# WIDS 25-26: GPU Programming

Week 1 Report

Anshika Jain

December 12, 2025

## 1 Introduction and Project Goal

I was thinking of using GPU eventually for the implemention of a **Tiny U-Net** architecture for medical image segmentation. U-Net models are widely used in medical imaging tasks such as organ and tumor segmentation from CT and MRI scans.

Before implementing the full model, the first step to understand the core computational building block used throughout U-Net: **convolution**. In this report, we analyze a **2D convolution workload** from both a mathematical and GPU execution perspective. The 2D case is chosen for clarity and ease of understanding; the same concepts directly extend to 3D convolutions used in volumetric medical data.

## 2 Mathematical Formulation of 2D Convolution

Let the input image be

$$X \in \mathbb{R}^{B \times C_{in} \times H \times W},$$

where $B$ is the batch size, $C_{in}$ is the number of input channels, and $H \times W$ is the image resolution.

The convolution kernel is

$$W \in \mathbb{R}^{C_{out} \times C_{in} \times K \times K},$$

where $C_{out}$ is the number of output channels and $K$ is the kernel size.

The output feature map is

$$Y \in \mathbb{R}^{B \times C_{out} \times H \times W}.$$

For a single output pixel, the convolution operation is defined as:

$$Y[b, c_o, y, x] = \sum_{c_i=0}^{C_{in}-1} \sum_{k_y=0}^{K-1} \sum_{k_x=0}^{K-1} W[c_o, c_i, k_y, k_x] \cdot X[b, c_i, y + k_y - r, x + k_x - r],$$

where $r = \lfloor K/2 \rfloor$ represents the padding radius.

## 3 Example Configuration

To make the analysis concrete, the following configuration is considered:

- Batch size: $B = 1$
- Input channels: $C_{in} = 1$
- Output channels: $C_{out} = 32$

- Image resolution: $256 \times 256$
- Kernel size: $K = 3$

This configuration is representative of early layers in a Tiny U-Net architecture.

## 4 Analysis

The total number of output pixels is:

$$1 \times 32 \times 256 \times 256 = 2{,}097{,}152.$$

Each output pixel requires a $3 \times 3$ convolution:

- 9 multiplications
- 9 additions

Thus, the total number of floating-point operations (FLOPs) is:

$$2{,}097{,}152 \times 18 \approx 3.77 \times 10^7 \text{ FLOPs}.$$

Although manageable in 2D, this cost increases dramatically in 3D convolution, motivating GPU acceleration.

## 5 Compute vs Memory Analysis

### 5.1 Compute-Bound vs Memory-Bound

The 2D convolution operation is neither purely compute-bound nor purely memory-bound. Each output pixel requires multiple floating-point operations as well as several memory accesses to the input feature map.

In a naive implementation where each thread loads all required input values directly from global memory, the operation tends to be memory-bound. However, with optimizations such as shared memory tiling, input values are reused by multiple threads, increasing computation per memory access. As a result, the operation moves closer to being compute-bound on modern GPUs.

### 5.2 Arithmetic Intensity

Arithmetic intensity is defined as:

$$\text{Arithmetic Intensity} = \frac{\text{Number of Floating Point Operations}}{\text{Number of Bytes Accessed}}.$$

For a typical $3 \times 3$ 2D convolution, each output pixel performs approximately 18 floating-point operations. Since input pixels are reused across neighboring output pixels, the arithmetic intensity is moderate. This places convolution between simple memory-bound operations and highly compute-bound operations such as large matrix multiplications.

### 5.3 Reuse Opportunities

There are significant data reuse opportunities in 2D convolution. Each input pixel contributes to multiple neighboring output pixels. By loading a tile of the input feature map into shared memory, threads within a block can reuse input values efficiently.

### 5.4 Dependencies and Parallelism

The computation of output pixels is largely independent, enabling a high degree of parallelism. Limited dependencies exist in the form of:

- Synchronization after loading shared memory tiles
- Boundary condition checks
- Sequential accumulation of convolution results within a thread

These dependencies are local and do not significantly limit overall parallel execution.

## 6 Memory Analysis

Using 32-bit floating point numbers (4 bytes):

- Input memory:
$$1 \times 1 \times 256 \times 256 \times 4 \approx 0.25 \text{ MB}$$
- Output memory:
$$32 \times 256 \times 256 \times 4 \approx 8 \text{ MB}$$
- Weights:
$$32 \times 1 \times 3 \times 3 \times 4 \approx 1.1 \text{ KB}$$

The total memory footprint is approximately 8.25 MB.

## 7 Expected Behavior on a GPU

### 7.1 Mapping CUDA Threads to the Iteration Space

In a 2D convolution operation, each output pixel can be computed independently. A natural CUDA mapping assigns one thread to compute one output pixel. Threads are organized into blocks, where each block processes a tile of the output feature map, and all blocks together form the grid.

This mapping exposes a large amount of data-level parallelism and is well suited to GPU execution.

### 7.2 Scalability

The convolution workload scales well to thousands of threads. Medical images typically contain millions of output pixels, allowing the GPU to launch a large number of threads and blocks. This enables high occupancy and efficient utilization of the GPU's streaming multiprocessors.

### 7.3 Challenges

Several challenges may arise during GPU execution:

- **Warp divergence:** Conditional statements, such as boundary checks, may cause threads within the same warp to follow different execution paths, reducing efficiency.
- **Irregular memory access:** Irregular global memory accesses can lead to inefficient memory bandwidth usage. Proper memory layout and shared memory tiling are required to address this.

- **Small batch sizes:** Medical imaging applications often use small batch sizes, which limits parallelism in the batch dimension. Parallelism must therefore be extracted from spatial dimensions and output channels.

## 8 Shared Memory Tiling

Global memory access is slow compared to on-chip shared memory. To improve performance:

1. Each block loads a tile of the input image into shared memory
2. Threads synchronize using `__syncthreads()`
3. Convolution is performed using shared memory

This technique significantly reduces memory bandwidth usage and improves performance. Not sure about this will have to read about it.

## 9 CPU Baseline

Not submitting in Week 1!!

A CPU baseline can be measured using PyTorch's `conv2d` function and `time.perf_counter()`. This baseline demonstrates the performance gap between CPUs and GPUs for convolution-heavy workloads.

## 10 Task 2: CUDA Execution Model Mapping Diagram

### 10.1 CUDA Execution Hierarchy

The CUDA execution model follows a hierarchical structure consisting of four levels:

$$\textbf{Grid} \rightarrow \textbf{Blocks} \rightarrow \textbf{Warps} \rightarrow \textbf{Threads}.$$

This hierarchy allows a large number of threads to execute in parallel on the GPU.

### 10.2 Mapping the Workload to the CUDA Hierarchy

The chosen workload is a 2D convolution operation used in medical image segmentation.

#### 10.2.1 Threads

Each CUDA thread is responsible for computing one output pixel of the convolution. The thread performs all multiply–add operations required for that pixel using values from the input feature map and the convolution kernel.

#### 10.2.2 Warps

Threads are grouped into warps of 32 threads. All threads within a warp execute the same instruction simultaneously. For convolution, threads in a warp typically compute neighboring output pixels, which results in coalesced memory access and efficient execution.

#### 10.2.3 Blocks

A block consists of multiple warps and is assigned a rectangular tile of the output feature map (for example, a $16 \times 16$ region). Threads within a block cooperate to load the required input data into shared memory.

#### 10.2.4 Grid

The grid contains all blocks required to cover the entire output image. Each block operates independently on a different tile of the output.

### 10.3 Synchronization

Synchronization is required within each block after loading input data into shared memory. This ensures that all threads have access to the required data before computation begins. Synchronization is performed using the `__syncthreads()` instruction.

### 10.4 Potential Memory Bottlenecks

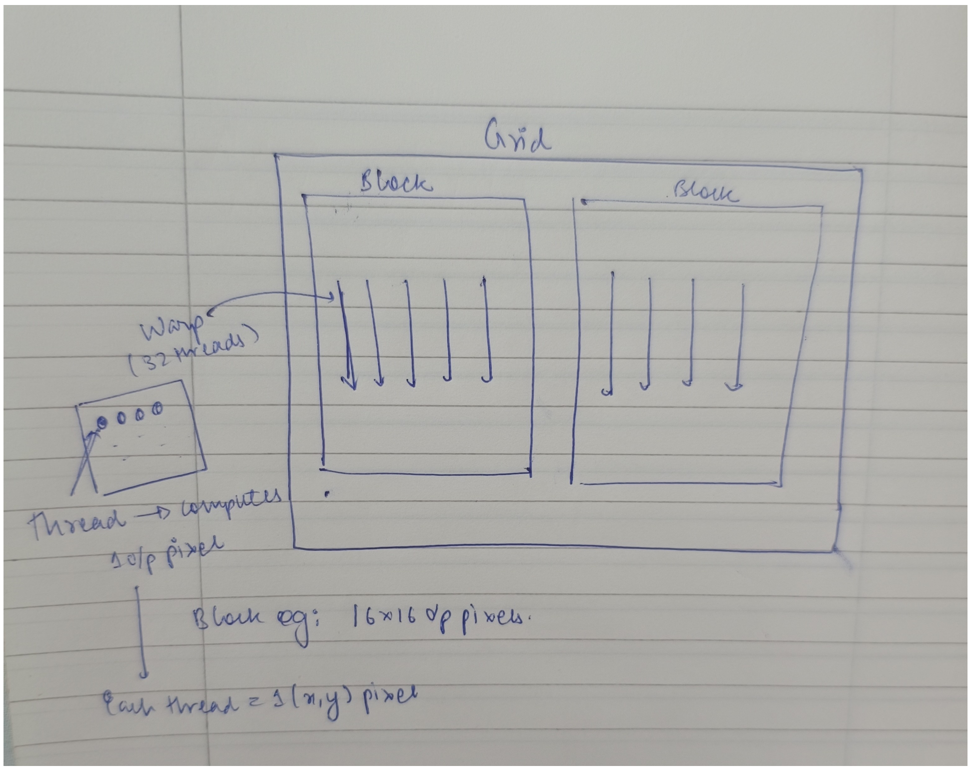
Without optimization, each thread may repeatedly access global memory to load input values, leading to high memory bandwidth usage. Non-coalesced memory accesses and redundant reads of overlapping input regions can further reduce performance.

### 10.5 Shared Memory Optimization

To reduce global memory traffic, shared memory is used at the block level. Each block loads a tile of the input feature map, including halo regions, into shared memory. Threads within the block reuse this data for convolution, significantly improving memory efficiency and overall performance.

### 10.6 Diagram Description

Figure 1 illustrates the CUDA execution hierarchy and its mapping to the convolution workload. The diagram shows how the grid is divided into blocks, blocks into warps, and warps into threads.



Grid
Block
Block
Warp (32 threads)
Thread → computes 1 o/p pixel
Block eg: 16×16 o/p pixels.
Each thread = 1 (x,y) pixel

Figure 1: CUDA execution model mapping for a 2D convolution workload.

## 11 Conclusion

This report analyzed a 2D convolution workload relevant to medical imaging. By understanding the mathematical formulation and GPU execution model, a foundation is established for implementing a Tiny U-Net. The same principles scale directly to 3D convolution, which is widely used in real-world medical image segmentation tasks.

This is just a plan and would like to change accordingly in the upcoming weeks as I don't have much knowledge about this now.