Assignment 1: GPU Intuition & Compute Foundations

Sruthi Manepalli

Chosen Workload

Direct 2D Sliding Window Convolution (Spatial Convolution with Fixed Kernel)

1 Task 1: Identify and Analyze a GPU-Accelerable Workload

1.1 1.1 Operation Breakdown

Direct 2D sliding window convolution is a fundamental operation in image processing and signal processing. It computes each output pixel as a weighted sum of a local neighborhood of input pixels defined by a fixed-size kernel. This operation is widely used in image filtering, edge detection, smoothing, and as the core computational primitive in convolutional neural networks.

Given a two-dimensional input image I of dimensions $H \times W$ and a convolution kernel K of dimensions $(2r+1) \times (2r+1)$, the output image O is computed by sliding the kernel across the image and performing element-wise multiplication followed by accumulation.

Mathematical Formulation

$$O(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} I(x+i, y+j) \cdot K(i, j) \tag{1}$$

where (x, y) denotes the output pixel location and r is the kernel radius. Boundary conditions are handled using zero-padding or clamping.

Pseudocode

```
for x in range(H):
    for y in range(W):
        sum = 0
        for i in range(-r, r+1):
            for j in range(-r, r+1):
                sum += I[x+i][y+j] * K[i][j]
        O[x][y] = sum
```

Input Sizes and Parameters

- Input image: $H \times W$ (e.g., 512×512 grayscale image)

- Kernel size: fixed (e.g., 3×3 or 5×5)
- Output image: $H \times W$

Independent Work Units

Each output pixel $O(x, y)$ is computed independently of all other output pixels. There are no data dependencies between output elements, making the workload parallel at the pixel level.

1.2 1.2 Compute vs Memory Analysis

Compute-Bound vs Memory-Bound

The naive implementation of direct convolution is primarily **memory-bound**. Each output pixel requires loading multiple input pixels from memory, and neighboring output pixels reuse many of the same input values. Without caching or tiling, the same input pixels are repeatedly fetched from global memory.

For small kernel sizes, the number of arithmetic operations per pixel is limited, while memory access latency dominates the execution time.

Arithmetic Intensity

For a 3×3 kernel:

- Arithmetic operations: 9 multiplications and 8 additions
- Memory accesses: 9 input reads and 1 output write

This results in low to moderate arithmetic intensity, indicating that performance is constrained by memory bandwidth rather than compute throughput.

Data Reuse Opportunities

Significant data reuse exists in convolution:

- Each input pixel contributes to multiple neighboring output pixels
- A single pixel may be reused up to $(2r + 1)^2$ times

This reuse makes the workload highly suitable for shared memory optimization on GPUs, where tiles of the image can be loaded once and reused by multiple threads.

Dependencies

There are no inter-output dependencies. The inner loops over kernel indices are local to each thread, and no global synchronization is required beyond block-level synchronization.

1.3 1.3 Expected Behavior on a GPU

CUDA Thread Mapping

A two-dimensional CUDA grid is used, where each thread computes one output pixel $O(x, y)$. Thread indices map directly to pixel coordinates. This mapping allows thousands of threads to execute concurrently.

Scalability

The workload scales efficiently on GPUs due to:

- High degree of data parallelism
- Uniform work per thread
- Regular memory access patterns for interior pixels

Potential Challenges in GPU Implementation

Despite the high degree of data parallelism in direct 2D sliding window convolution, several GPU-specific challenges can limit performance if not carefully handled. These challenges arise primarily from memory behavior, control flow divergence, and hardware resource constraints.

1. **Global Memory Bandwidth Limitation**

   Global memory on a GPU is implemented using off-chip DRAM and has significantly higher latency compared to on-chip memories such as registers and shared memory. In a naive GPU implementation of convolution, each thread independently loads all the input pixels required to compute its output pixel directly from global memory.

   Since adjacent output pixels share many common input pixels, neighboring threads repeatedly load the same input data from global memory. For example, in a 3×3 convolution, each output pixel requires nine input pixel reads. As a result, the same input pixel may be fetched multiple times by different threads, leading to excessive and redundant global memory accesses.

   Because the number of arithmetic operations per output pixel is relatively small for small kernel sizes, the GPU often becomes memory-bound. In such cases, performance is limited by memory bandwidth rather than computational throughput, with arithmetic units remaining idle while waiting for data to be fetched from global memory.

2. **Boundary Handling and Warp Divergence**

   GPUs execute threads in groups known as warps, typically consisting of 32 threads that follow a single instruction stream. Warp divergence occurs when threads within the same warp follow different execution paths due to conditional branches, causing the GPU to serialize the execution of each path.

   In convolution, boundary handling introduces conditional logic for pixels near the image edges, where the convolution kernel partially extends beyond the valid image region. To

prevent invalid memory accesses, conditional checks are used to determine whether neighboring pixel indices fall within image bounds, often applying zero-padding or clamping for out-of-range accesses.

When a warp contains a mix of threads corresponding to interior pixels and boundary pixels, some threads satisfy the boundary condition while others do not. This results in warp divergence, forcing the GPU to execute both branches sequentially. During each branch execution, threads that do not satisfy the condition remain inactive, reducing effective parallelism and lowering overall throughput. Although divergence is confined to boundary regions, its impact becomes more noticeable for small images or small block sizes, where a larger fraction of threads lie near image edges.

3. **Shared Memory Limits and Tile Size Constraints**

   Shared memory is a low-latency, on-chip memory shared by all threads within a block. To reduce redundant global memory accesses, convolution implementations often divide the input image into smaller regions known as tiles, which are cooperatively loaded into shared memory by threads within a block.

   For a block of size $B_x \times B_y$ and a convolution kernel of radius r, the required shared memory tile dimensions become $(B_x + 2r) \times (B_y + 2r)$, including the halo region needed for convolution near tile boundaries. As either the block size or kernel size increases, the shared memory footprint of each tile grows correspondingly.

   However, the amount of shared memory available per block is limited by the GPU hardware, typically to tens of kilobytes. If the tile size exceeds this limit, the kernel may fail to launch or experience reduced occupancy, meaning fewer blocks can reside on a streaming multiprocessor simultaneously. Even when the tile fits within shared memory, large tile sizes can reduce occupancy by consuming a significant portion of the shared memory budget. Consequently, selecting appropriate block dimensions and tile sizes requires a careful balance between maximizing data reuse and maintaining sufficient parallelism.

Optimization Opportunity

By loading image tiles (including halo regions) into shared memory, redundant global memory accesses are reduced. This enables efficient reuse of input data and significantly improves performance.

2 Task 2: CUDA Execution Model Mapping Diagram

The CUDA execution hierarchy for this workload is as follows:

- **Grid:** Covers the entire output image
- **Blocks:** Each block processes a tile of output pixels
- **Warps:** Groups of 32 threads executing in lockstep
- **Threads:** Each thread computes one output pixel

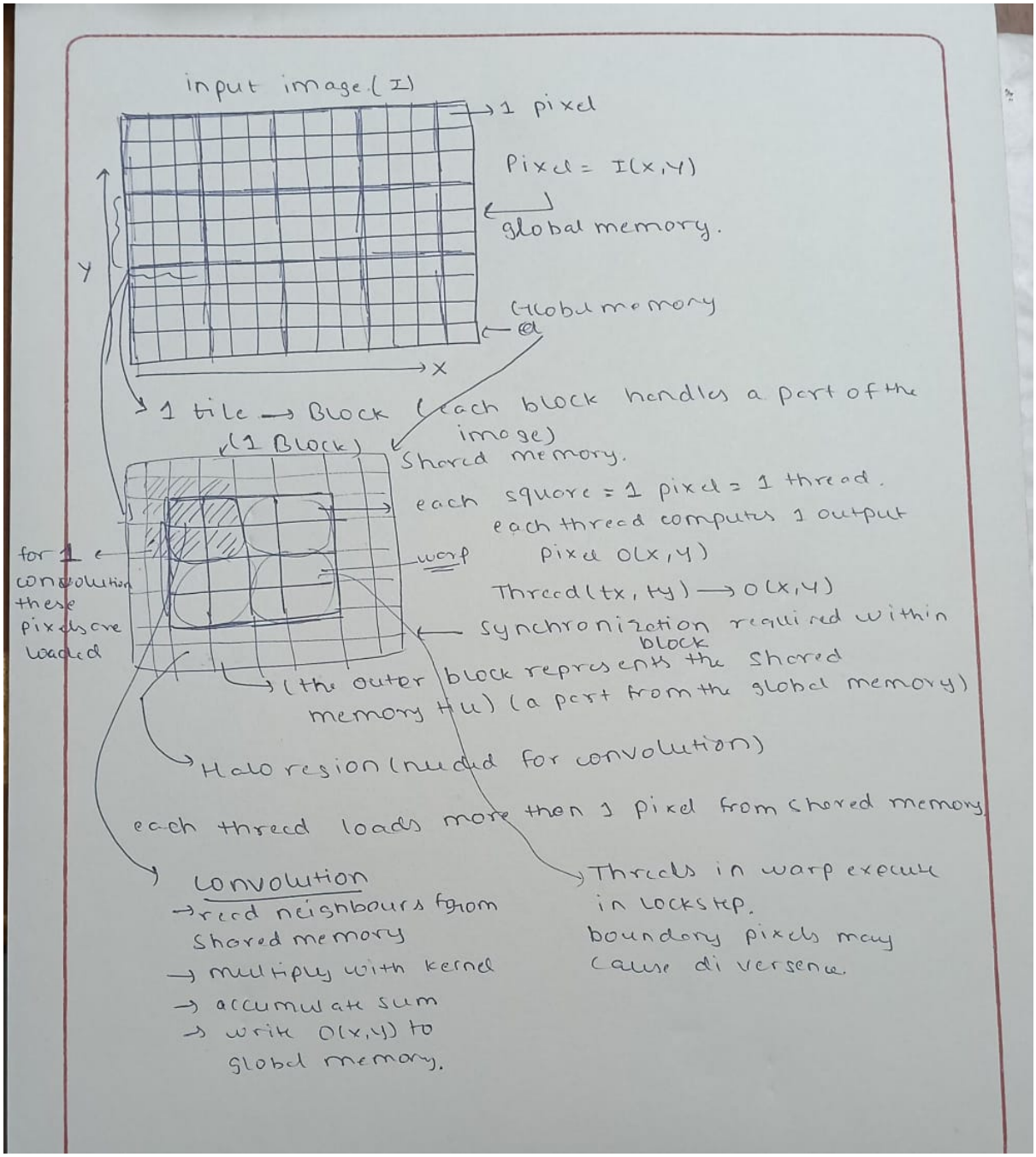

Figure 1:

Synchronization

Block-level synchronization is required after loading data into shared memory. No synchronization is required across blocks.

Memory Bottlenecks

Global memory bandwidth is the primary bottleneck in naive implementations, while shared memory size constraints limit tile dimensions in optimized implementations.

Conclusion

Direct 2D sliding window convolution is a well-defined, computation-heavy workload that clearly demonstrates the advantages of GPU architectures over CPUs. It exhibits high data parallelism, significant memory reuse opportunities, and well-understood optimization strategies using shared memory. This makes it an ideal workload for analyzing CPU versus GPU performance and understanding GPU execution models.