 seconds at a block size of 8. This indicates that increasing the block size improves performance by allowing more threads per block, enhancing parallelism, and better utilizing the GPU's computational resources.

- **Optimal Performance at Block Size of 8:** The lowest execution time is observed at a block size of 8. This suggests that at this block size, there is an optimal balance between the number of threads per block and the GPU's ability to manage and schedule these threads efficiently.

- **Performance Degradation Beyond Optimal Block Size:** When the block size increases beyond 8, the execution time starts to increase again. At a block size of 16, the execution time slightly increases to 0.763605 seconds, and at a block size of 32, it rises significantly to 2.00348 seconds. This performance degradation can be attributed to resource contention

and decreased occupancy. Larger block sizes consume more shared memory and registers per block, which can limit the number of active blocks per Streaming Multiprocessor (SM) and reduce overall parallelism.

- **Impact of Fixed Grid Size:** The grid size is kept constant at 512×512 across all block sizes. As the block size increases, the total number of threads launched changes, affecting performance. With larger block sizes and a fixed grid size, the total number of threads can exceed the optimal amount, leading to inefficient utilization of the GPU's resources.

The results align with the expectation that increasing the block size will improve performance up to a certain point, after which performance may degrade due to resource limitations. It was anticipated that there would be an optimal block size that balances thread utilization and resource availability. The observed optimal performance at a block size of 8 confirms this expectation. The significant increase in execution time at a block size of 32 was more pronounced than expected. This suggests that resource contention and reduced occupancy have a substantial impact at larger block sizes.

3 Conlusion

The performance analysis confirms that there is an optimal block size that balances thread utilization and resource availability. In this implementation, the optimal block size is 8 when using a fixed grid size of 512×512. However, to achieve consistent and optimal performance across different block sizes, it is essential to adjust the grid size according to the block size and image dimensions.

By recalculating the grid size based on the block size, we can ensure efficient utilization of the GPU's computational resources, prevent resource contention, and maintain high occupancy. This adjustment would likely result in improved performance at larger block sizes and provide more consistent execution times across varying configurations.

Overall, the results highlight the importance of considering both block size and grid size when optimizing GPU kernels for performance. Proper configuration ensures that the GPU's architecture is leveraged effectively, leading to faster execution and better resource utilization.