CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

4.Basics: CUDA basics, scheduling, and matmul optimization

Lecturer: Hao Zhang

Scribes: Yiting Dai, Kaiqin Kong, Tianyu Xia, Yucheng Mao, Qingran Wu, Xinkai Zou, Shenzhuo Zhang, Lingqi Zeng

1 Overview

This lecture develops the mental model needed to reason about GPU programs. The story starts from the hardware constraint that makes GPUs fast: SIMD-style execution, where many lanes advance in lockstep. It then introduces CUDA's execution and memory model, shows a first optimization on a sliding-window kernel, and finally uses matrix multiplication as a case study in progressively more sophisticated tiling strategies. The recurring theme is that good GPU programs are not only parallel; they also match the memory hierarchy and scheduling rules of the hardware.

2 SIMD Constraints and Warp Divergence

Before writing CUDA code, it helps to understand the execution model underneath it. Threads are grouped into warps, and lanes inside the same warp share instruction fetch and decode. As a result, they are intended to execute the same instruction stream at the same time. This is why the lecture emphasizes the phrase "convert your brain to be SIMD": one must design work so that nearby threads behave similarly.

Control flow creates tension with SIMD hardware. Different threads may evaluate a branch condition differently, but a warp cannot literally split into fully independent processors. Instead, the hardware executes the different paths with masking or predication. Threads that are not active on a path still occupy the warp while that path executes, but their writes are suppressed. The performance consequence is important: a divergent branch often costs close to the sum of both paths rather than the cost of only the path taken by one thread.

This leads to the distinction between *coherent* and *divergent* execution.

- Coherent execution means the same instructions apply to all data elements in the warp, so the hardware delivers close to peak throughput.
- Divergent execution means threads in the same warp want different control-flow paths, forcing the hardware to serialize work that would ideally run together.

The lecture notes that divergence should be minimized in CUDA programs whenever possible. This observation becomes especially relevant in machine learning settings such as masking in language models or sliding-window attention, where branch behavior can easily depend on position.

3 CUDA Execution and Memory Model

3.1 Host and Device Address Spaces

CUDA programs run across two distinct processors and two distinct address spaces:

- **Host memory**: CPU RAM, managed by the operating system.
- **Device memory**: GPU memory, typically high-bandwidth memory (HBM).

These spaces are separate. CPU code cannot directly dereference device pointers, and GPU kernels cannot directly access ordinary host pointers. Therefore, moving data to the GPU requires explicit runtime calls such as `cudaMemcpy`. A common beginner mistake is to pass a host pointer into a kernel and expect it to behave like unified shared memory; in the standard CUDA model that is incorrect.

3.2 Pinned Memory

The lecture also highlights *pinned* host memory. Pinned memory is a region of host memory that the operating system will not page out. Because it is locked in physical memory, it supports faster and more predictable transfers between CPU and GPU, and some CUDA APIs require it. Conceptually, it serves as a more efficient staging area for host-device communication.

3.3 Memory Hierarchy from a Kernel's Perspective

Within the GPU, CUDA exposes a hierarchy of memories that trade off speed, capacity, and sharing scope.

Scope	Memory	Who can access it
Thread	Registers	Private to one thread; fastest storage
Block	Shared memory (SRAM)	Read/write by all threads in one block
Device	Global memory (HBM)	Read/write by all threads on the device

The point of this hierarchy is not complexity for its own sake. It gives programmers several places to store data depending on how much sharing is needed and how expensive a memory access can be. Most of the optimizations later in the lecture are really about moving data down this hierarchy: from global memory into shared memory, and then from shared memory into registers.

4 First CUDA Example: Window Average

The running example is a simple 1D sliding-window average:

```
for (int i = 0; i < len(input) - 2; i++) {
    output[i] = (input[i] + input[i+1] + input[i+2]) / 3.0;
}
```

Each output element depends on three consecutive inputs, but different output positions are independent of one another. That independence is the parallelizable unit: one CUDA thread can compute one output element.

4.1 GPU v1: One Thread per Output

The first kernel maps each reduction to a thread. If the thread's global index is `index`, it loads `input[index]`, `input[index+1]`, and `input[index+2]`, sums them, and writes one output. This mapping is correct and naturally parallel. However, it reveals the first major GPU optimization lesson: correctness is not the same as efficiency.

Neighboring threads access highly overlapping input regions. Thread t loads $(t, t+1, t+2)$, while thread $t+1$ loads $(t+1, t+2, t+3)$. Two of the three values are duplicated across adjacent threads. When these repeated loads come from global memory, the kernel performs unnecessary HBM traffic.

4.2 GPU v2: Shared-Memory Reuse

The second version lets threads in the same block cooperate. Rather than having every thread fetch all three values independently from global memory, the block first loads a contiguous support region into shared memory:

- with 128 threads per block, the block needs 128 primary elements,
- plus two halo elements for the length-3 window,
- so the shared buffer has size $128 + 2 = 130$.

After the cooperative load, the block calls `__syncthreads()` so that every thread waits until the shared buffer is fully populated. Only then do threads read their 3-element windows from shared memory.

This optimization is easy to quantify. In the naive version, each of the 128 threads reads 3 input values, for a total of 384 global reads per block. In the shared-memory version, the whole block performs only 130 global reads. The kernel still computes the same outputs, but it removes a large amount of redundant HBM traffic by exploiting block-level reuse.

4.3 Synchronization

The example introduces two distinct forms of synchronization:

- `__syncthreads()` is a *block-local barrier*. All threads in the same block must arrive before any of them may proceed.
- Host-device synchronization is different. A kernel launch is generally asynchronous with respect to the CPU, so the host program can continue immediately after launching work. If the CPU needs the result, it must explicitly synchronize or perform a transfer that implies synchronization.

This distinction matters because CUDA programming always involves two levels of orchestration: coordination among threads inside a kernel and coordination between the CPU runtime and the GPU device.

5 How CUDA Schedules Thread Blocks

5.1 The Key Assumption: Blocks Are Independent

The lecture next turns from a single kernel to the scheduler that runs it. A CUDA kernel may launch a large static grid of thread blocks, but GPUs have only a limited number of streaming multiprocessors (SMs).

Different GPUs have very different SM counts, so CUDA cannot assume that all blocks run simultaneously. Instead, the hardware scheduler maps blocks onto SMs dynamically.

This dynamic scheduling relies on one crucial design rule: **thread blocks must be independent**. Because there are no cross-block dependencies within a single kernel launch, the hardware is free to execute blocks in any order and on any SM.

5.2 A Scheduling Walkthrough

The lecture uses the window-average kernel as a concrete example. Suppose a launch creates 1024 blocks, each with 128 threads and 520 bytes of shared memory usage. Now imagine a small GPU with only two SMs and limited shared-memory capacity per SM.

The scheduler proceeds conceptually as follows:

1. The host launches the kernel and passes the kernel configuration to the GPU.
2. The scheduler maps block 0 onto SM 0 and reserves the execution contexts and shared memory required by that block.
3. It continues mapping additional ready blocks onto any SM that still has sufficient resources.
4. Once all currently available resources are occupied, remaining blocks wait in a queue.
5. When a resident block finishes, its resources are released and the scheduler assigns the next waiting block to that SM.

The important point is that the same program works on a GPU with 2 SMs, 16 SMs, or over 100 SMs. More SMs increase parallel throughput, but the programming abstraction stays unchanged. This is why oversubscription is desirable: programmers typically launch far more blocks than can run concurrently so the device always has work ready when resources free up.

5.3 Occupancy and Resource Limits

Not every block can be placed on an SM merely because there are threads remaining. A block consumes several resources, including:

- thread slots,
- shared memory,
- registers.

These constraints determine how many blocks can be resident on an SM at once, which in turn affects occupancy and latency hiding. The lecture's simple example illustrates that even small shared-memory allocations can cap the number of simultaneous blocks if the per-SM memory budget is tight.

5.4 Hardware Evolution

To build intuition, the lecture compares a GTX 980 (2014) with an H100 (2022). The interesting observation is that the basic per-SM execution model did not change dramatically: threads per SM and CUDA cores per SM are in the same ballpark, while shared memory per SM increased moderately. The dramatic throughput growth instead came largely from two sources:

- many more SMs per chip,

- specialized tensor cores that accelerate matrix-heavy workloads.

This reinforces a systems lesson: scaling up total throughput is often achieved by replicating moderately sized execution units and adding domain-specific accelerators, not by making one unit arbitrarily large.

5.5 A Brief Note on Groq

The lecture briefly revisits Groq to contrast it with NVIDIA GPUs. The memorable takeaway is that Groq dedicates an unusually large amount of on-chip SRAM and avoids HBM in the way an ordinary GPU uses it. That design can make decoding extremely fast because it skips expensive off-chip memory traffic, but it imposes a strict capacity budget and pushes more burden onto the programmer or compiler to fit computations into available SRAM.

6 Case Study: Matrix Multiplication on GPU

The final part of the lecture develops GPU optimization as a sequence of matmul kernels. For square matrices $A, B, C \in \mathbb{R}^{N \times N}$,

$$C = AB, \qquad C_{xy} = \sum_{k=0}^{N-1} A_{xk} B_{ky}.$$

The lecture frames CUDA design as three recurring steps:

1. identify work that can run in parallel,
2. partition both work and data,
3. manage data access, communication, and synchronization.

A good design must also create enough tasks to keep the GPU busy, avoid stragglers, and minimize expensive traffic across memory hierarchies.

6.1 GPU Matmul v1: One Thread per Output Element

The most direct strategy is to assign one thread to compute one output scalar C_{xy}. A 2D grid covers the output matrix, and each thread performs a dot product between one row of A and one column of B.

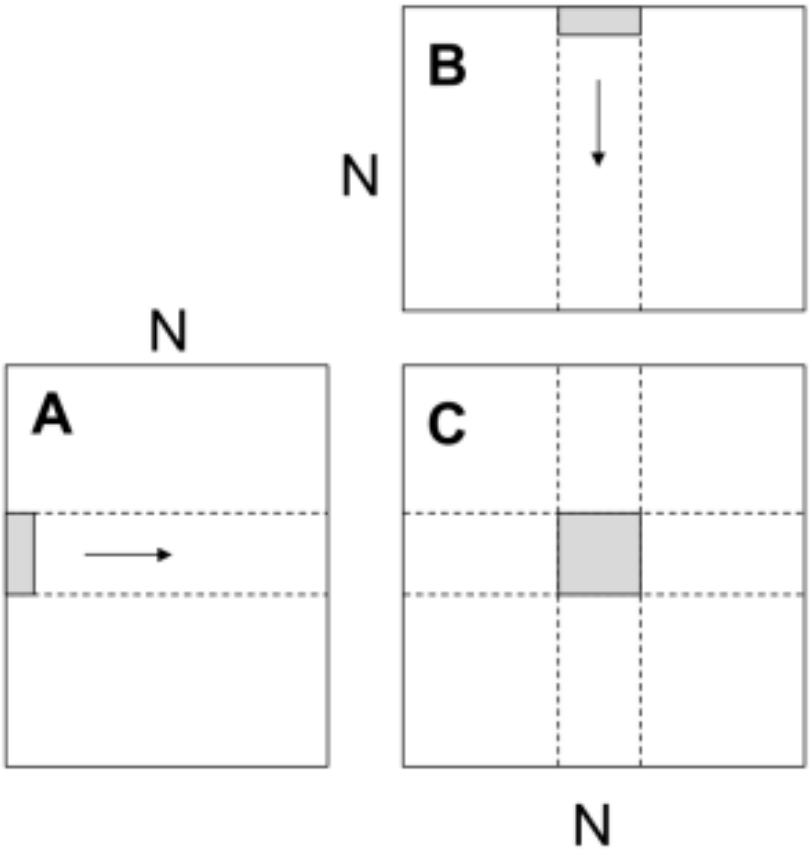

Figure 1: Matmul v1 assigns one output element to each thread.

This design is simple and perfectly parallel, but it is memory hungry:

$$\text{global reads per thread} = N + N = 2N,$$

$$\text{threads} = N^2, \qquad \text{total global memory access} = 2N^3.$$

The problem is that neighboring threads redundantly fetch overlapping rows and columns. Each thread keeps almost no local state, but the device pays heavily in global memory traffic.

6.2 GPU Matmul v1.5: Register Tiling

The next idea is to make each thread do more work. Instead of computing a single scalar, one thread computes a small $V \times V$ output tile and keeps the partial results in registers.

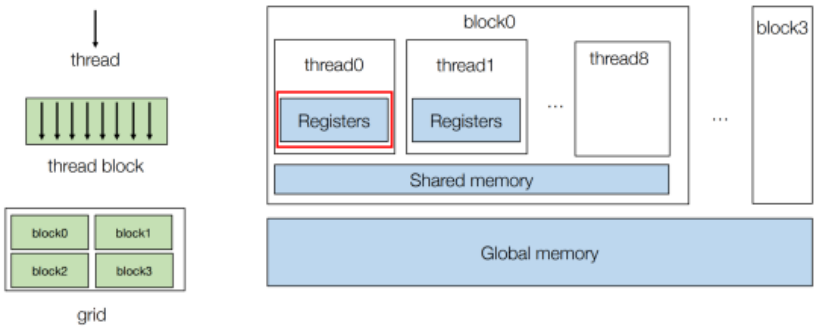

Figure 2: Register tiling increases reuse inside one thread.

This improves arithmetic intensity because one thread can reuse values while accumulating several outputs. In the lecture's accounting:

$$\text{global reads per thread} = NV + NV^2,$$

$$\text{threads} = \frac{N^2}{V^2}, \qquad \text{total global memory access} = \frac{N^3}{V} + N^3.$$

This is better than v1, but still not ideal. The thread now has more register reuse, yet it still performs a relatively inefficient pattern of loading rows of A and columns of B.

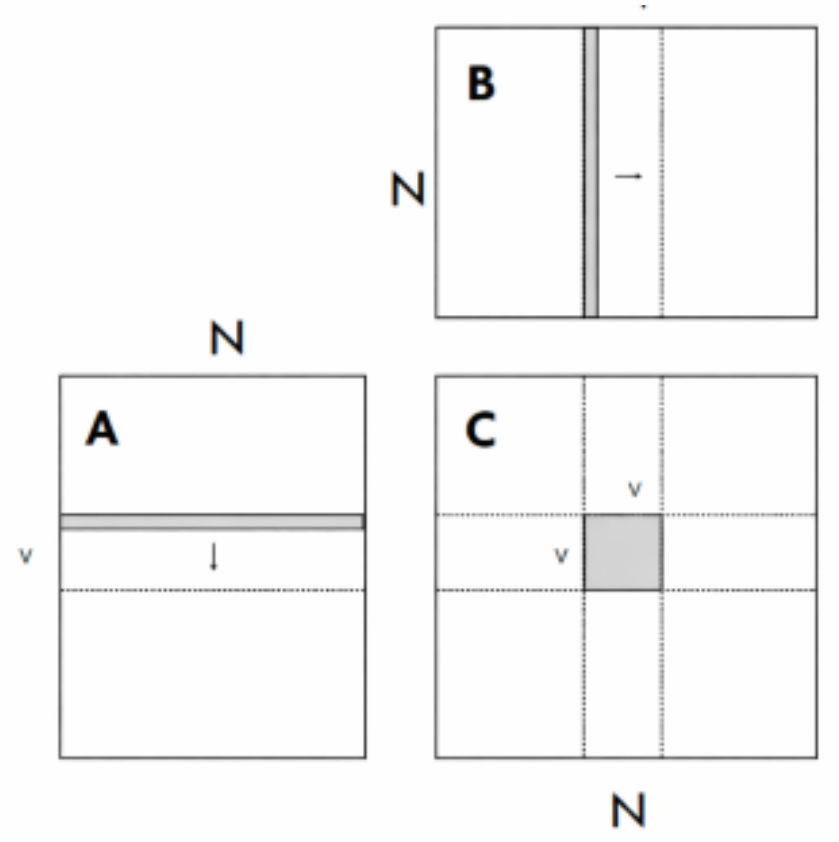

Figure 3: Matmul v1.5 lets each thread compute a small tile of outputs.

6.3 GPU Matmul v2: Partial-Sum Outer Products

The lecture then reorganizes the computation around partial sums. For each reduction step k, the thread loads:

- a short vector from A,
- a short vector from B,
- and uses their outer product to update the full local $V \times V$ accumulator tile.

This is a more efficient use of loaded data because each fetched value participates in multiple multiply-adds before being discarded.

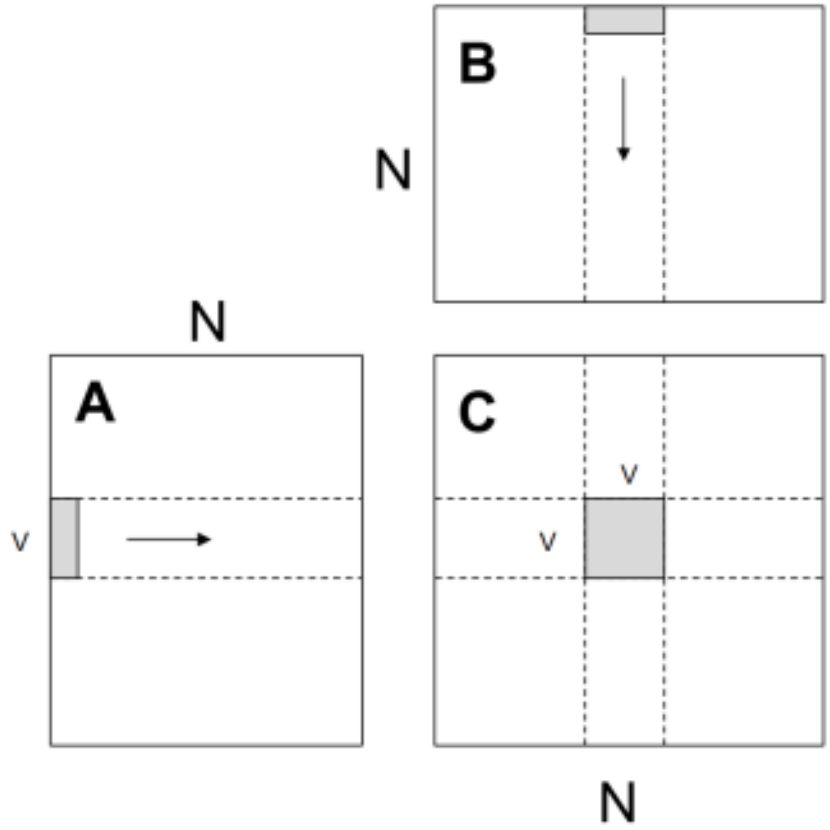

Figure 4: Matmul v2 updates a local tile via repeated partial-sum outer products.

The memory accounting improves to

$$\text{global reads per thread} = 2NV,$$

$$\text{threads} = \frac{N^2}{V^2}, \qquad \text{total global memory access} = \frac{2N^3}{V}.$$

Compared with v1.5, the same thread-local tile size now yields a much cleaner reuse pattern. This is the lecture's first major matmul optimization: do not finish one output before starting the next if shared operands can update many outputs at once.

6.4 GPU Matmul v3: Shared-Memory (SRAM) Tiling

The next jump is from *thread-local reuse* to *block-level reuse*. Registers are private, so they cannot help neighboring threads share the tiles of A and B that they all need. Shared memory solves this problem.

In v3, a thread block computes a larger $L \times L$ tile of the output matrix. Within that block, each thread still computes a smaller $V \times V$ sub-tile in registers. The block cooperatively loads tiles of A and B into shared memory in chunks of depth S, synchronizes, computes on those shared tiles, and repeats until the reduction dimension is exhausted.

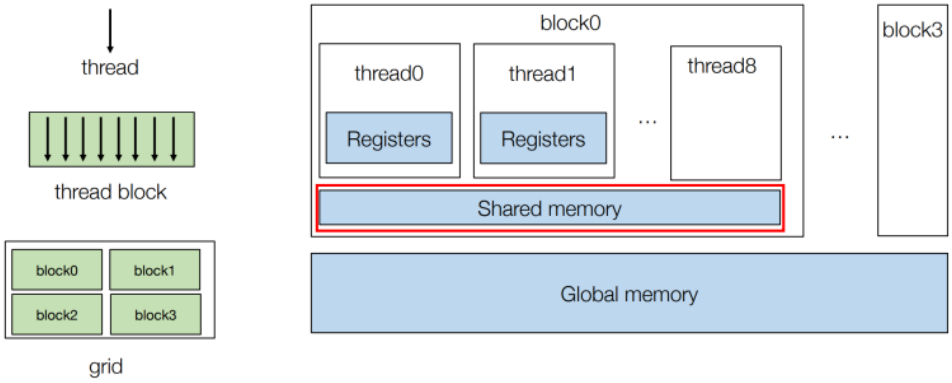

Figure 5: Shared memory acts as a software-managed cache for block-level tiling.

The nested tiling structure is the key idea:

- the **block** thinks in terms of an $L \times L$ tile of C,
- the **thread** thinks in terms of a $V \times V$ tile inside that block tile.

This shifts expensive traffic from global memory to shared memory. For one thread block, the total global memory required to compute its output tile is $2LN$, so with N^2/L^2 blocks overall, the total global memory access becomes

$$\frac{2N^3}{L}.$$

Since typically $L > V$, this is a substantial improvement over the thread-only tiling of v2. Shared-memory traffic is still present, but SRAM is much cheaper than HBM, so this is exactly the trade we want.

6.5 Cooperative Fetching

The lecture singles out *cooperative fetching* as a core CUDA pattern. Instead of having one thread load an entire tile, all threads in the block participate:

- each thread is assigned part of the shared-memory tile,
- together they load the full tile exactly once,
- a barrier ensures the tile is ready before computation begins.

This pattern is both a performance technique and a programming discipline. It balances work across threads, prevents stragglers during loading, and makes data reuse explicit.

6.6 The Remaining Optimization Problem

Even v3 is not the end of the story. Once a kernel reaches this level, performance depends heavily on tuning the tile sizes and the precise access patterns. The lecture lists several additional concerns:

- global memory coalescing,
- shared-memory bank conflicts,
- pipelining loads with computation,
- tensor cores,
- reduced precision such as FP16, FP8, or FP4.

The tuning problem centers on parameters such as L and V. Larger tiles can improve reuse, but they also consume more registers and shared memory, and may reduce occupancy. The best setting depends on tensor shape, operator structure, numeric precision, and GPU hardware. This is precisely why hand optimization remains difficult and why operator compilers and auto-tuners are valuable.

7 Takeaways

Several high-level lessons connect the entire lecture.

- **Parallelism alone is not enough.** A kernel can be correct and massively parallel yet still be slow if it generates excessive global memory traffic.
- **The memory hierarchy is the central optimization target.** Good GPU kernels progressively move reusable data closer to the compute units: first into shared memory, then into registers.
- **Block independence enables portability.** CUDA's scheduler can map the same kernel to GPUs with very different SM counts because thread blocks are designed to be independent.
- **Matmul exposes the general recipe.** Start with a trivial mapping, identify redundant traffic, exploit thread-local reuse, then exploit block-level reuse.
- **Modern performance requires tuning or compilation support.** Once kernels become deeply tiled, the search space of launch configurations and tile sizes becomes too large to manage casually, motivating systems such as cuBLAS, TVM, Triton, and other operator compilers.

In short, the lecture turns GPU programming from a collection of CUDA keywords into a systems way of thinking: structure work for SIMD execution, treat memory movement as a first-class cost, and use hierarchical tiling to convert bandwidth-limited code into high-throughput kernels.