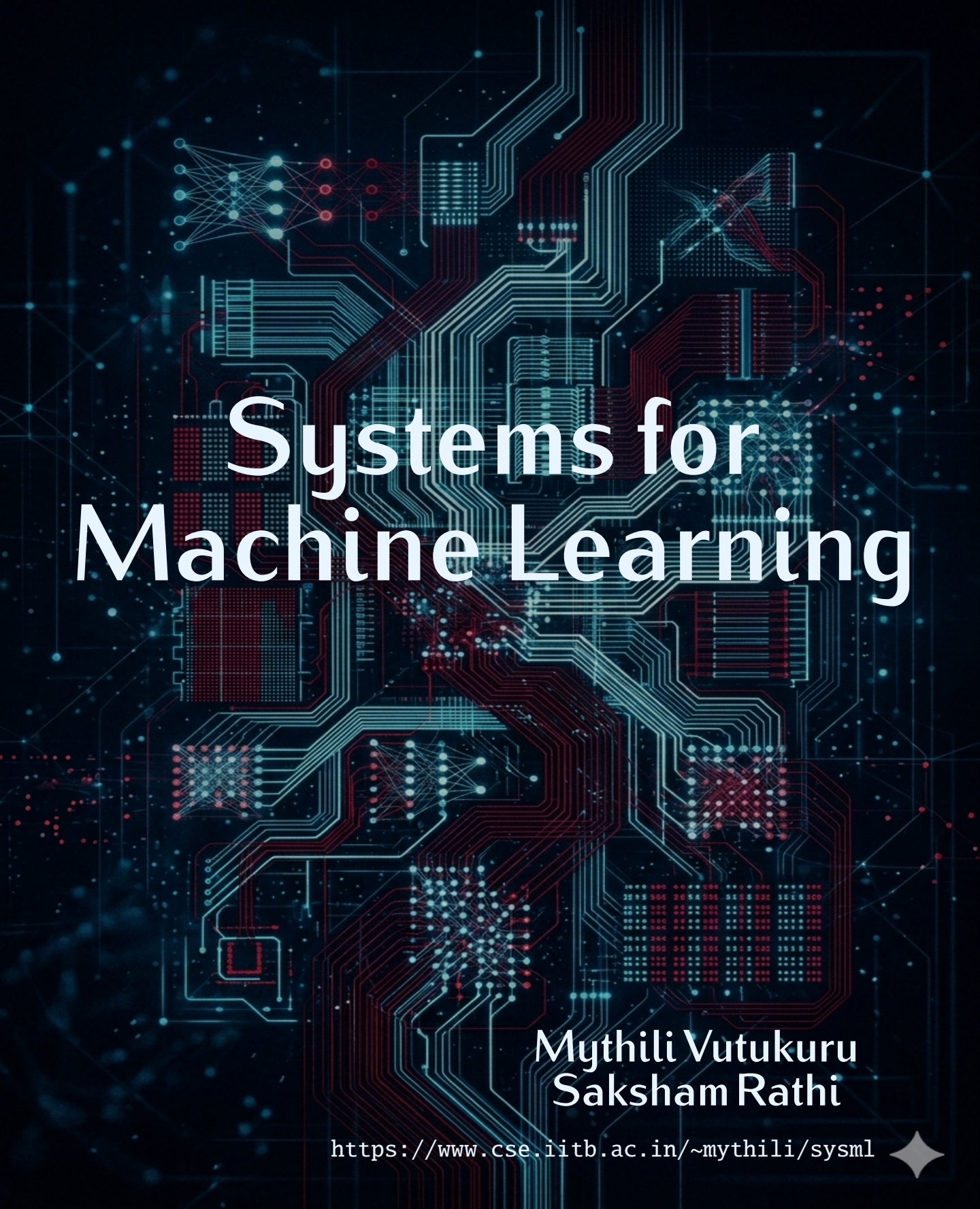

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

June 2026

3

Programming AI Hardware

3.1 Why GPUs?

Modern deep learning models place extraordinary demands on computation and memory resources. Large neural networks, such as GPT-3, contain hundreds of billions of parameters — 175 billion in GPT-3's case — and ML operations on such models require performing trillions of floating-point operations over massive datasets. To appreciate why both compute and memory matter, consider a single matrix multiplication: computing one output element requires loading an entire row from one matrix and an entire column from the other into registers, multiplying corresponding pairs of values, and accumulating their sum. A full matrix multiplication repeats this process for every output element, thereby requiring significant compute and memory bandwidth. Traditional general-purpose CPUs were not designed for this style of massively parallel, memory-intensive workloads, which led to the rise of GPUs (graphics processing units) and later domain-specific AI accelerators such as TPUs (tensor processing units).

3.1.1 CPU vs. GPU

The architectural differences between CPUs and GPUs arise from fundamentally different design goals. The architecture of general-purpose CPUs is designed to efficiently execute a wide variety of applications, including operating systems, databases, web servers, compilers, and interactive software. As a result, they devote a substantial fraction of their transistor budget to sophisticated control hardware (Figure 3.1). This includes instruction fetch and decode units, branch prediction logic, speculative execution mechanisms, out-of-order scheduling hardware, and large multi-level cache hierarchies. These components are necessary because CPUs must efficiently execute highly irregular programs with complex control flow. Further, CPUs typically expose only a small number of powerful cores (e.g., 8-64), where each core is optimized for low-latency execution of complex instruction streams. In contrast, machine learning workloads often have simpler and massively parallelizable instruction streams, and have to optimize for overall arithmetic throughput.

Another limitation is memory bandwidth. CPUs are designed around cache efficiency for irregular memory access patterns, and have lower memory access bandwidths, e.g., tens or hundreds of gigabytes-per-second. In contrast, deep learning workloads continuously stream large tensors between memory and compute units, and have a highly predictable and structured memory access pattern. This mismatch can create severe memory bandwidth bottlenecks on CPUs during large-scale ML workloads.

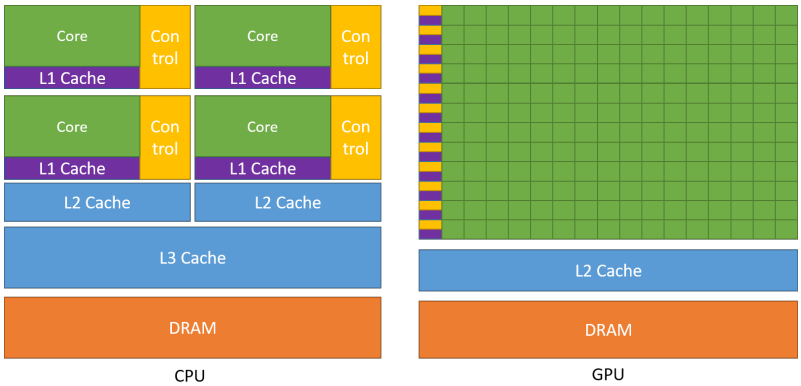


Figure 3.1: GPUs devote a larger fraction of transistors to arithmetic throughput, while CPUs devote more hardware to control logic and caching. Image credit: NVIDIA, 2025.

Characteristic	**CPU**	**GPU**
Core Design	Few complex cores	Many simple cores
Control Hardware	Extensive	Minimal
Parallelism	Limited, higher overhead	Massive, lightweight
Memory bandwidth	Low	High
Optimized for	Irregular memory access, complex control flow	Structured memory access, simpler control flow
Useful for	Sequential, low latency general purpose workloads	Parallel, high throughput numerical computation

Table 3.1: CPU vs. GPU

In contrast to CPUs, GPUs dedicate most of their transistors to arithmetic throughput (Figure 3.1). A GPU contains thousands of lightweight execution units capable of performing arithmetic operations concurrently. Rather than optimizing for low-latency sequential execution, GPUs optimize for keeping many arithmetic operations in flight simultaneously. This execution style maps naturally to machine learning workloads, because tensor computations can be decomposed into many independent parallel operations, e.g. matrix multiplication, convolution, attention computation, and vector arithmetic can all be parallelized across thousands of threads executing concurrently.

GPUs also commonly use High Bandwidth Memory (HBM), which provides terabytes-per-second data transfer rates between memory and compute units. GPU memory hierarchies also contain large register files and shared on-chip memory. These characteristics allow GPU to feed data fast enough to the arithmetic hardware, and sustain high utilization during large tensor operations typical of

deep learning training and inference.

Table 3.1 summarizes the key differences between CPUs and GPUs.

3.1.2 Evolution of AI Accelerators

Early GPUs were designed exclusively for graphics rendering workloads, such as vertex processing and pixel shading. These workloads already exhibited massive parallelism because the same mathematical operations needed to be independently applied across millions of pixels and vertices. As GPUs became increasingly programmable, researchers recognized that the architecture was also highly effective for scientific computing and machine learning workloads. NVIDIA's Tesla architecture [Lindholm et al., 2008] represented a major turning point in this evolution. Tesla unified graphics and general-purpose computation into a single programmable architecture and introduced scalable arrays of streaming multiprocessors capable of executing thousands of concurrent threads.

This transition marked the emergence of General-Purpose GPU (GPGPU) computing. NVIDIA subsequently introduced CUDA (Compute Unified Device Architecture), a programming framework that allowed developers to write parallel programs directly for GPUs using extensions to C/C++. CUDA dramatically simplified GPU programming and enabled the widespread adoption of GPUs for scientific computing, high-performance computing, and eventually deep learning. Modern NVIDIA GPUs have further evolved this architecture with specialized hardware units, such as tensor cores, which are optimized specifically for matrix multiplication and other tensor operations commonly used in neural networks.

Although GPUs are highly effective for machine learning, they remain general-purpose accelerators. Over time, the industry moved toward domain-specific architectures specifically optimized for different kinds of ML workloads.

Tensor Processing Units (TPUs)

Google's Tensor Processing Unit represents one of the most prominent departures from general-purpose GPU design. The fundamental operation in deep learning is the multiply-accumulate (MAC) computation that underpins matrix multiplication — adding the product of two numbers to a running sum, repeated billions of times per forward pass. Rather than relying on the flexible but generalized execution model of a GPU, TPUs implement a systolic array architecture in which thousands of MAC units are arranged in a grid and directly wired to their neighbours. Data flows through this grid in a pipelined fashion, and the processing elements accumulate partial sums, eliminating the need to repeatedly read and write intermediate values to/from memory. The tight coupling of compute and data movement yields substantial gains in both throughput and energy efficiency compared to GPU-based execution.

Neural Processing Units (NPUs)

Neural Processing Units are lightweight AI accelerators embedded into mobile devices, laptops, and IoT devices. Their primary role is to handle inference workloads such as speech recognition, image

classification, and real-time camera processing without draining the battery or generating excessive heat. To achieve this, NPUs are architected around fixed, low-precision datapaths (commonly INT8 or lower) and tightly coupled on-chip memory that minimizes expensive off-chip data movement. Prominent examples include Apple's Neural Engine, Qualcomm's Hexagon NPU, and Intel's AI Boost.

Intelligence Processing Units (IPUs)

Rather than a small number of heavyweight streaming multiprocessors sharing a large memory pool, Graphcore's Intelligence Processing Unit partitions compute across thousands of small, independent processor tiles, each equipped with its own local SRAM. This bulk-synchronous parallel execution model means that threads rarely need to coordinate or contend for shared memory, making the architecture particularly well-suited to sparse computations, graph neural networks, and dynamic workloads where control flow varies across examples — patterns that tend to underutilize conventional GPU pipelines.

Language Processing Units (LPUs)

Groq's Language Processing Unit is purpose-built for one of the most latency-sensitive tasks in modern AI: autoregressive token generation in large language models. During inference, each new token depends on all previously generated tokens, meaning computation cannot be easily batched across time steps. LPUs address this through deterministic, compiler-driven execution pipelines, in which every operation is scheduled at compile-time, leaving no room for runtime variability or memory access unpredictability. By eliminating the dynamic scheduling overhead present in GPU execution, LPUs achieve consistent and low per-token latency, making them attractive for real-time language model serving.

3.1.3 Computation Patterns in Machine Learning Workloads

Most machine learning workloads consist of a relatively small set of computational patterns, which strongly influence the design of modern AI hardware accelerators. Understanding them is essential for comprehending why specialized execution models are required for these workloads.

- **Vector Operations**: Many ML computations involve applying the same arithmetic operation independently across elements of a vector. Examples include element-wise addition, multiplication, normalization, simple activation functions, and reductions. These workloads are efficiently accelerated using vectorized execution models such as SIMD (Single Instruction Multiple Data) and SIMT (Single Instruction Multiple Threads).
- **Matrix Multiplication**: Matrix multiplication is the dominant computational pattern in modern deep learning, and specialized hardware units such as tensor cores are optimized specifically for high-throughput matrix multiply-accumulate operations.
- **Non-Linear Operations**: Neural networks also require non-linear activation functions such as sigmoid, GELU (Gaussian Error Linear Unit), tanh, and softmax. GPUs often include Special

Function Units (SFUs) optimized for transcendental and non-linear mathematical operations, enabling these to be evaluated efficiently without consuming general-purpose compute resources.

We now examine SIMD, the foundational vectorized execution model used in CPUs.

3.1.4 SIMD: Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) is a parallel execution model in which a single instruction operates simultaneously on multiple data elements packed into wide vector registers. Modern CPUs support SIMD through vector instruction sets such as Intel's SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), AVX-512, and ARM SVE (Scalable Vector Extension). Instead of processing one value at a time, SIMD instructions can operate on many values concurrently. For example, a 512-bit vector register can hold sixteen 32-bit floating-point values, allowing a single instruction to perform sixteen arithmetic operations in parallel. SIMD execution is highly effective for machine learning workloads because many tensor operations repeatedly apply identical arithmetic operations across large arrays of data. A simplified SIMD-style vector addition example is shown below:

Code 3.1: Vector Addition Using 256-Bit Registers

```
void vadd(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; i += 8)
        _mm256_storeu_ps(&c[i], _mm256_add_ps(
            _mm256_loadu_ps(&a[i]),
            _mm256_loadu_ps(&b[i])));
}
```

In this example, eight values are loaded into vector registers simultaneously, processed together using vector arithmetic instructions, and then written back to memory. The function operates on three floating-point arrays `a`, `b`, and `c`, moving through them in strides of eight elements per iteration to match the 256-bit register width (8×32-bit floats). The intrinsic `_mm256_loadu_ps` loads eight single-precision floats from memory into a 256-bit `__m256` register. `_mm256_add_ps` then performs eight independent floating-point additions in a single instruction. Finally, `_mm256_storeu_ps` writes all eight results back to contiguous memory in one store operation.

3.1.5 SIMT: Single Instruction Multiple Threads

While SIMD executes a single instruction across multiple data elements using vector registers, GPUs commonly employ a related but distinct execution model known as Single Instruction Multiple Threads (SIMT). In the SIMT model, many lightweight threads execute the same instruction sequence concurrently on different data elements. Rather than explicitly manipulating vector registers,

programmers write code that specifies the behaviour of a single thread. The partitioning of the data across different threads is also specified by the programmer. The GPU hardware then groups threads together and executes them in parallel with different data. This abstraction greatly simplifies parallel programming as compared to explicit vector programming. SIMT is especially well suited for machine learning workloads where operations decompose into many independent computations over different tensor elements.

CUDA is NVIDIA's parallel programming framework for GPUs, and follows this SIMT model. CUDA exposes the GPU as a massively parallel execution device capable of running thousands of lightweight threads concurrently. A minimal CUDA example for vector addition using a group of threads is shown below.

Code 3.2: Vector Addition Using a Single CUDA Block

```
__global__ void vadd(float *a, float *b, float *c) {
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

Each thread independently reads from `a[i]` and `b[i]`, computes their sum, and writes to `c[i]`, with all threads executing concurrently on the GPU. The index `i` that each thread operates on is identified by its unique thread index `threadIdx.x`. Contrast this with the SIMD version: in SIMD, the programmer explicitly loops over vector-width chunks; in SIMT, the loop is replaced entirely by the thread index, and the hardware manages parallel execution transparently.

Programming Assignment 3.1: SIMD Programming

In this assignment, you will implement a double-precision matrix multiplication using Intel AVX-512 intrinsics. A reference CPU implementation and a correctness checker are already supplied, and you need to fill in the SIMD function. You will learn new techniques, e.g. loading and storing aligned blocks of memory, and accumulating results with fused multiply-add instructions. Upon completion, you can directly observe the speedup the vectorized kernel achieves over the scalar baseline. The assignment, along with a detailed `README` covering build instructions and the relevant intrinsics is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/simd

3.2 GPU Architecture

The SIMT execution model is not merely a programming abstraction; it is directly embodied in the physical design of GPU hardware. To understand how thousands of threads are scheduled, executed, and coordinated, we now examine the hardware architecture of GPUs.

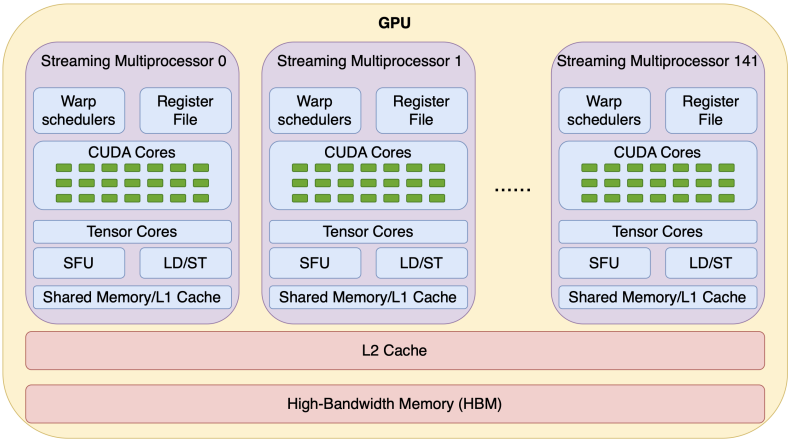


Figure 3.2: GPU hardware

3.2.1 GPU Compute

Streaming Multiprocessors, Warps, and Threads

The fundamental building block of a GPU is the **Streaming Multiprocessor** (SM). A modern GPU contains anywhere from tens to over a hundred SMs; for example, the NVIDIA H100 contains 132 SMs. Each SM is a self-contained processing unit with its own execution pipelines, register files, schedulers, and on-chip memory (Figure 3.2). The GPU achieves its aggregate throughput by running all SMs in parallel, each independently executing the threads assigned to it.

Threads are the smallest unit of execution on a GPU. Each thread runs the same kernel program but operates on a different element of data, identified by its unique thread index. GPU threads are extremely lightweight as compared to CPU threads; they have no OS scheduling overhead and no context-switch penalty because their state (registers, program counter) is held entirely in hardware on-chip. A single SM is designed to hold the register state of thousands of threads simultaneously, enabling it to switch between groups of threads in a single clock cycle at zero cost.

The GPU hardware does not schedule individual threads; instead, threads are grouped into **warps**. A warp is a fixed group of 32 threads that execute in lockstep. All 32 threads in a warp issue the same instruction in the same clock cycle, operating on different data elements. This is the SIMT model realized in hardware: the GPU broadcasts one instruction to 32 execution lanes simultaneously. A

warp is therefore the true unit of scheduling on an SM.

Each SM maintains a pool of warps in hardware. At any given clock cycle, the SM's warp schedulers select one or more ready warps and issue their next instruction to the execution pipelines. A warp is considered "ready" if its operands are available and it is not waiting on a memory operation via the load/store units. On the H100 architecture, an SM can hold up to 64 warps (i.e. 2048 threads) simultaneously. This large pool of resident warps is the key mechanism by which the GPU hides latency: while some warps stall waiting for data from off-chip memory (which may take hundreds of cycles), other ready warps continue executing, keeping the execution units busy. This is the primary reason GPUs can sustain high throughput despite relatively slow off-chip memory access.

At the software level, threads are organized into a two-level hierarchy. A thread block is a group of threads co-scheduled onto a single SM, and a grid is the full collection of blocks launched for a kernel invocation, with blocks distributed across SMs by the hardware scheduler. We cover the software hierarchy in much more detail in § 3.3.

CUDA Cores and Tensor Cores

CUDA cores are the general-purpose floating point and integer execution units within an SM. Each CUDA core executes one floating-point or integer operation per clock cycle. Modern SM designs contain 64 to 128 FP32 (single-precision floating-point) CUDA cores per SM, along with a parallel bank of INT32 cores for integer arithmetic. Since a warp consists of 32 threads, an SM with 64 FP32 CUDA cores can execute two warps' worth of FP32 arithmetic simultaneously. CUDA cores handle the bulk of general computation: element-wise tensor operations, address arithmetic, and loop control.

Tensor cores are a qualitatively different class of execution unit. Rather than performing a single multiply-add per cycle like a CUDA core, a tensor core performs an entire small matrix multiply-accumulate in a single operation. One tensor core instruction can compute $D = A \times B + C$ where A, B, and C are small matrices (e.g., 4×4 in Volta, 8×4 or 16×16 with larger tiles in later generations). Importantly, while a warp normally occupies 32 CUDA cores, a single tensor core operation can service the computation requested collectively by the entire warp. Because the overwhelming majority of deep learning computations reduce to large matrix multiplications, tensor cores provide a great throughput advantage. For example, a single SM in the H100 contains four tensor cores, together capable of processing 512 multiply-accumulate operations per clock cycle.

Warp Divergence

Warp divergence occurs when threads within the same warp take different paths through a conditional branch — for example, when some threads satisfy `if (x > 0)` and others do not. Since all threads in a warp must execute the same instruction at the same time, the GPU handles this by serializing the two paths: it first executes the `if`-branch with the non-participating threads masked off, then executes the `else`-branch with the other threads masked off. Both sides of the branch are executed by the warp, even though only half the threads do useful work in each pass. This serializa-

tion reduces effective throughput in proportion to the divergence and should be minimized in GPU code by reconstructing control flow. An example of warp divergence is illustrated in Figure 3.3.

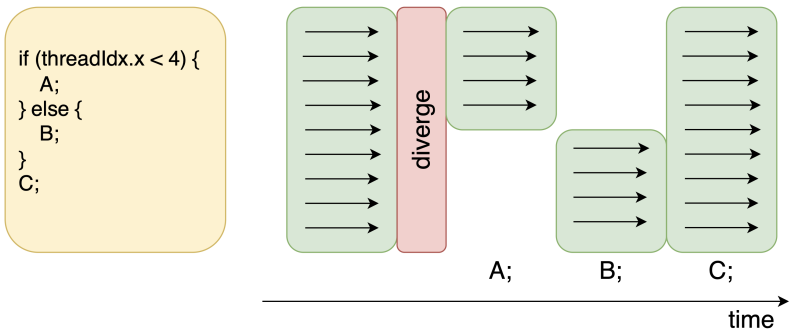


Figure 3.3: Warp divergence

Compute Throughput and FLOPs

GPU performance is most commonly expressed in **FLOP/s** (floating-point operations per second). The theoretical peak FLOP/s of a GPU can be calculated from first principles as:

$$\text{Peak FLOP/s} = \text{Number of SMs} \times \text{CUDA cores per SM} \times 2 \times \text{Clock Frequency (Hz)} \tag{3.1}$$

The factor of 2 arises because a fused multiply-add (FMA) instruction, the standard arithmetic operation in GPU pipelines, performs one multiplication and one addition in a single cycle, counting as two floating-point operations. As an example, consider the NVIDIA H100 GPU:

- 132 SMs, each with 128 FP32 CUDA cores
- Clock Frequency of 1.98 GHz
- Peak FP32 FLOP/s from CUDA cores = $132 \times 128 \times 2 \times 1.98 \times 10^9 \approx 67$ TFLOP/s (1 TFLOP = 10^{12} FLOP).

When tensor cores are engaged for matrix operations, the effective throughput is significantly higher because a single tensor core instruction completes many more multiply-adds per cycle. As an example, consider the NVIDIA A100 GPU with TF32 (tensor float 32-bit) precision:

- 108 SMs, clock rate 1.41 GHz
- Each SM contains tensor cores capable of 512 TF32 fused-multiply-add operations per clock cycle
- Counting each FMA as 2 ops, peak FLOP/s from tensor cores = $108 \times 512 \times 2 \times 1.41 \times 10^9 \approx 156 TFLOP/s$.

In practice, a GPU's reported peak throughput is the sum of contributions from both CUDA and tensor cores, since modern GPUs expose both in the same chip. Finally, note that GPU programs

rarely achieve theoretical peak FLOP/s in practice, due to memory bandwidth bottlenecks, warp divergence, and load imbalance. The ratio of achieved to theoretical peak is called **compute utilization**, and profiling tools such as Nsight Compute report this quantity directly.

Mixed-Precision Computing

Modern deep learning workloads routinely combine multiple numeric formats within a single training or inference pass, a practice known as **mixed-precision computing**. The key observation is that different parts of a computation have different sensitivity to numerical precision: weight updates and gradient accumulation require high precision to remain stable, while forward-pass matrix multiplications during inference can tolerate lower precision without a loss in model accuracy. By using lower-precision formats where possible and higher precision where necessary, mixed-precision training achieves near-optimal model quality while significantly increasing throughput and reducing memory consumption.

3.2.2 GPU Memory

Modern GPUs derive their enormous computational throughput not only from massive parallel execution, but also from a sophisticated memory system designed to feed thousands of concurrent threads with data. In practice, the performance of GPU software is determined less by arithmetic capability and more by how efficiently data can be moved through the memory hierarchy. Understanding the GPU memory hierarchy is therefore essential for understanding GPU performance itself.

The Memory Wall

As GPU architectures evolved, compute throughput has increased significantly faster than memory bandwidth. Tensor cores and a larger number of SMs allowed GPUs to execute trillions of arithmetic operations per second. However, DRAM latency and throughput improved only modestly. This growing imbalance between computation and data movement is known as the "**memory wall**".

One of the implications of the memory wall is that arithmetic units become starved for data because memory systems cannot supply operands quickly enough. Even though modern GPUs may sustain tens or hundreds of TFLOP/s of compute throughput, they can only achieve this performance when data is already available close to the execution units. A floating-point fused multiply-add operation may take only a few cycles, while fetching data from off-chip memory can take several hundred cycles. Consequently, without careful memory management, most execution units would remain idle waiting for data rather than performing useful computation.

This imbalance has become even more severe in modern AI workloads. Consider matrix multiplication in deep learning. The arithmetic operations themselves are extremely fast on tensor cores, but the matrices involved are often hundreds of megabytes or even gigabytes in size. If every arithmetic operation required a fresh DRAM access, the GPU would spend nearly all its time stalled on memory traffic. The fundamental solution to the memory wall is the **memory hierarchy**: a layered organization of memories with different capacities, latencies, and bandwidths. Smaller caches close to

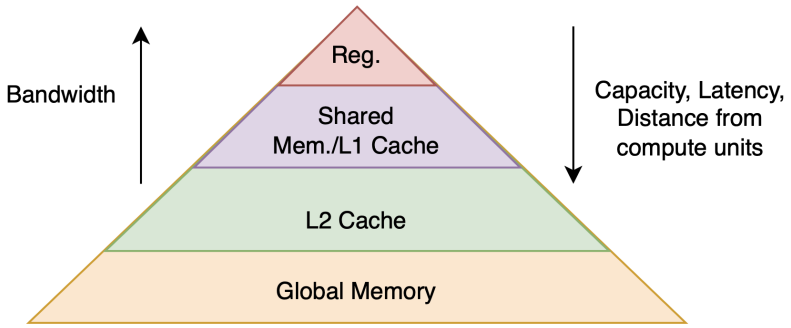


Figure 3.4: GPU memory hierarchy

the compute units provide extremely fast access, while larger memories farther away provide higher capacity at lower speed.

Figure 3.4 shows the GPU memory hierarchy, starting from registers to the global memory. As we move downward, capacity, latency, and distance from compute units increase, but bandwidth decreases. Registers offer the highest bandwidth (~20–70 TB/s) at just 1 clock cycle of latency, but with very limited capacity (~256 KB per SM). At the other extreme, global memory (HBM) provides 16–96+ GB of capacity but at a fraction of the bandwidth (0.5–3 TB/s) and latencies exceeding 300 cycles.

Registers

Registers are the fastest and smallest storage structures in the GPU. Every thread has its own private register state stored physically inside the SM. Registers hold local variables, intermediate arithmetic values, loop counters, and addresses. Register access occurs at essentially execution-unit speed, typically in a single clock cycle. Modern GPUs contain enormous register files. For example, NVIDIA's A100 and H100 both contain 256 KB of registers per SM. Since thousands of threads may reside simultaneously on an SM, and the register file must store the complete context for all resident warps, using too many registers in a thread means that the GPU cannot schedule enough warps to hide latency effectively.

Shared Memory

Shared memory (also called scratchpad memory) is an explicitly managed on-chip memory region shared among all threads in a thread block. Unlike caches, shared memory is controlled directly by the programmer. Threads cooperate to load data into shared memory, synchronize, and reuse the data multiple times. A classical example of the use of shared memory is tiled GEMM (General Matrix Multiply). Instead of repeatedly loading matrix elements from HBM, threads collaboratively load a tile each of the input matrices into shared memory and reuse these values many times to

compute the output matrix. This significantly reduces global memory traffic.

L1 Cache

Each SM also contains an L1 cache that automatically caches frequently accessed global memory data. Modern NVIDIA GPUs unify shared memory and L1 cache into a single configurable on-chip memory structure. As an example, NVIDIA's A100 has up to 192 KB of combined L1 and shared memory per SM. Latency of access is typically 20-30 cycles, which is much faster than HBM accesses. The L1 cache is particularly effective when either threads exhibit spatial locality or data reuse occurs within an SM.

L2 cache

The L2 cache is a large shared cache visible to the entire GPU, unlike L1 which is private to each SM. Modern GPUs contain very large L2 caches; for example, 40 MB for A100, 50 MB for H100, and 96 MB for RTX6000. The L2 cache is especially important for inter-SM communication, large tensor reuse, and attention workloads.

Global Memory

At the bottom of the hierarchy lies off-chip global memory, which provides the large-capacity storage required for modern AI and HPC workloads. Modern GPUs primarily use two types of global memory technologies: **High Bandwidth Memory (HBM)** and **Graphics Double Data Rate (GDDR)** memory. These technologies differ significantly in bandwidth, latency, power efficiency, packaging complexity, and cost.

Among these, High Bandwidth Memory (HBM) has become the dominant memory technology for datacenter GPUs and AI accelerators. HBM stacks multiple DRAM dies vertically through silicon vias and integrates them close to the GPU package using advanced 2.5D or 3D packaging technologies. This design delivers very high aggregate bandwidth at relatively lower clock frequencies. Modern GPUs integrate large HBM capacities; for example, NVIDIA A100 GPUs provide 40–80 GB of HBM2e, H100 GPUs provide 80–120 GB of HBM3, and B200 GPUs provide up to 192 GB of HBM3e. HBM stores model parameters, activations, gradients, optimizer states, KV caches, and other large tensors required during training and inference. Despite its enormous bandwidth, HBM latency remains relatively high, typically in the hundreds of cycles. Consequently, efficient GPU programs attempt to minimize HBM accesses and reuse data in caches/shared memory.

In contrast, many consumer and gaming GPUs rely on GDDR memory, particularly GDDR6 and GDDR6X. Unlike HBM, GDDR memories are placed around the GPU on the PCB and communicate through narrower but much higher frequency memory channels. While GDDR-based systems typically provide lower bandwidth and lower energy efficiency than HBM, they are substantially cheaper and easier to manufacture.

Table 3.2 shows typical latency and bandwidth numbers for all the hierarchy levels of GPU memory, along with system RAM.

Memory Tier	Location	Typical Capacity	Approximate Bandwidth	Latency
Registers	On-chip	256 KB per SM	20-70 TB/s	1 clock cycle
Shared Memory/L1	On-chip	64-128 KB per SM	10 TB/s	20-40 clock cycles
L2 cache	On-chip	50-192 MB	3-6 TB/s	100-200 clock cycles
Global Memory (HBM)	Off-chip	16-96+ GB	0.5-3 TB/s	300-500+ clock cycles
System RAM	Off-device	Hundreds of GBs	10-50 GB/s	10,000 cycles

Table 3.2: GPU memory hierarchy

3.2.3 GPU Communication

A GPU needs to communicate with the CPU to exchange data and programs. In addition, modern AI systems rarely operate on a single GPU in isolation. Training large language models and other modern deep learning systems typically requires many GPUs working together across one or more servers. As model sizes increase, communication between GPUs becomes just as important as the compute capability of the GPUs themselves. In many workloads, overall performance is limited not by arithmetic throughput, but by how quickly GPUs can exchange data. GPU communication occurs at multiple levels. Data may move between the CPU and GPUs inside a server, between GPUs within the same server, or across GPUs located in different servers connected through a high-speed network.

Host-to-GPU Communication

The CPU is commonly referred to as the host, while the GPUs are referred to as devices. Communication between CPU and GPU is therefore called host-to-device communication. The CPU allocates memory in system DRAM, copies data to the GPU, launches GPU programs, and copies results back after computation. The primary communication technology connecting CPUs and GPUs is PCI Express (**PCIe**). PCIe is a high-speed serial interconnect organized into lanes. Modern GPUs commonly use x16 PCIe links, meaning sixteen lanes operate in parallel. Bandwidth depends on PCIe generation, e.g. PCIe Gen3 provides approximately 16 GB/s, PCIe Gen4 provides roughly 32 GB/s, and PCIe Gen5 provides approximately 64 GB/s of bidirectional bandwidth.

Intra-Node Communication

Although PCIe is universal and flexible, it was not originally designed specifically for large-scale GPU communication. As AI workloads became increasingly communication-heavy, PCIe alone became insufficient for scaling multi-GPU systems efficiently. Modern GPU systems are organized into GPU "nodes". A GPU node typically consists of multiple GPUs, one or two CPUs, large amounts of system DRAM, local storage, and high-speed networking hardware. Inside these nodes, GPUs often communicate directly with each other using **NVLink**, completely bypassing the system RAM. NVLink is a specialized high-bandwidth interconnect designed specifically for GPU-to-GPU communication and optimized for transferring massive tensors. NVLink 3.0 in the A100 architecture provides roughly

600 GB/s of GPU-to-GPU bandwidth, while NVLink 4.0 in H100 systems provides approximately 900 GB/s, which is much higher than host-to-device PCIe bandwidth.

As GPU counts inside a server increase, pairwise NVLink connections alone become insufficient. Directly wiring every GPU to every other GPU becomes physically impractical as system size grows. To address this problem, NVIDIA uses **NVSwitch**. Instead of connecting GPUs through direct point-to-point links alone, GPUs connect to one or more NVSwitch chips, which dynamically route traffic between GPUs. Conceptually, NVSwitch behaves similarly to a network switch but is optimized specifically for GPU communication.

Inter-Node Communication

Large AI systems often require hundreds of GPUs distributed across many physical servers, which requires efficient communication between GPU nodes. The dominant networking technology in modern AI clusters is **Remote Direct Memory Access (RDMA)**. Normally, network communication requires CPU intervention, operating system involvement, and multiple memory copies between user-space and kernel-space buffers. These overheads significantly increase latency and reduce throughput. RDMA bypasses most of these overheads. Using RDMA, a network interface card can directly read from or write to memory without involving the CPU. In GPU systems, RDMA becomes even more powerful through **GPUDirect RDMA**. In this model, network adapters directly access GPU memory itself rather than staging transfers through CPU DRAM.

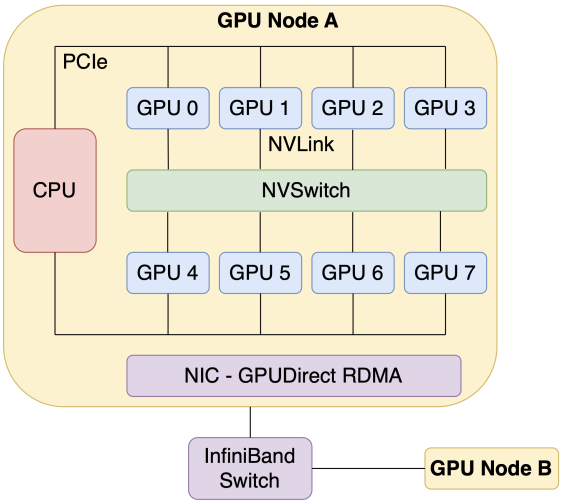


Figure 3.5: GPU communication

High throughput networking of RDMA interfaces is better achieved using **Infiniband** networking instead of traditional Ethernet networking. Compared to Ethernet, Infiniband provides lower latency,

better congestion control, hardware acceleration for collective communication operations, and native support for RDMA. Infiniband is also widely used in supercomputers and HPC systems. HDR Infiniband supports 25 GB/s links, and NDR Infiniband supports 50 GB/s links.

Figure 3.5 shows all the forms of GPU communication. At large scale, efficient communication infrastructure is often just as important as raw compute capability. A cluster with extremely fast GPUs but weak interconnects may scale poorly, while a balanced system with strong networking can sustain efficient scaling across thousands of accelerators.

Table 3.3 compares different NVIDIA GPU generations in terms of compute resources, memory subsystem characteristics, and GPU interconnect bandwidth.

Generation	SMs	CUDA Cores	Tensor Cores	HBM Capacity	Memory Bandwidth	NVLink Bandwidth
Volta	80	5,120	640	16/32 GB	900 GB/s	300 GB/s
Ampere	108	6,912	432	40/80 GB	1.6–2.0 TB/s	600 GB/s
Hopper	132	16,896	528	80 GB	3.0–3.35 TB/s	900 GB/s
Blackwell	192	24,576	768	192 GB	8 TB/s	1.8 TB/s

Table 3.3: Comparison of NVIDIA GPU generations across compute, memory, and interconnect capabilities

Practice Problem 3.1

Which of the following statements is/are true about threads in a warp?

(A) All active threads in a warp run the same instruction on the same data

(B) All active threads in a warp run the same instruction but on different data

(C) Threads in a warp that cannot execute the warp's common instruction due to a conditional statement stay inactive in that clock cycle

(D) Threads in a warp can execute different instructions in the same clock cycle depending on which branch of a conditional statement they are on

Solution: (B), (C)

(A) False. A warp follows the SIMT (Single Instruction, Multiple Threads) execution model. All active threads execute the same instruction, but each thread typically operates on its own data element.

(B) True. This is the fundamental idea behind the SIMT model. Threads within a warp execute the same instruction in lockstep, while each thread works on different data values (e.g., different array elements).

(C) True. When a branch divergence occurs (e.g., due to an if-else statement), the warp executes one branch at a time. Threads that do not belong to the currently executing branch are masked off (inactive) and remain idle during those cycles. After one branch completes, the warp executes the other branch with the corresponding threads active.

(D) False. In the SIMT execution model, a warp issues one instruction at a time. During branch divergence, different paths are executed serially, not simultaneously.

Practice Problem 3.2

Which of the following statements is/are true?

(A) Register context is saved and restored when threads are switched in and out of GPU cores

(B) Register state of a thread persists throughout the duration of execution of a thread

(C) L2 cache in the GPU is managed by the programmer

(D) Scratchpad memory in the GPU is managed by the programmer

Solution: (B), (D)

(A) False. Unlike CPU context switching, GPU threads do not typically require saving and restoring register contents when warps are scheduled. Registers for all resident threads are allocated on-chip, and the hardware scheduler simply switches between ready warps with very low overhead.

(B) True. Each thread is allocated its own set of registers when it begins execution. The values stored in these registers remain available to that thread throughout its lifetime (unless overwritten by the thread itself).

(C) False. The L2 cache is a hardware-managed cache. Data movement into and out of L2 is handled automatically by the GPU memory system; programmers do not explicitly allocate or manage L2 cache contents.

(D) True. Scratchpad memory (another name for shared memory) is explicitly managed by the programmer. The programmer decides what data to place in shared memory, when to load it, and how it is used by threads within a thread block.

3.3 CUDA Programming Model

As we saw before, neural network training and inference involve enormous numbers of arithmetic operations applied independently across large tensors, making them highly amenable to parallel execution. CUDA is the dominant programming framework used to harness this computational capability on NVIDIA GPUs. CUDA, which stands for Compute Unified Device Architecture, is NVIDIA's parallel computing platform and programming model for general-purpose GPU computing. GPUs were originally designed as fixed-function hardware accelerators for graphics rendering. Early GPUs specialized in operations such as rasterization, shading, texture mapping, and geometric transformations. Over time, GPU architectures evolved into programmable massively parallel processors capable of executing arbitrary numerical computations. CUDA provides the software abstraction layer that exposes this capability to programmers.

CUDA consists of several components. At the lowest level, the CUDA driver interfaces directly with the GPU hardware. On top of the driver sits the CUDA runtime API, which provides a simple programming interface for allocating memory, launching kernels, and managing execution. CUDA also includes highly optimized libraries such as cuBLAS for dense linear algebra, cuFFT for Fourier transforms, cuDNN for deep neural networks, and CUTLASS for tensor operations and matrix multiplication kernels. These libraries form the foundation of many modern ML frameworks.

The CUDA programming model follows the SIMT paradigm. Instead of writing a single sequential stream of instructions, programmers describe computations that can be executed simultaneously by many lightweight threads. The CUDA runtime maps these threads onto the underlying GPU hardware automatically.

A typical CUDA execution sequence proceeds as follows (Figure 3.6). First, host code running on the CPU allocates memory on both the host and device. Input data is copied from CPU memory to GPU memory using CUDA memory transfer APIs. Next, the CPU launches GPU code, called a kernel, that executes on the GPU across many parallel threads. After computation finishes, results are copied back from device memory to host memory once again using CUDA APIs.

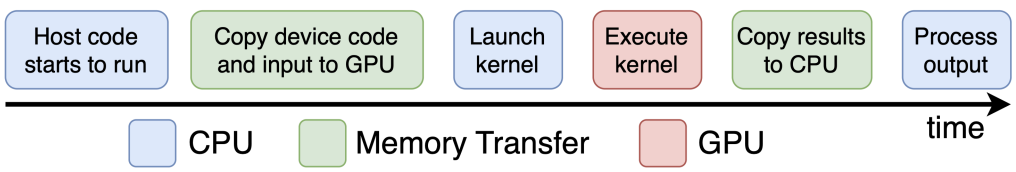


Figure 3.6: CUDA execution sequence

The CUDA programming model separates software abstractions from hardware implementation details. Programmers reason about logical thread hierarchies, while the runtime system maps those abstractions onto available hardware resources.

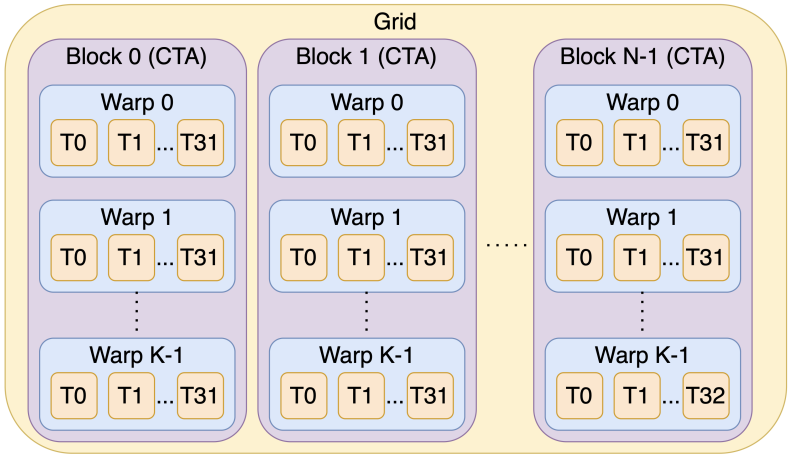


Figure 3.7: GPU software hierarchy

3.3.1 Threads, Blocks, Grids

Parallelism in CUDA programs can be understood as a hierarchy of threads, blocks, and grids (Figure 3.7). A **thread** is the smallest execution unit in CUDA. Each thread executes the same CUDA kernel code independently but operates on different data elements. Threads possess private registers and local variables. Threads are grouped into **thread blocks**, often simply called blocks. A block is a collection of threads that executes on the same SM and can cooperate using synchronization primitives and shared memory. Modern NVIDIA GPUs support up to 1024 threads per block, although the exact limit depends on the hardware architecture. The term CTA, or Cooperative Thread Array, is another name for a thread block. The terminology emphasizes that threads within a block are intended to cooperate during execution. A **grid** is a collection of thread blocks launched together for a kernel invocation. Different blocks within a grid execute independently and may run on different SMs. Both blocks and grids may be one-dimensional, two-dimensional, or three-dimensional. This multidimensional structure is extremely useful for parallelizing multi-dimensional tensor computations.

3.3.2 The Concept of a Kernel

A CUDA kernel is a special C++ function executed on the GPU. Unlike ordinary CPU functions, which execute once when called, a kernel executes simultaneously across many threads. A kernel launch uses the syntax:

```
kernel<<<numBlocks, threadsPerBlock>>>(arguments);
```

The triple-angle-bracket (triple chevron) syntax specifies the execution configuration. The first

parameter defines the number of thread blocks in the grid, while the second parameter defines the number of threads per block. Conceptually, CUDA launches a large collection of threads, each executing the same kernel code independently. A CUDA kernel also specifies which portion of the data each thread processes, using built-in indexing variables.

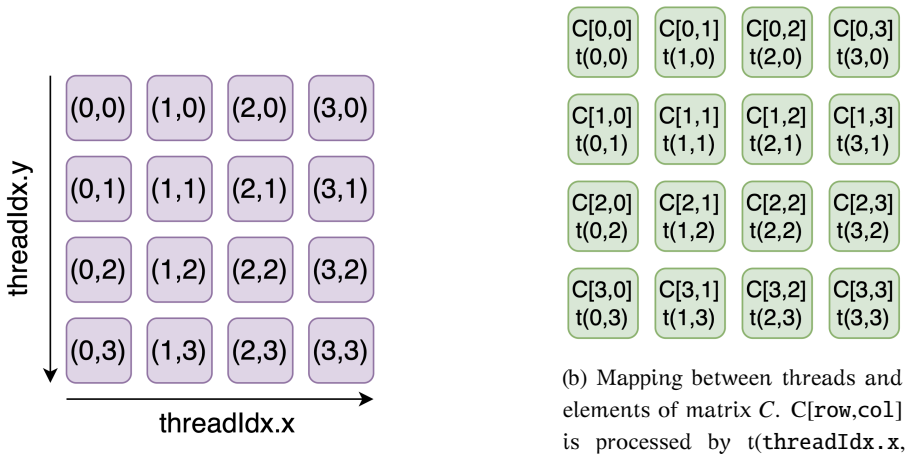
Let us take an example of adding two $N \times N$ matrices A and B (laid out in row-major order in memory) to produce $N \times N$ matrix C.

Code 3.3: Matrix Addition Using a Single CUDA Block

```
1  // Kernel definition
2  __global__ void MatAdd(float* A, float* B, float* C, int N) {
3      int row = threadIdx.y;
4      int col = threadIdx.x;
5      int index = row * N + col;
6      C[index] = A[index] + B[index];
7  }
8
9  int main() {
10     // Kernel invocation
11     int numBlocks = 1;
12     int N = 16;
13     dim3 threadsPerBlock(N, N);
14     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
15 }
```

In this example, the kernel launches a single two-dimensional block containing $N \times N$ threads, numbered using built-in variables `threadIdx.x` and `threadIdx.y`, as shown in Figure 3.8a. Note that by convention, the x-dimension of the thread identifier increases horizontally, and the y-dimension vertically in a thread block. Each thread computes one element of the output matrix C, where the mapping between row/column of a matrix element and the thread responsible for it is determined using the thread identifiers `threadIdx.x` and `threadIdx.y`. Note that there are many ways to do this mapping, and the code example is one of the possible ways. Figure 3.8b shows the elements of matrix C, and the two-dimensional identifier of the thread that computes each element for this code example shown above.

The keyword `__global__` indicates that this function is a GPU kernel callable from the CPU. When this kernel is launched on the GPU, $N \times N$ lightweight threads execute identical code in parallel, each operating on a different portion of the input and output matrices.



(a) Two-dimensional thread block

(b) Mapping between threads and elements of matrix C. C[row,col] is processed by t(threadIdx.x, threadIdx.y.

Figure 3.8: Thread identifiers and mapping with the output matrix C

3.3.3 Multidimensional Thread Hierarchy

CUDA organizes threads hierarchically because large tensors often possess a multi-dimensional structure. A matrix naturally has rows and columns, while images in deep learning may possess batch, channel, height, and width dimensions. Thread blocks may therefore be one-dimensional, two-dimensional, or three-dimensional. Similarly, grids themselves may also be multi-dimensional.

For example, the code shown below configures a two-dimensional grid with 64×64 blocks, with each block containing $16 \times 16 = 256$ threads.

```
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(64, 64);
```

CUDA provides several built-in variables that define the execution hierarchy. `threadIdx` denotes the unique index of a thread within its thread block. Similarly, `blockIdx` is the unique index of a thread block within the grid. These indices can be multi-dimensional with `x`, `y`, and `z` components. Next, `blockDim` specifies the dimensions of the thread block, and `gridDim` specifies the dimensions of the grid. These dimensions also have `x`, `y`, and `z` components. These built-in variables help us partition input data across threads and blocks in a grid, as per the SIMT model, so that each thread can execute identical kernel code but on different data elements.

The following example illustrates a vector addition CUDA kernel using multiple blocks. This code adds two 1024x1024 vectors A and B to form a 1024-sized vector C. This task is accomplished using several one-dimensional thread blocks with 256 threads each. Given the total size of the tensor, we require 1024/256 = 4 such blocks to provision one thread for each vector element. The expression `(N`

+ `threadsPerBlock - 1) / threadsPerBlock` computes the ceiling of the division, ensuring that enough blocks are launched even when N is not evenly divisible by the block size. For example, if $N = 1000$ and `threadsPerBlock` = 256, then four blocks are required.

Code 3.4: Vector Addition Using Multiple Blocks

```
__global__ void vecAdd(float* A, float* B, float* C, int N) {
    int globalIndex = threadIdx.x + blockDim.x * blockIdx.x;
    if (globalIndex < N) C[globalIndex] = A[globalIndex] + B[globalIndex];
}
int main() {
    int N = 1024;
    int threadsPerBlock = 256;
    int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;
    vecAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
}
```

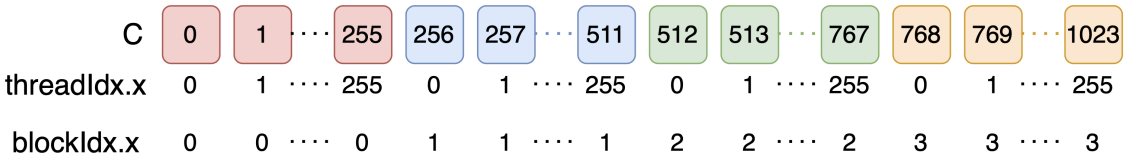


Figure 3.9: Output array C and `threadIdx.x` and `blockIdx.x` of threads that compute its elements

The `globalIndex` variable maps the thread and block index values of a thread to a global index in the vector. It achieves this by first skipping all the threads belonging to the preceding blocks (`blockIdx.x` × `blockDim.x`) and then adding the local thread number within the current block (`threadId.x`). Figure 3.9 illustrates the output array C, and the thread and block identifiers of the thread that computes each element of this array.

Next, we look at a more complicated example of matrix addition using a two-dimensional thread block and grid hierarchy. The code below shows the addition to two NxN matrices A and B to produce a matrix C. In this example, each thread block of 16x16 threads computes a 16×16 tile of the matrix. The code launches enough such thread blocks to span the entire NxN matrix. The partitioning of matrix elements to threads happens in both row and column dimensions, using the built-in block and thread identifiers and dimensions, much like we saw for the one-dimensional case.

Code 3.5: Matrix Addition Using Multiple Blocks

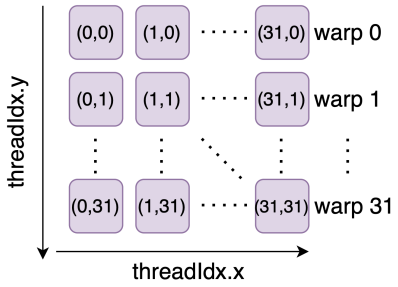
```
__global__ void matAdd(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        int index = row * N + col;
        C[index] = A[index] + B[index];
    }
}

int main() {
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + 15) / 16, (N + 15) / 16);
    matAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
}
```

As a final example to illustrate multi-dimensional thread hierarchies, the code below shows a simple matrix multiplication kernel.

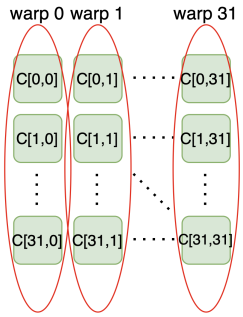
Code 3.6: Matrix Multiplication Using Multiple Blocks

```
__global__ void matMul(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++)
            sum += A[row * N + k] * B[k * N + col];
        C[row * N + col] = sum;
    }
}
int main() {
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + 15) / 16, (N + 15) / 16);
    matMul<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
}
```

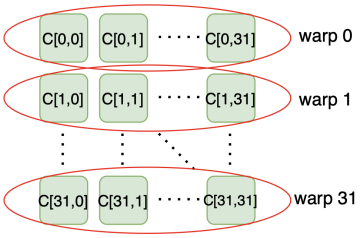
Like matrix addition, the kernel uses a two-dimensional thread and block hierarchy, with each thread uniquely identifying a `(row, col)` element computed from its `blockIdx`, `blockDim`, and



(a) Two-dimensional thread block and warps



(b) Mapping between warps and elements of matrix C for uncoalessed access pattern



(c) Mapping between warps and elements of matrix C for coalessed access pattern

Figure 3.10: Uncoalesced and coalesced memory accesses

`threadIdx` values. Each thread is responsible for computing this single element of the output matrix C, which requires accumulating the dot product of an entire row of A with an entire column of B. Specifically, the thread responsible for `(row, col)` iterates over $k = 0, 1, \ldots, N-1$, accumulating the product `A[row][k] * B[k][col]` into a local variable `sum`, and writes the result to `C[row * N + col]` once the loop completes. The grid and block dimensions are identical to those used in the matrix addition example: each thread block covers a 16×16 tile of the output matrix, and enough blocks are launched in both dimensions to span the full $N \times N$ output.

3.3.4 Memory Coalescing

In GPUs, global memory has a high memory bandwidth but long access latency. Servicing memory requests separately leads to an underutilization of this memory bandwidth. Therefore, memory hardware fetches multiple consecutive addresses together on every memory access. The access and transfer of these consecutive memory locations is referred to as a DRAM burst. Now, if multiple consecutive addresses are issued together by GPU code, a single DRAM burst can service them, leading to improved performance. Such **coalesced** memory access patterns occur when threads in a warp access consecutive memory addresses.

Now, how are threads grouped into warps? Consider a thread block of 32×32 threads, shown in Figure 3.10a. Threads in this block can be represented by their two-dimensional identifier (`threadIdx.x` and `threadIdx.y`), or a one-dimensional identifier computed as `threadID = threadIdx.x + threadIdx.y` × `blockDim.x`. A set of 32 threads with consecutive `threadID` values are grouped into warps, as shown in Figure 3.10a. In this example thread block, all threads with the same `threadIdx.y` get grouped into the same warp, and `threadIdx.x` varies inside a warp.

Consider the addition of two matrices A and B (laid out in row-major format in memory), each

of size $N \times N$, where $N = 32$, as shown below.

Code 3.7: Uncoalesced Matrix Addition

```
__global__ void uncoalesced_matAdd(float* A, float* B, float* C, int N) {
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        int index = row * N + col;
        C[index] = A[index] + B[index];
    }
}
```

This task is performed by a thread block of 32×32 threads, as shown in Figure 3.10a, where the matrix dimension `row` maps to `threadIdx.x` (which varies across threads in a warp), and matrix dimension `col` maps to `threadIdx.y` (which stays the same across all the threads in a warp). As a result, consecutive threads in a warp access elements along the column dimension of the matrix (Figure 3.10b), which are scattered in memory for a row-major matrix layout. Such an assignment of data elements to threads in a warp makes it difficult to coalesce memory accesses, leading to poor performance.

If we modify the code instead to that shown in below, where `row` is mapped to `threadIdx.y` and `col` is mapped to `threadIdx.x`, consecutive threads in a warp access elements along the row dimension of the matrix (Figure 3.10c), which are contiguously placed in memory. This is a coalesced access pattern, and aligns with what DRAM bursts are designed to service, and therefore leads to better performance.

Code 3.8: Coalesced Matrix Addition

```
__global__ void coalesced_matAdd(float* A, float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        int index = row * N + col;
        C[index] = A[index] + B[index];
    }
}
```

This discussion illustrates the need to pay close attention to how data elements are partitioned across threads in the SIMT model, in order to optimize performance.

Programming Assignment 3.2: Vector Addition

In this assignment, you will be given a partially complete CUDA source file where the core kernel function `vecAdd` has three lines intentionally left blank: computing the unique global thread index, adding a bounds check to prevent out-of-bounds memory access when the thread count exceeds the array size, and writing the actual element-wise sum into the output array. The surrounding infrastructure of memory allocation, data transfer, kernel launch, and a CPU reference implementation are all provided. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/vector_addition

Programming Assignment 3.3: Convolutional Neural Network

In this assignment, you will implement two CUDA kernels: a 2D convolution operation and a ReLU activation function. You will be writing CUDA code, where each thread independently computes a single output element. For convolution, this means sliding a small filter kernel across an input matrix and computing weighted sums; for ReLU, it simply means clamping negative values to zero. The exercise is designed to build intuition for GPU thread organization, with students using a 2D thread grid for the convolution kernel and a 1D grid for ReLU, while handling boundary conditions to avoid out-of-bounds memory accesses. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/convolutional_neural_network

Programming Assignment 3.4: Sliding Window Attention

In this assignment, you will implement a CUDA kernel for causal sliding window attention, a memory-efficient variant of the standard transformer self-attention mechanism. Instead of allowing every token to attend to all previous tokens (which costs $O(N^2)$ in memory and compute), each token is restricted to only look back at the most recent W positions. Your job is to write a GPU kernel that takes batched query, key, and value tensors of shape $[B, H, N, D]$ and produces the attention output Y — doing so with a three-pass softmax: first finding the max score in the window, then computing the sum of exponentials, then accumulating the weighted value sum. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/sliding_window_attention_cuda

3.4 CUDA APIs

So far, we have studied the CUDA programming model, focussing primarily on the thread hierarchy. Next, we will discuss CUDA APIs for memory, synchronization, events, error handling, atomic operations and so on.

3.4.1 CUDA Memory APIs

CUDA programs execute across two physically separate memory spaces: host memory and device memory. Host memory refers to the main system memory managed by the CPU, while device memory refers to the GPU's onboard DRAM/HBM. Since the CPU and GPU possess independent memory spaces, data required by GPU kernels must generally be transferred explicitly between host and device memory. CUDA therefore provides a rich collection of memory management APIs for allocation, initialization, transfer, synchronization, and unified memory management.

Host Memory Allocation

Before discussing GPU memory allocation, it is useful to briefly review ordinary host-side memory allocation on the CPU. In CUDA applications, host memory typically stores input data before transfer to the GPU and receives output data after GPU execution completes. The most common host memory allocation mechanism in C uses the `malloc()` function:

```
void* malloc(size_t size);
```

The argument specifies the number of bytes to allocate in system memory. The function returns a pointer to the allocated memory region. Allocated host memory should eventually be released using `free()`. Memory allocated using `malloc()` is pageable memory, meaning the operating system may move pages between RAM and disk as part of virtual memory management.

Pageable memory transfer requires an intermediate staging operation before data can be transferred to the GPU. CUDA therefore also provides pinned, or page-locked, memory allocations using `cudaMallocHost` and `cudaHostAlloc`. Pinned memory cannot be paged out by the operating system, allowing direct DMA transfers between CPU and GPU memory. The prototype is[1]:

```
cudaError_t cudaMallocHost(void** ptr, size_t size);
```

Pinned memory significantly improves transfer bandwidth. However, excessive pinned allocations can reduce overall system performance because pinned pages are unavailable to the operating system's virtual memory manager. Pinned memory allocated with `cudaMallocHost` must be released

[1]The function accepts `void**` rather than `void*` because it must write the allocated address back to the caller's pointer variable. In C, all arguments are passed by value, so if the signature were `void*`, the function would receive only a copy of the pointer, and any assignment made inside the function would not be visible to the caller. By accepting `void**`, the function receives the address of the caller's pointer variable, allowing it to dereference it and store the newly allocated address directly into the caller's own `ptr`.

using `cudaFreeHost()` rather than the standard `free()`, as the CUDA runtime requires its own deallocation path to properly unpin the pages and update its internal memory tracking.

Device Memory Allocation

The most common form of GPU memory allocation uses device global memory. As we saw before, global memory resides physically on the GPU and is accessible by all threads of all blocks. However, since device memory is separate from CPU memory, pointers allocated on the GPU are valid only inside device code. CUDA provides the function `cudaMalloc()` for allocating memory on the GPU:

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

The first parameter is a pointer to a device pointer, while the second parameter specifies the number of bytes to allocate. The function returns a CUDA error code indicating success or failure. A typical allocation sequence appears as follows:

Code 3.9: Allocating Device Memory

```
1 float* d_A;
2 int N = 1024;
3 cudaMalloc((void**)&d_A, N*sizeof(float));
```

After allocation, `d_A` points to memory located on the GPU. The CPU cannot dereference this pointer directly because it refers to device address space rather than host address space. Unlike ordinary CPU allocations using `malloc` or `new`, CUDA allocations are relatively expensive operations because they involve communication with the GPU driver and device memory manager. Therefore, in high-performance CUDA applications, buffers are usually allocated once and reused repeatedly.

Releasing Device Memory

Memory allocated using `cudaMalloc` must eventually be released using `cudaFree`. Failure to release device memory results in memory leaks, exactly as with CPU-side dynamic allocations. The function syntax is:

```
cudaError_t cudaFree(void* devPtr);
```

After the call completes, the pointer becomes invalid and must not be used again. GPU memory is a limited resource, and unreleased allocations may eventually prevent further kernel launches or memory allocations.

Memory Initialization

CUDA provides the `cudaMemset` API for initializing device memory to a fixed byte value. The API resembles the standard C library function `memset`, but it operates on GPU memory rather than host memory. The function syntax is:

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count);
```

The function sets the first `count` bytes of device memory to the specified byte value. It is important to note that `cudaMemset` operates at the byte level rather than at the element level; therefore, initializing arrays to arbitrary floating-point values using this function is incorrect.

Data Transfer

Since host memory and device memory are usually separate physical memory spaces, data transfer is a central component of CUDA programming. The `cudaMemcpy` API copies data between host and device memory regions. The prototype is:

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                       cudaMemcpyKind kind);
```

The final parameter specifies the direction of transfer. Common transfer types include `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, and `cudaMemcpyHostToHost`. Memory transfer operations are often expensive because data must traverse the PCIe or some other kind of interconnect between the CPU and GPU.

Asynchronous Transfers

Ordinary `cudaMemcpy` operations are synchronous with respect to the host. That is, the CPU waits until the transfer completes before continuing execution. CUDA also provides asynchronous transfer APIs that allow communication and computation to overlap. The prototype is:

```
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
                            cudaMemcpyKind kind, cudaStream_t stream = 0);
```

Asynchronous copies enable concurrent kernel execution and data transfer when supported by hardware. Such transfers generally require pinned host memory, which was discussed before.

Unified Memory Management

Another important software abstraction inside modern GPU systems is **Unified Virtual Memory (UVM)**. Traditional CUDA programming requires explicit separation between host memory and device memory. This model provides fine-grained control but increases programming complexity considerably. UVM simplifies this model by exposing CPU memory and GPU memory as a single, unified virtual address space. Memory pages can automatically migrate between the CPU and GPU depending on access patterns. If a GPU accesses data currently located in CPU memory, the runtime system migrates the required pages to GPU memory automatically. The primary API is:

```
cudaError_t cudaMallocManaged(void** devPtr, size_t size,
                              unsigned int flags = cudaMemAttachGlobal);
```

An example appears below:

Code 3.10: Usage of Unified Memory

```
float* A;
cudaMallocManaged(&A, N * sizeof(float));
for (int i = 0; i < N; i++) A[i] = i;
kernel<<<numBlocks, threadsPerBlock>>>(A);
cudaDeviceSynchronize();
printf("%f\n", A[0]);
cudaFree(A);
```

In this model, the same pointer is accessible from both CPU and GPU code. CUDA automatically migrates memory pages to the processor currently accessing them. Unified memory simplifies programming, however, automatic page migration introduces runtime overheads. If page movement occurs frequently during kernel execution, performance may degrade substantially. Modern CUDA implementations also provide APIs for controlling unified memory behaviour explicitly. For example, `cudaMemPrefetchAsync` allows programmers to prefetch managed memory to a specific device before computation begins. Similarly, `cudaMemAdvise` allows programmers to provide hints regarding expected access patterns.

Shared Memory

Shared memory is an explicitly managed on-chip memory region shared among threads belonging to the same thread block. It allows threads within a block to cooperate efficiently by exchanging intermediate data. Shared memory is allocated once per thread block, and every block receives its own independent shared memory region. Since all threads in a block execute on the same SM, they can access the same shared memory allocation. Threads from different blocks cannot access each other's shared memory.

A simple shared memory declaration appears below:

Code 3.11: Static Shared Memory

```
__global__ void kernel(float* A) {
    __shared__ float cache[256];
    int idx = threadIdx.x;
    cache[idx] = A[idx];
    A[idx] = cache[idx] * 2.0f;
}
```

In this example, each thread copies one value from global memory into shared memory. CUDA also supports dynamically allocated shared memory, which is useful when the size is not known at

compile time:

Code 3.12: Dynamic Shared Memory

```
__global__ void kernel(float* A) {
    extern __shared__ float cache[];
}
```

The shared memory size for the code above must be specified during kernel launch as follows:

```
kernel<<<numBlocks, threadsPerBlock, sharedMemoryBytes>>>(A);
```

With this, each thread block gets `sharedMemoryBytes` size of shared memory. Shared memory is heavily used in matrix multiplication, convolution kernels, reductions, and tensor computations because it allows data reuse while avoiding repeated expensive global memory accesses.

3.4.2 CUDA APIs for Synchronization and Control

Beyond memory allocation, CUDA provides a wide collection of APIs that enable thread cooperation, synchronization, execution control and error handling. In real GPU applications, programmers rarely write kernels consisting only of arithmetic operations. Instead, kernels continuously coordinate threads, synchronize execution, communicate through shared memory, and detect runtime failure through specialized CUDA language qualifiers and runtime APIs.

CUDA Function Specifiers

CUDA extends the C++ language with several function specifiers that define where a function executes and from where it may be called. The most commonly used specifier is `__global__`. Functions marked with `__global__` are CUDA kernels. These functions execute on the GPU but are launched from the CPU. Kernel functions always return `void`. The second important specifier is `__device__`. A function marked with `__device__` executes on the GPU and may only be called from other GPU functions. They can also have a return type other than void. They run on the same thread as the calling kernel. CUDA also provides the `__host__` specifier. Host functions execute on the CPU and behave like ordinary C++ functions. In practice, ordinary functions are implicitly host functions even if the specifier is omitted. A function may also simultaneously possess both host and device qualifiers. Such functions are compiled twice, once for CPU execution and once for GPU execution. Table 3.4 provides a summary of all the specifiers. CUDA also provides the `__forceinline__` and `__noinline__` qualifiers, which influence compiler inlining behaviour.

Thread Synchronization

Threads within a block frequently cooperate through shared memory. Since threads execute concurrently and potentially at different speeds (only threads belonging to the same warp are executed in a

Specifier	Executes on	Called from	Return type
`__global__`	GPU	CPU	`void` only
`__device__`	GPU	GPU only	Any
`__host__`	CPU	CPU only	Any
`__host__ __device__`	CPU and GPU	CPU and GPU	Any

Table 3.4: CUDA function specifiers

lock-step fashion), synchronization becomes necessary to guarantee correctness. CUDA provides the barrier synchronization primitive:

```
__syncthreads();
```

This function blocks execution until all threads in the block reach the synchronization point. Consider the following example:

Code 3.13: Thread Synchronization

```
__global__ void kernel(int* input_data, int* output_data) {
    __shared__ int shared_data[128];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    __syncthreads();
    if (threadIdx.x == 0) {
        int sum = 0;
        for (int i = 0; i < blockDim.x ; ++i) {
            sum += shared_data[i];
        }
        output_data[blockIdx.x] = sum;
    }
}
```

Without synchronization, thread-0 might attempt to read shared memory values before other threads finish writing them. Please note that this mechanism cannot be used to synchronize across blocks inside a kernel launch because blocks may execute independently on different SMs.

Device Synchronization

Kernel launches are generally asynchronous with respect to the CPU. After launching a kernel, the CPU immediately continues execution without waiting for GPU completion. CUDA therefore provides explicit synchronization APIs. The most important function is:

```
cudaDeviceSynchronize();
```

This function blocks the CPU until all previously launched GPU work completes.

CUDA Events

CUDA events are lightweight synchronization and timing primitives. They are commonly used for performance measurement. A typical event workflow appears below:

Code 3.14: CUDA Events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
kernel<<<blocks, threads>>>();
cudaEvenRecord(stop);
cudaEventSynchronize(stop);
float ms;
cudaEventElapsedTime(&ms, start, stop);
```

CUDA events provide highly accurate GPU timing measurements and are frequently used during kernel optimization.

Error Detection and Error Handling

CUDA operations may fail for numerous reasons including invalid memory accesses, illegal launch configurations, insufficient memory, driver failures, synchronization errors, and unsupported hardware features. CUDA therefore provides extensive error handling APIs. Almost every CUDA runtime API returns a value of type `cudaError_t`. This value indicates success or failure. The simplest error-checking pattern appears below:

Code 3.15: Basic CUDA Error Checking

```
cudaError_t err;
err = cudaMalloc((void**)&d_A, N * sizeof(float));
if (err != cudaSuccess) {
    printf("CUDA Error: %s\n",
    cudaGetErrorString(err));
}
```

The function `cudaGetErrorString()` converts CUDA error codes into human-readable strings.

Kernel launches require special attention because launches are asynchronous. A launch may appear successful initally even if the kernel later fails during execution. CUDA therefore provides `cudaGetLastError()`. A common debugging pattern appears below:

Code 3.16: Kernel Error Detection

```
kernel<<<blocks, threads>>>(A);
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("Launch Error: %s\n",
    cudaGetErrorString(err));
}
```

Atomic Operations

CUDA supports atomic memory operations that allow multiple threads to update shared variables safely without race conditions. Examples include: `atomicAdd()`, `atomicSub()`, `atomicCAS()`, `atomicMin()`, and `atomicMax()`. Atomic operations are essential in reductions, histograms, and graph algorithms. However, excessive atomic operations may serialize execution and reduce performance substantially.

CUDA Streams

By default, CUDA operations such as kernel launches, memory copies, and synchronization calls are issued into a single execution stream known as the **default stream**. Operations inside a stream execute sequentially in issue order. Consequently, if all GPU work is submitted to a single stream, memory transfers and kernel executions occur one after another, potentially leaving portions of the GPU idle. CUDA streams provide a mechanism for expressing concurrency. Operations within one stream execute one after other, while operations in different streams may execute concurrently. Conceptually, streams allow the programmer to decompose GPU work into multiple independent pipelines. A CUDA stream is represented using the type `cudaStream_t`. Streams are created using:

Code 3.17: Creating CUDA Streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
```

Once created, streams may be associated with kernel launches and asynchronous memory operations. For example:

```
kernel<<<numBlocks, threadsPerBlock, 0, stream1>>>(A);
```

The fourth kernel launch parameter specifies the stream into which the kernel is issued. Similarly, asynchronous memory transfers may also be associated with streams. For example, consider two independent computation pipelines:

Code 3.18: Concurrent Stream Execution

```
1 cudaMemcpyAsync(..., stream1);
2 kernelA<<<..., stream1>>>();
3
4 cudaMemcpyAsync(..., stream2);
5 kernelB<<<..., stream2>>>();
```

The GPU scheduler may overlap these operations concurrently. While one kernel executes, another transfer may proceed simultaneously. CUDA also provides APIs for explicit stream synchronization. The function `cudaStreamSynchronize(stream)` blocks the CPU until all operations in the specified stream complete.

Practice Problem 3.3

Consider two independent operations that need to be executed. Described below are different ways in which the operations can be distributed and parallelized for execution on a GPU.

(A) The two operations are distributed across different threads within the same warp.

(B) The operations are assigned to different thread blocks within the same grid.

(C) The operations are executed as part of different kernel launches.

(D) Each thread alternates between executing instructions from the two operations (i.e., instruction-level interleaving within a thread).

For which of the above scenarios are the two operations guaranteed to execute on the same Streaming Multiprocessor (SM)?

Solution: (A), (D)

All threads within a thread block are always scheduled together on the same SM and share its resources (registers, shared memory, warp schedulers). Therefore:

(A) Guaranteed. All threads in a warp belong to the same thread block and hence always reside on the same SM. Since a warp is the unit of scheduling on an SM, both operations are guaranteed to execute on the same SM.

(B) Not Guaranteed. Different thread blocks may be scheduled on different SMs by the runtime, depending on resource availability. There is no mechanism to force two blocks onto the same SM.

(C) Not Guaranteed. Different kernel launches are fully independent; blocks from each launch can be spread across any available SMs. Even if the two kernels are launched sequentially, there is no guarantee of

co-location on the same SM.

(D) Guaranteed. Both operations execute within a single thread, and a thread always runs on exactly one SM for its entire lifetime. Since no migration across SMs occurs mid-execution, both operations are trivially co-located.

Practice Problem 3.4

Consider the following CUDA kernel that computes the transpose of a matrix. The kernel takes as arguments the pointers to two matrices A and B, each of size $N \times N$. The matrices are stored in row-major order, and the elements must be accessed using a single index, e.g., $A[i]$. The program uses thread blocks of 32x32 threads each, and creates as many blocks as required to transpose the entire matrix. Each thread in a block reads one element of A at a certain (row,col) index and writes it to its suitable position in the transposed matrix B.

(i) Complete the CUDA code shown below to correctly implement the transpose functionality. Note that there may be multiple ways of assigning matrix elements to threads. You can use the built-in variables `blockDim`, `blockIdx`, and `threadIdx` in your solution.

```
__global__ void transposeKernel(float *A, float *B, int N) {
  int row = ________________________________________;
  int col = ________________________________________;
  if (row < N && col < N) {
    B[________________________] = A[__________________];
  }
}
```

Solution: Note that multiple solutions are possible, one of which is below. Each thread computes its unique `(row, col)` coordinate from its block and thread indices, then reads $A[\text{row} \cdot N + \text{col}]$ (the element at that position in row-major order) and writes it to $B[\text{col} \cdot N + \text{row}]$, which is the transposed position.

```
__global__ void transposeKernel(float *A, float *B, int N) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < N && col < N) {
    B[col * N + row] = A[row * N + col];
  }
}
```

(ii) Does your solution ensure coalesced access in the global memory (HBM) for both arrays? That is, do threads in the same warp access contiguous elements in both the arrays? Answer yes or no. If yes, justify your answer. If no, describe a method by which you can ensure contiguous accesses on the global memory of both arrays during the transpose operation.

Solution: No. With the way the code is currently written, it is not possible to ensure coalesced access for both arrays simultaneously. Threads in a warp share the same `threadIdx.y` and differ only in `threadIdx.x`, so they read consecutive columns in the same row of A (coalesced), but write to consecutive rows in the same column of B (strided, non-coalesced), since the write index is `col * N + row`.

We can solve this problem using shared memory tiling. The idea is to load a 32×32 tile of A into a `__shared__` array with coalesced reads (threads in a warp read a contiguous row of A), then call `__syncthreads()`, and finally write the tile to B in a transposed fashion. Crucially, we change the order of the indices during the write of B from shared memory, to ensure coalesced access over B as well.

Practice Problem 3.5

Consider the following CUDA kernel which uses a shared memory array to compute an output from the input. Suppose the code is invoked with a one-dimensional block of 1024 threads, and performs its computation on an input array of size 1024. Assume the code runs without any errors.

```
1  __global__ void mysteryKernel(int *input, int *output) {
2    int tid = threadIdx.x;
3    shared[tid] = input[tid]; //shared memory array
4    __syncthreads();
5    for (int s = blockDim.x / 2; s > 0; s /= 2) {
6      if (tid < s) {
7        shared[tid] += shared[tid + s];
8      }
9      __syncthreads();
10   }
11   if (tid == 0) {
12     output[blockIdx.x] = shared[0];
13   }
14 }
```

1. Describe briefly what this function computes. Given an array of elements as input, what is the output produced?
   Solution: This kernel implements the parallel reduction algorithm to compute the sum of all elements of the array, and stores the result into `output[0]` (i.e., the first element of the output array for this block). The kernel works by repeatedly halving the active set of threads, with each active thread accumulating a partial sum into the lower half of shared memory, until a single total remains in `shared[0]`. For example, in the first iteration of the for-loop, `shared[0]` will also accumulate `shared[512]` in itself. Similarly, `shared[1]` will accumulate `shared[513]` and so on. In the next iteration, the last 512 threads are inactive, and the accumulation happens between the first 512 threads. This pattern is known as a tree reduction and completes in $\log_2(N)$ steps for an array of size N.
2. Over how many iterations does the for-loop in the kernel run?
   Solution: The loop variable s starts at `blockDim.x`$/2 = 512$ and is halved each iteration (via `s /= 2`), terminating when $s = 0$. The sequence of values taken by s is $512, 256, 128, 64, 32, 16, 8, 4, 2, 1$, giving 10 iterations in total.
3. Consider the first iteration of the for-loop for the input provided above. How many warps (of 32-consecutive threads) execute the instructions corresponding to the summation operation (`shared[tid] += shared[tid`

`+ s];`) in the loop? For each of these warps that execute, how many threads in the warp run actively in any instruction cycle, and how many are blocked due to the conditional statement?

Solution: In the first iteration, $s = 512$, so the condition `tid < 512` is satisfied by threads 0 through 511. These 512 threads span exactly 16 warps (warps 0–15, each containing 32 consecutive threads), and all 32 threads in each of these warps satisfy the condition — so there is no divergence and every thread in every active warp executes the addition. The remaining 16 warps (threads 512–1023) do not satisfy the condition and are entirely idle for this iteration.

4. Repeat the above question for the last iteration of the for-loop.

   Solution: In the last iteration, $s = 1$, so only `tid = 0` satisfies the condition `tid < 1`. Therefore, only one warp (warp 0, containing threads 0–31) is scheduled, and within that warp **only 1 thread** (thread 0) actively executes the addition while the remaining 31 threads are blocked by the conditional.

5. Now consider the following alternative kernel that attempts to compute the same result:

```
__global__ void slowKernel(int *input, int *output) {
  int tid = threadIdx.x;
  shared[tid] = input[tid];
  __syncthreads();
  for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    if ((tid % (2 * s)) == 0) {
      shared[tid] += shared[tid + s];
    }
    __syncthreads();
  }
  if (tid == 0) {
    output[blockIdx.x] = shared[0];
  }
}
```

   Does `slowKernel` produce the same result as `mysteryKernel`? Identify reasons why `slowKernel` is expected to run slower than `mysteryKernel` on a GPU, even though both kernels perform the same number of additions.

   Solution: Yes, both kernels produce the same result — the sum of all input elements stored in `output[0]`. The difference lies entirely in how the additions are scheduled across threads, not in what is computed. `slowKernel` is slower for the following reasons:

   - **Warp divergence.** The condition `(tid % (2*s)) == 0` partitions threads within the same warp into active and idle groups. For example, in the first iteration ($s = 1$), even-numbered threads (0, 2, 4, ...) are active while odd-numbered threads (1, 3, 5, ...) are idle — these are interleaved within every warp. Because a GPU warp executes in lockstep, it must serialise both paths: it first executes the addition for the active threads while masking the idle ones, then steps through the idle threads (doing nothing), effectively halving the useful throughput of every warp. In `mysteryKernel`, the condition `tid < s` ensures active threads occupy only the lowest-numbered, consecutive warps, so each warp is either fully active or fully idle — no serialisation penalty is incurred within a warp.
   - **Expensive modulo operation.** The expression `tid % (2*s)` uses integer modulo arithmetic, which

is not natively supported as a single instruction on GPU hardware and is instead emulated using a sequence of integer division, multiplication, and subtraction instructions. This adds several cycles of instruction overhead per thread per iteration, and since this expression is evaluated inside the loop by every participating thread, the cost accumulates significantly over all 10 iterations and all active threads.

6. In `mysteryKernel`, consider the very first iteration of the for-loop ($s = 512$). Exactly how many threads across the entire block are idle (i.e., do not execute the addition)? What fraction of the total thread-slots in the block does this represent, and what does this suggest about a remaining inefficiency even in `mysteryKernel`?
   Solution: With $s = 512$, all threads with `tid` ≥ 512 skip the addition, so exactly **512** threads out of 1024 are idle in the very first iteration. This means that right from the start, half the GPU's allocated compute resources for this block are doing no useful arithmetic work. This problem can be addressed by another optimization: instead of each thread loading a single element, each thread loads two elements from global memory and adds them immediately during the load phase into shared memory, effectively performing the first reduction step for free. This halves the number of blocks needed and ensures every thread is occupied from the very first instruction of the kernel.

Practice Problem 3.6

Which of the following statements is/are true for CUDA's host-locked memory allocation APIs like `cudaMallocHost()`?

(A) This memory is not allocated at page granularity, and is not managed by the OS paging system.

(B) This memory is pinned in main memory, and cannot be swapped out.

(C) This memory is mapped into a unified virtual address space between CPU and GPU, and need not be explicitly transferred between host and device.

(D) This memory is required for using the asynchronous memory copy APIs.

Solution: (B), (D)

(A) False. Host-locked memory is still allocated at page granularity and is still managed by the OS virtual memory system. The key distinction is that the OS is instructed to lock those pages (i.e. mark them as non-swappable), not to bypass the paging system entirely.

(B) True. This is the defining property of `cudaMallocHost()`. The OS pins the allocated pages in physical DRAM, preventing from ever swapping them to disk.

(C) False. This statement describes Unified Memory (`cudaMallocManaged()`), not pinned host memory. Unified Memory creates a single virtual address space accessible from both CPU and GPU, with the CUDA runtime and OS cooperating to migrate pages on demand — no explicit `cudaMemcpy` is needed.

(D) True. The asynchronous memory copy API, `cudaMemcpyAsync()`, requires the host buffer to be pinned. This is because `cudaMemcpyAsync()` returns control to the CPU immediately, before the DMA transfer has completed. If the host buffer were pageable, the OS could evict the page between the moment the CPU call returns and the moment the DMA engine actually reads that page, leading to data corruption.

Practice Problem 3.7

Consider the following CUDA kernel. Here, `z` and `data` are arrays present in the global memory. The `data` array is a 2D matrix with `nx` rows and `ny` columns laid out in row-major order.

```
1  __global__ void Kernel(uint32_t* z, const uint8_t* data,
2                         const uint32_t nx, const uint32_t ny) {
3    __shared__ uint32_t s[256];
4    const uint32_t nThreads = blockDim.x * blockDim.y;
5    const uint32_t threadId = threadIdx.y * blockDim.x + threadIdx.x;
6    for (uint32_t i = 0 ; i < 256 ; i += 1) {
7      s[i] = 0;
8    }
9    __syncthreads();
10   const uint32_t xStart = blockIdx.x * blockDim.x + threadIdx.x;
11   const uint32_t yStart = blockIdx.y * blockDim.y + threadIdx.y;
12   const uint32_t xStride = gridDim.x * blockDim.x;
13   const uint32_t yStride = gridDim.y * blockDim.y;
14   for (uint32_t y = yStart; y < ny; y += yStride) {
15     for (uint32_t x = xStart; x < nx; x += xStride) {
16       const uint8_t val = data[y * nx + x];
17       atomicAdd(&s[val], 1);
18     }
19   }
20   __syncthreads();
21   for (uint32_t i = threadId; i < 256; i += nThreads) {
22     atomicAdd(&z[i], s[i]);
23   }
24 }
```

A sample invocation of the kernel is as follows:

```
dim3 threadBlock(32, 32);
dim3 numBlocks(4, 4);
Kernel<<<numBlocks, threadBlock>>>(z, data, nx, ny);
```

(i) Explain clearly what this kernel computes. What does output `z` store by the end of the kernel execution, with respect to the input `data` array?

Solution: This kernel computes a histogram of the `data` array. Since `data` is an array of `uint8_t` values, each element lies in the range $[0, 255]$ (for example, image pixels). By the end of kernel execution, `z[i]` stores the total number of times the byte value i appears across the entire 2D `data` matrix, for each $i \in \{0, 1, \ldots, 255\}$. Each thread block accumulates a partial histogram in shared memory and then merges it atomically into the global result array `z`.

(ii) At the end of line 20, what does the shared memory array `s` have?

Solution: The shared memory array `s` holds a partial histogram local to the current thread block. Specifically, `s[i]` contains the count of how many times byte value i appeared in the portion of `data` processed by threads in this block. This partial histogram will subsequently be merged into the global array `z` in lines 21–23.

(iii) Identify any inefficiency or redundancy in the use of shared memory in this kernel. Explain the issue and write a corrected version.

Solution: The initialization loop at lines 6–8 is executed by every thread in the block independently, meaning all `nThreads` threads each write all 256 entries of `s`, resulting in `nThreads` $\times$ 256 redundant writes. Since `s` is in shared memory, only one set of writes is needed. A corrected approach distributes the initialization work evenly across threads:

```
for (uint32_t i = threadId; i < 256; i += nThreads) {
  s[i] = 0;
}
```

This ensures each entry of `s` is initialized exactly once, with the 256 entries divided among all threads in the block.

(iv) Why is an `atomicAdd` operation required when updating `s[val]` inside the nested loops in line 16? What problem would occur if it were removed? Specify the race condition clearly.

Solution: Multiple threads within the same block may read the same value `val` from different locations in `data` simultaneously, and all would attempt to increment `s[val]` at the same time. Without `atomicAdd`, two threads could both read the current value of `s[val]`, both compute `s[val] + 1`, and then both write that same incremented value back—effectively losing one of the two increments. The `atomicAdd` serialises the read-modify-write sequence for each bucket, ensuring that every increment is recorded correctly.

(v) Is the `atomicAdd` operation also necessary when updating the global memory array `z` in line 22? Justify your answer.

Solution: Yes, the `atomicAdd` on `z` is necessary. Although intra-block race conditions are avoided by assigning disjoint indices of `s` to threads within a block, threads from different blocks of the grid can and do update the same `z[i]` concurrently. Since all $4 \times 4 = 16$ blocks each accumulate a partial histogram in their own `s`, but all write to the same shared global array `z`, the atomic operation is essential to prevent lost updates across blocks.

Practice Problem 3.8

Which of the following statements is/are true for CUDA's streams API?

(A) Streams allow a better overlap of compute and data transfer between host and device.

(B) Tasks assigned to the same stream are executed concurrently, and start at the same time.

(C) Tasks assigned to the same stream are executed one after another, in the order in which they are added to the stream.

(D) Tasks assigned to different streams can execute in parallel where possible.

Solution: (A), (C), (D)

(A) True. One of the primary motivations for streams is compute-transfer overlap. Without streams, a typical program follows a strict sequence, leaving the GPU compute units idle during transfers and the DMA engine idle during kernel execution. With streams, data transfers for batch i+1 can be issued in a different stream from the kernel processing batch i.

(B) False. Operations within the same stream are strictly sequential, not concurrent. The stream is an ordered queue: the GPU will not begin operation $k+1$ in a stream until operation k has completed.

(C) True. Operations are dispatched to the GPU in FIFO order and the hardware respects that ordering: each operation in a stream begins only after all preceding operations in that same stream have finished.

(D) True. Operations residing in different streams have no ordering constraint with respect to each other (unless explicit synchronization is used). The GPU scheduler is free to overlap them whenever hardware resources permit — e.g., two independent kernels from two different streams may execute concurrently on disjoint sets of SMs, or a kernel in one stream may run while a memory transfer in another stream is in progress.

Programming Assignment 3.5: Matrix Multiplication

In this assignment, you will implement matrix multiplication in CUDA with two tasks. In the first task, you will write a basic GPU kernel where each thread computes one output element by mapping 2D thread/block indices to a matrix row and column, then loops over the inner dimension to accumulate the dot product. In the second task, you will perform matrix multiplication using the optimized `cublasSgemm` library function from NVIDIA (which requires column-major arrays). The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/matrix_multiplication

Programming Assignment 3.6: Standard Attention Using Shared Memory

In this assignment, you will implement a CUDA kernel for self-attention that uses shared memory to avoid redundant global memory reads. The kernel operates on small problems where both sequence length and head dimension are at most 32, with one CUDA block per (batch, head, query token) triplet and one thread per feature dimension. The implementation follows four steps: each thread computes one scaled dot-product attention score between the query row and a key row, storing results in shared memory; thread 0 then serially runs softmax over those scores (finding the max, exponentiating, normalizing); and finally all threads in parallel accumulate the weighted sum over value rows to produce the output. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/standard_attention_shared_memory

Programming Assignment 3.7: Matrix Transpose Using Shared Memory

In this assignment, you will implement a CUDA kernel to transpose a square matrix efficiently using shared memory tiling. Rather than naively reading or writing to global memory in an uncoalesced pattern, the approach breaks the matrix into 32×32 tiles — each thread block loads one tile into shared memory in a coalesced row-major fashion, synchronizes, and then writes the transposed tile back to global memory, also coalesced. You implement two functions: the kernel which does the actual tiling and transposition, and a wrapper which handles all the boilerplate (memory allocation, data transfer, kernel launch, and cleanup). The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/matrix_transpose_shared_memory

Programming Assignment 3.8: MLP Forward Pass

In this assignment, you will implement the forward pass of a two-layer Multi-Layer Perceptron. Given an input batch matrix X and two sets of weights and biases, you must write two CUDA kernels — a `matmul_kernel` that computes matrix multiplication with bias addition (where each thread owns exactly one output element) and a `relu_kernel` that clamps negative values to zero element-wise — then wire both together inside a launch wrapper that orchestrates the full computation. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/mlp_forward_pass

Programming Assignment 3.9: MLP Forward Pass with Streams

This is pretty similar to the previous programming assignment. However, the kernels are orchestrated inside a launch wrapper using the CUDA Streams API, where you must allocate pinned host memory via `cudaMallocHost`, asynchronously transfer all weights and inputs to the device with `cudaMemcpyAsync`, launch both kernels sequentially into the same stream, copy the result back asynchronously, and finally synchronize and destroy the stream. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/mlp_forward_pass_streams

3.5 CUDA Compilation Workflow

CUDA applications contain both CPU code and GPU code within the same source file. Since these two components target fundamentally different architectures, the CUDA compilation process must separate, transform, compile, assemble, and finally reconnect these components into a single executable program. NVIDIA provides a specialized compiler driver called `nvcc` (NVIDIA CUDA Compiler) to manage this process. `nvcc` is not a single monolithic compiler. Instead, it acts as a driver that invokes multiple internal tools, including the host C++ compiler, CUDA front-end parsers, PTX generators, assemblers, linkers, and device code generators. A CUDA source file typically uses the `.cu` extension. A basic CUDA compilation command appears as follows:

```
nvcc vectorAdd.cu -o vectorAdd
```

This command compiles the CUDA source file and produces an executable named `vectorAdd`. Next, let us go through the sequence of transformations performed by `nvcc` while converting CUDA source code into an executable program. Conceptually, the process proceeds through the following stages.

Preprocessing and Source Separation

The first phase resembles ordinary C/C++ preprocessing. Macros are expanded, header files are included, and conditional compilation directives are processed. After preprocessing, `nvcc` separates the CUDA source into two components: host code and device code. Host code consists of ordinary C++ instructions intended for execution on the CPU. Device code consists of GPU kernels and device functions intended for execution on NVIDIA GPUs.

Host Code Compilation

After separation, host code is transformed into ordinary C++ code. CUDA-specific constructs such as kernel launches are converted into runtime API calls that configure and launch GPU kernels dynamically. The transformed host code is then forwarded to a standard host compiler such as `g++` or `clang++` depending on platform and configuration. Thus, `nvcc` does not replace the ordinary CPU compiler. Instead, it cooperates with the system compiler. The host compiler ultimately produces a host-side object file (`.o`), which will later be linked together with the generated device code.

Device Compilation and PTX Generation

The device portion of the source file undergoes a completely different compilation trajectory. CUDA device code is first compiled into an intermediate representation called PTX or Parallel Thread Execution, which is a virtual instruction set architecture for NVIDIA GPUs. It acts as a low-level assembly-like intermediate representation that remains independent of specific GPU generations. A PTX file typically possesses the extension `.ptx`. It is human-readable and resembles assembly language.

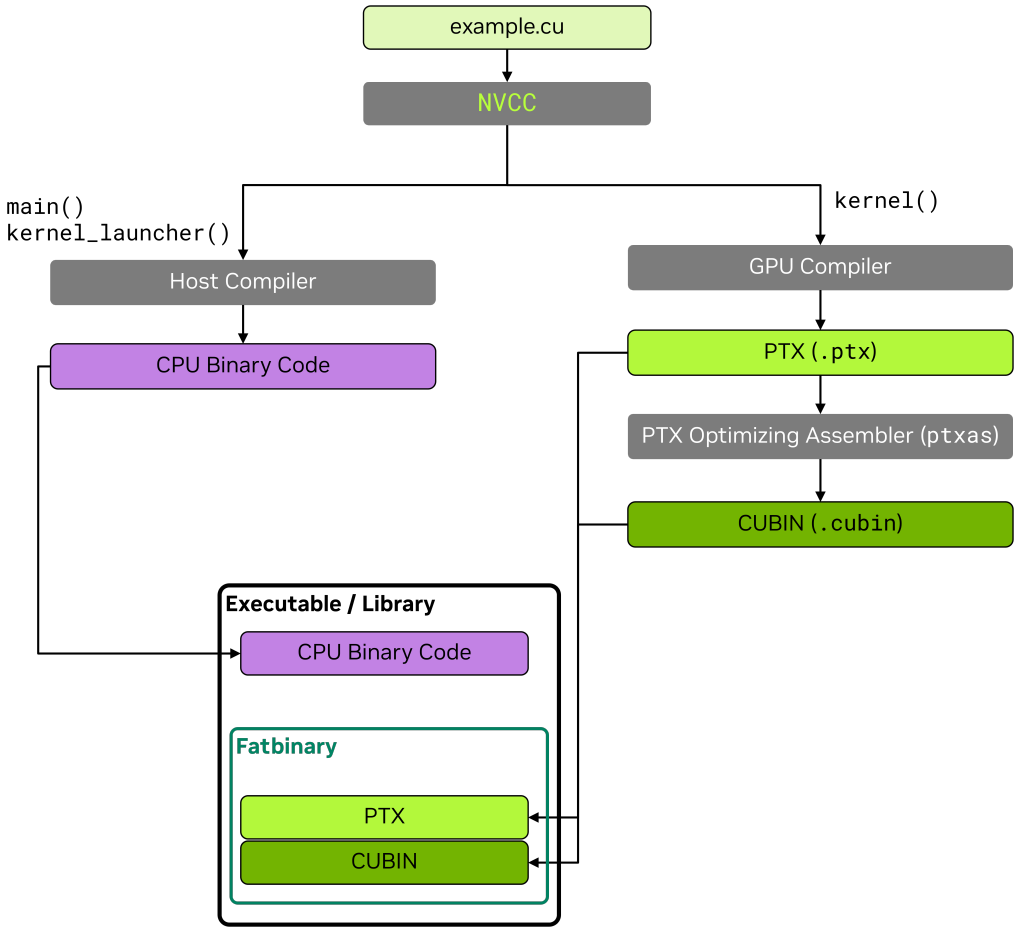


Figure 3.11: NVCC compilation (image credit: NVIDIA, 2025)

CUBIN Generation

Although PTX is portable, GPUs ultimately execute hardware-specific binary instructions. The CUDA assembler converts PTX into GPU-specific binary machine code called a CUBIN file. A CUBIN file contains native instructions for a specific GPU architecture such as `sm_70`, `sm_80`, or `sm_90`. Here, `sm` stands for streaming multiprocessor architecture, and the numeric suffix indicates compute capability generation.

Fatbinary Generation

Modern CUDA executables frequently support multiple GPU architectures simultaneously. CUDA therefore packages multiple device representations together into a **fatbinary**. A fatbinary may con-

tain PTX code and multiple cubins for different architectures. At runtime, the CUDA driver selects the most appropriate device binary for the installed GPU. If no matching CUBIN exists, the driver may JIT-compile PTX into native instructions dynamically.

Just-In-Time (JIT) Compilation

When PTX rather than native CUBIN code is available for a GPU, the CUDA driver performs Just-In-Time compilation at runtime. The PTX code is translated into architecture-specific machine instructions for the installed GPU. JIT compilation enables forward compatibility because applications compiled years earlier may still execute on newer GPUs. However, JIT introduces startup overhead because compilation occurs during program execution. CUDA therefore caches generated binaries internally to reduce repeated compilation costs.

Intermediate Files Generated During Compilation

CUDA compilation may generate numerous intermediate files depending on compiler options.

- `.cpp1.ii`: Preprocessed CUDA files
- `.cudafe1.cpp`: Transformed host-side source
- `.ptx`: PTX intermediate code
- `.cubin`: Native GPU binary code
- `.fatbin`: Embedded multi-architecture GPU binary
- `.o`/`.obj`: Traditional compiled object files

Figure 3.11 shows the entire compilation process.

3.6 Mapping Software Abstractions to Hardware

As we saw before, the CUDA programming model presents the programmer with a hierarchy of abstractions consisting of grids, blocks, and threads (Figure 3.7). Although these abstractions appear purely software-defined, they are designed to mirror the physical organization of modern GPUs. Understanding the mapping between these abstractions and hardware components is essential for a GPU programmer.

The highest-level execution abstraction in CUDA is the grid. A grid represents the complete set of work associated with a kernel launch. Unlike lower-level abstractions such as warps or blocks, the grid does not correspond to a dedicated hardware structure. Instead, the grid holds all metadata describing the kernel launch. The runtime stores information about the number of blocks, block dimensions, resource requirements, and scheduling state. When a kernel launches, the GPU does not instantiate all thread blocks simultaneously. Instead, the hardware progressively distributes thread blocks from the grid onto available SMs. The grid therefore behaves like a global work queue from which blocks are dynamically assigned to hardware resources. Note that some blocks of a grid may begin execution much later than others depending on resource availability and scheduling decisions. Figure 3.12 shows an example of how thread blocks of a grid are distributed to available SMs automatically.

The thread block is the primary unit of scheduling at the hardware level. Each block is assigned to exactly one Streaming Multiprocessor for the duration of its execution. Once assigned, the block never migrates to another SM. This is essential, because both shared memory and synchronization (say, using `__syncthreads()`) rely on the assumption that all threads of the block execute within the same SM. The mapping relation between thread blocks and SMs is therefore many-to-one, i.e., multiple thread blocks may reside simultaneously on an SM, but each individual block belongs entirely to one SM. The number of simultaneously resident blocks depends on available hardware resources. Every block consumes registers, shared memory, scheduler entries, and thread capacity. A block may only be admitted to an SM only if sufficient resources exist for the entire block.

When a block is assigned to an SM, the hardware partitions its threads into warps. For example, a block containing 256 threads is divided into eight warps. These warps become independently schedulable entities, and are scheduled by the warp scheduler. A single instruction fetched by the scheduler is executed simultaneously by all active threads in the warp. As we saw in Figure 3.10a, warps are formed with groups of 32 threads with consecutive thread indices.

Table 3.5 summarizes this mapping between software abstractions and hardware components.

Software	Hardware
Grid (all blocks from one kernel launch)	Entire GPU (distributed across all SMs)
Thread block (CTA)	Streaming multiprocessor (always on one SM, never splits)
Warp (32 threads)	Warp scheduler (issues one instruction for all 32 threads)
Thread (smallest execution unit)	Instructions issued to CUDA/tensor cores

Table 3.5: GPU software to hardware mapping

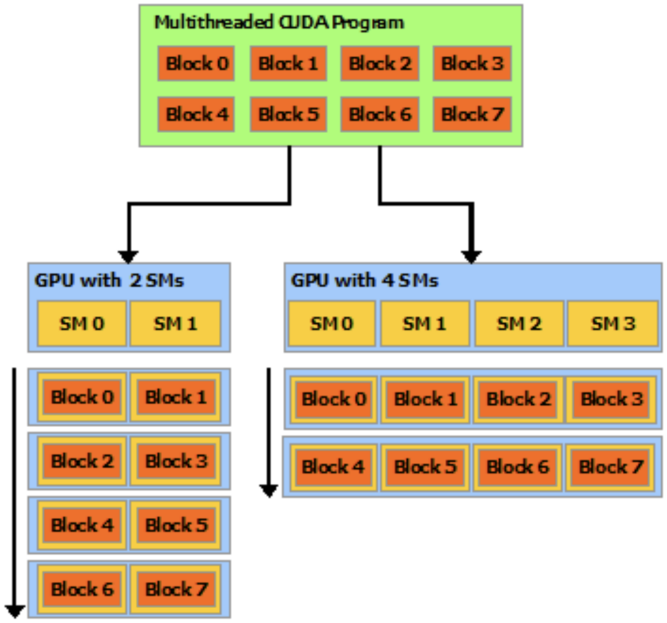


Figure 3.12: Automatic scalability (Image credit: NVIDIA, 2025)

3.6.1 Thread Block Assignment to Streaming Multiprocessors

When a kernel launches, the GPU gets a pool of blocks, which it must assign to SMs. When launched, the blocks begin in a pending state. As soon as an SM has sufficient free resources, the scheduler dispatches a block from the grid for execution. This assignment process is dynamic rather than static, i.e., blocks are not permanently pre-assigned to specific SMs before execution begins. Instead, scheduling decisions occur incrementally as blocks complete and resources become available. The precise scheduling implementation is hardware-dependent and largely undocumented. Nevertheless, some general principles govern block placement. For example, the runtime attempts to maximize utilization by ensuring that SMs remain busy whenever possible. Two commonly discussed strategies

in the literature for assigning blocks to SMs are most-room placement and least-room placement, which are described below. Actual implementations are significantly more sophisticated and may combine multiple heuristics simultaneously.

Most-Room Policy

Under the most-room scheduling policy, incoming thread blocks are assigned to the SM possessing the largest amount of remaining free resources. The objective is to balance compute across the device and minimize fragmentation. Consider a GPU where different SMs possess varying amounts of remaining shared memory or register capacity due to prior scheduling decisions. A most-room policy attempts to spread incoming work evenly so that future blocks have the highest probability of fitting successfully. This strategy often improves long-term utilization.

Least-Room Policy

Under the least-room policy, the scheduler instead packs blocks into already occupied SMs whenever possible. Rather than distributing work evenly, the scheduler attempts to fill partially occupied SMs before utilizing emptier ones. This policy may improve locality and can also reduce power consumption by concentrating activity on fewer SMs.

SM Occupancy

The occupancy of an SM refers to the fraction of maximum warp (thread) capacity currently occupied by active or resident warps (threads). It is formally defined as:

$$\text{Occupancy} = \frac{\text{Active Warps}}{\text{Maximum Warps per SM}}$$

Occupancy is important because GPUs rely heavily on thread-level parallelism to hide latency. When one warp stalls waiting for memory or instruction dependencies the scheduler can issue instructions from another ready warp. However, occupancy is not equivalent to utilization. A kernel may achieve high occupancy while still underutilizing arithmetic pipelines due to memory bottlenecks or divergence. Conversely, lower occupancy kernels may perform extremely well if they exhibit stronger parallelism and efficient memory behaviour.

Occupancy is constrained by several independent hardware limits simultaneously. Each streaming multiprocessor contains finite hardware resources. These include the register memory, shared memory and maximum resident thread count. When a block is admitted onto an SM, resources are allocated statically for the lifetime of that block. without any oversubscription. Registers are reserved for every thread, and shared memory allocations are reserved for the entire block. Once one or more of the resources of an SM are exhausted, one cannot assign any more blocks to the SM. The effective occupancy of a kernel is determined by whichever resource becomes exhausted first.

The first major hardware limit is thread capacity. Every SM suports only a finite number of resident threads. If an SM supports 2048 threads and blocks contain 1024 threads, at most two

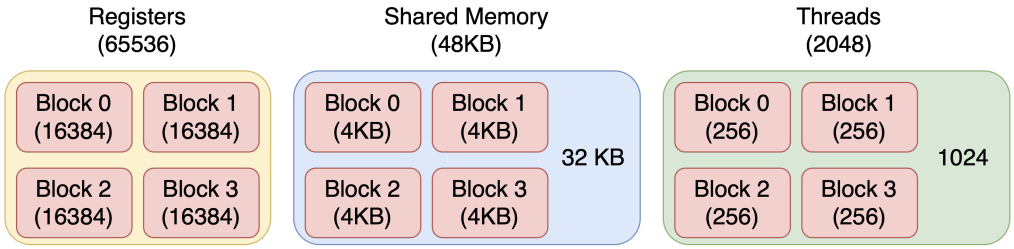


Figure 3.13: Resource constraints in an SM

blocks may coexist. The second limit is register file capacity. Suppose an SM contains 65536 registers and a kernel uses 64 registers per thread. A block containing 256 threads therefore requires $256 \times 64 = 16384$ registers. The maximum number of such blocks that can reside simultaneously is therefore $\left\lfloor \frac{65536}{16384} \right\rfloor = 4$, even if other hardware limits theoretically permit more blocks. Note that registers are allocated statically per thread. High register usage directly reduces the number of resident threads and warps. The third limit is shared memory capacity. Shared memory allocations occur per block. Large shared memory tiles therefore reduce the number of simultaneously resident blocks, and hence impact occupancy.

Consider the example shown in Figure 3.13. Each SM has 65,536 registers, 48 KB of shared memory, and supports a maximum of 2,048 concurrent threads. We launch a kernel in which each thread block requires 16,384 registers, 4 KB of shared memory, and contains 256 threads. To determine how many thread blocks can reside simultaneously on a single SM, we evaluate the constraint imposed by each resource: (i) registers — each SM has 65,536 registers and each block requires 16,384, allowing $\lfloor 65536/16384 \rfloor = 4$ blocks; (ii) shared memory — each SM has 48 KB and each block requires 4 KB, allowing $\lfloor 48/4 \rfloor = 12$ blocks; and (iii) threads — each SM supports at most 2,048 concurrent threads and each block contains 256 threads, allowing $\lfloor 2048/256 \rfloor = 8$ blocks. The maximum number of resident thread blocks per SM is therefore $\min(4, 12, 8) = 4$, with the register file acting as the binding constraint. With 4 resident blocks of 256 threads each, the total active thread count is $4 \times 256 = 1{,}024$, yielding $1024/32 = 32$ active warps. Given that the SM can support at most 2048 threads, or $2048/32 = 64$ warps, the SM occupancy is thus $32/64 = 0.5$, or 50%.

Wave Quantization and Tail Effects

Wave quantization is an important performance phenomenon arising from finite SM resources and discrete scheduling granularity. Suppose a GPU contains 80 SMs and a particular kernel configuration allows 4 resident blocks per SM. The device can therefore execute $80 \times 4 = 320$ blocks concurrently. If the grid contains exactly 320 blocks, the entire kernel executes in one wave and all blocks become resident simultaneously. However, if the grid contains 321 blocks, the final block cannot begin until one earlier block finishes. Therefore, execution requires two waves (Figure 3.14). The

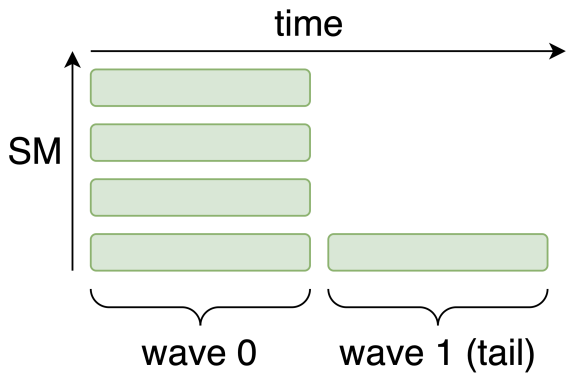


Figure 3.14: There are 5 thread blocks, and 4 SMs. In wave-0, each SM gets one block. However, in wave-1, only one SM gets a block, and all others remain idle. This is a simple example of wave quantization.

second wave contains only one block, meaning that only one SM remains active while the remaining 79 SMs sit idle. This creates a sharp utilization drop. Wave quantization causes performance discontinuities where tiny changes in grid size produce disproportionately large runtime differences. This issue is particularly common in practice because many workloads launch power-of-two numbers of blocks, whereas modern GPUs often contain irregular SM counts such as 108 or 132 SMs, making perfect wave alignment relatively uncommon.

Even when multiple blocks exist in the final wave, utilization often continues to decrease due to tail effects. Blocks rarely complete at exactly the same time because of differences in memory latency, cache behaviour, synchronization overhead, and control-flow divergence. Since thread blocks cannot migrate between SMs after assignment, some SMs finish their assigned work earlier and become idle while others continue executing longer-running blocks. As execution approaches completion, the remaining work gradually becomes concentrated on fewer SMs, reducing parallel efficiency. Tail effects therefore amplify the utilization loss already introduced by wave quantization, particularly for small grids or workloads with highly imbalanced block execution times.

3.6.2 Warp Scheduling

Each SM contains one or more warp schedulers responsible for selecting ready warps for instruction issue. Warp scheduling is central to GPU throughput because schedulers determine how effectively execution pipelines remain utilized. At every cycle, the scheduler examines resident warps and identifies those eligible for execution. A warp becomes ineligible if it is stalled due to memory dependencies, synchronization barriers, or unavailable functional units. A warp basically cannot issue instructions whose operands are not yet ready. Schedulers interact closely with the SM scoreboard mechanism, which tracks pending operations and register dependencies, and then selects one or more

eligible warps and issues instructions into execution pipelines.

Warp Context Switching

Unlike CPU context switches, warp switches are effectively free because all warp state remains resident on-chip. Every active warp has its own register allocation and execution state stored directly inside the SM. When one warp stalls, the scheduler simply selects another ready warp during the next cycle. This hardware multithreading model enables GPUs to tolerate extremely long memory latencies efficiently, and the effectiveness of latency hiding depends on maintaining enough independent warps. If too few warps are resident, memory stalls propagate directly into idle execution cycles.

Warp Scheduling Policies

Different GPU generations employ different warp scheduling policies. Here are a few of them:

- **Round Robin (RR):** Under the Round Robin policy, the scheduler selects eligible warps cyclically in a fixed rotating order. This approach is simple and ensures that all active warps receive regular execution opportunities, preventing starvation.
- **Least Recently Fetched (LRF):** This policy prioritizes the warp whose instruction has not been issued for the longest time. By favouring neglected warps, LRF attempts to improve fairness while also helping stalled warps make progress once dependencies are resolved.
- **Fair (FAIR):** This policy attempts to equalize the number of instructions issued across all resident warps. Warps that received fewer instruction issues are given higher priority so that execution opportunities remain balanced among threads.
- **Criticality Aware Warp Scheduling (CAWS):** It allocates more scheduling priority to warps that are expected to lie on the critical execution path. By accelerating long-latency or performance-critical warps, CAWS attempts to reduce overall kernel completion time.
- **Greedy-Then-Oldest (GTO):** In this policy, the scheduler continues issuing instructions from the same warp as long as it remains ready for execution. When that warp stalls, the scheduler switches to the oldest waiting eligible warp.

Early GPUs used relatively simple round-robin scheduling strategies. More recent architectures implement more sophisticated heuristics balancing fairness, locality, dependency resolution, and execution throughput.

Practice Problem 3.9

Suppose an SM on a GPU has C cores and N registers. Suppose a user program creates a large number of threads, and every thread needs to access K distinct registers during its execution. Assuming all threads are ready to run, how many threads can execute on the GPU concurrently? Select the most appropriate answer.

(A) C

(B) $\frac{N}{K}$

(C) $\min(C, \frac{N}{K})$

(D) $\max(C, \frac{N}{K})$

Solution: (C)

A thread can execute only if two resources are available: a core to run on and K registers for its private state. Since the SM has C cores, at most C threads can execute simultaneously. On the other hand, with N total registers and each thread requiring K registers, the register file can support at most $\frac{N}{K}$ concurrent threads. Therefore, the actual number of threads that can execute concurrently is limited by whichever resource runs out first. For example, if $C = 128$, $N = 65536$, and $K = 256$, then the registers can support $\frac{65536}{256} = 256$ threads, but only 128 cores are available, so only 128 threads can execute concurrently. Conversely, if $K = 1024$, then only $\frac{65536}{1024} = 64$ threads can be supported by the registers, even though 128 cores exist.

Practice Problem 3.10

Consider a GPU SM that has 128 KB shared memory, 64,000 registers, and 2,000 cores. The maximum blocks per SM is 32, and maximum threads per block is 1024. Given below are specifications of blocks launched on the SM. Calculate the expected SM occupancy in each case, assuming the SM is only running blocks of that type.

(i) Blocks with 32 threads each, using 256 registers per block.

Solution: SM occupancy is the fraction of the SM's execution capacity that is occupied by active threads. Since the SM has 2000 cores, we compute occupancy as

$$\text{Occupancy} = \frac{\text{Number of active threads on the SM}}{2000}.$$

The SM can support at most 32 blocks. Each block contains 32 threads, so the maximum number of active threads is $32 \times 32 = 1024$. The total register requirement is $32 \times 256 = 8192$, which is much smaller than the available 64,000 registers. Hence, the block limit of 32 is reached before any resource limit is encountered. Therefore, Occupancy = $\frac{1024}{2000} = 0.512$.

(ii) Blocks with 1024 threads each, using 1024 registers per block, and 96KB shared memory per block.

Solution: The shared-memory requirement limits the number of resident blocks to $\left\lfloor \frac{128 \text{ KB}}{96 \text{ KB}} \right\rfloor = 1$. Thus, only one block can reside on the SM at a time. Since each block contains 1024 threads, the number of active threads is $1 \times 1024 = 1024$. The register requirement is only 1024, which is well below the available 64,000 registers. Hence, shared memory is the limiting resource. Therefore, Occupancy = $\frac{1024}{2000} = 0.512$.

Practice Problem 3.11

Consider a GPU SM that has 256 KB shared memory, 1024 registers and 2048 cores. The maximum blocks per SM is 32, and maximum threads per block is 1024. Assuming that the SM is only running blocks of one type, calculate the expected SM occupancy when we have blocks with 256 threads each, using 512 registers and 64 KB shared memory per block.

Solution: To determine occupancy, we first identify how many blocks can simultaneously reside on the SM based on each resource constraint.

- Shared memory limit:
$$\frac{256\ \text{KB}}{64\ \text{KB}} = 4\ \text{blocks}.$$
- Register limit:
$$\frac{1024}{512} = 2\ \text{blocks}.$$
- Thread limit:
$$\frac{2048}{256} = 8\ \text{blocks}.$$
- Maximum blocks per SM:
$$32\ \text{blocks}.$$

The number of resident blocks is determined by the most restrictive resource. Since the register file allows only 2 blocks, registers are the binding constraint.

Therefore, the number of active threads on the SM is $2 \times 256 = 512$. Since the SM can support at most 2048 threads, the occupancy is $\frac{512}{2048} = 0.25$.

Practice Problem 3.12

Which entity schedules warps on to SM cores?

(A) OS scheduler running on the CPU

(B) Software scheduler running on the GPU

(C) Hardware scheduler on the GPU

(D) Control logic written by user in CUDA

Solution: (C)

As we discussed above, warp scheduling is performed by a dedicated hardware scheduler within each Streaming Multiprocessor (SM). The scheduler dynamically selects ready warps and issues instructions to the SM cores, enabling efficient latency hiding without intervention from the CPU, software, or user-written CUDA code.

Practice Problem 3.13

Block-to-SM mapping proceeds in waves. In the initial waves, we have enough blocks to fill all SMs to capacity (until some resource runs out). In the final wave, we may have fewer thread blocks than what is needed to occupy all SMs, leading to some wave quantization tail effects. Consider the following two block-to-SM scheduling policies that differ in how they handle the scheduling of blocks in the last wave. In policy A, we try to pack all remaining thread blocks on as few SMs as possible, while leaving the rest empty. In policy B, we distribute the remaining thread blocks as evenly as possible across all SMs.

(i) Consider a scenario where we have a kernel with g thread blocks (TBs), N streaming multiprocessors (SMs), and we can run b TBs per SM. For policy A, how many waves w will be required in total (express in terms of g, N, b)?

Solution: Each full wave schedules exactly $N \times b$ thread blocks — one batch of b blocks on every one of the N SMs. After enough full waves, any leftover blocks form a final (tail) wave. Even if only a single

block remains, it still constitutes its own wave. Formally, the total number of waves is the smallest integer w such that $w \cdot N \cdot b \geq g$, i.e.,

$$w = \left\lceil \frac{g}{N \cdot b} \right\rceil.$$

(ii) For policy A, how many SMs will be active in the tail wave (express in terms of g, N, b, w)?
Solution: After $w-1$ complete waves, the number of blocks already scheduled is $(w-1) \cdot N \cdot b$. The number of blocks remaining for the tail wave is therefore $r = g - (w-1) \cdot N \cdot b$. Note that $1 \leq r \leq N \cdot b$ (there is at least one block left, and at most a full wave's worth). Policy A packs greedily: each active SM is filled to its maximum capacity of b blocks before the next SM is used. So the number of SMs needed to handle r remaining blocks, each running up to b blocks, is

$$\text{Active SMs} = \left\lceil \frac{r}{b} \right\rceil = \left\lceil \frac{g - (w-1) \cdot N \cdot b}{b} \right\rceil.$$

(iii) Continuing with the previous part, for policy B, how many waves w will be required in total (express in terms of g, N, b)?
Solution: The number of waves is determined solely by the total work g and the per-wave capacity $N \cdot b$. Spreading the tail-wave blocks across more SMs does not reduce or increase the number of waves — it only changes which SMs are active during the last wave. Therefore, the wave count is identical to Policy A:

$$w = \left\lceil \frac{g}{N \cdot b} \right\rceil.$$

(iv) For policy B, how many SMs will be active in the tail wave (express in terms of g, N, b, w)?
Solution: As before, after $w-1$ full waves we have $r = g - (w-1) \cdot N \cdot b$ blocks remaining. Policy B spreads evenly: every SM gets at most one "slot" before any SM gets a second one. If $r \geq N$, all N SMs receive at least one block. If $r < N$, only r SMs can receive even a single block. Therefore, the number of active SMs in the tail wave is

$$\text{Active SMs} = \min(N, r).$$

(v) For policy B, in the final wave, what will be the maximum number of blocks being run on any single SM?
Solution: From part (iv), $\min(N, r)$ SMs are active. Policy B distributes the r remaining blocks as evenly as possible across those $\min(N, r)$ SMs, so the most heavily loaded SM runs

$$\left\lceil \frac{r}{\min(N, r)} \right\rceil \text{ blocks.}$$

We can simplify by considering the two cases separately:

- **Case 1**: $r < N$.
  Only r SMs are active, each receiving exactly 1 block (since r blocks spread over r SMs gives $\lceil r/r \rceil = 1$). Every active SM runs exactly one block.
- **Case 2**: $r \geq N$.
  All N SMs are active. The maximum load on any SM is

  $$\left\lceil \frac{r}{N} \right\rceil.$$

  For example, if $r = 7$ and $N = 3$, then $\lceil 7/3 \rceil = 3$, so one SM gets 3 blocks and the others get 2.

3.7 Tensor Processing Units (TPUs)

As we saw before, training and inference for neural networks involve repeated execution of operations such as matrix multiplication, convolution, and vector addition. These operations are executed across billions or trillions of parameters and often over enormous datasets. Conventional CPUs were never designed for this scale of arithmetic throughput. GPUs improved performance significantly by exploiting data parallelism, but retained substantial power overhead. Google introduced the Tensor Processing Unit (TPU) to address these limitations. TPUs are domain-specific accelerators designed specifically for machine learning workloads. Rather than optimizing for general-purpose programmability, TPUs optimize for tensor algebra execution at large scale with high energy efficiency.

3.7.1 TPU Architecture

Before understanding how a TPU performs tensor operations, let us first understand what are the fundamental components of a TPU (Figure 3.15). The compute core of the TPU is also called TensorCore, and is attached to a stack of fast memory, also called HBM (High Bandwidth Memory). TensorCore acts as a highly optimized matrix-processing engine. Internally, it contains several compute, memory, and control components:

- **Matrix Multiply Unit (MXU):** It is the core component of the TensorCore. For most TPU generations, it performs one matrix multiply (between 8×128 and 128×128 matrices) every few cycles using a systolic array. Most TensorCores have 2 or 4 MXUs.
- **Vector Processing Unit (VPU):** It performs general matrix operations like ReLU activations, pointwise addition/multiplication between vectors, and reductions.
- **Vector Memory (VMEM):** It is an on-chip programmer-controlled memory located in the TensorCore, close to the compute units. It is much smaller in size as compared to the HBM, but offers a higher bandwidth to the MXU. For example, for TPU v5e, the VMEM is of size 128 MB. Data in HBM needs to be copied into VMEM before the TensorCore can do any computation with it.
- **Registers / Register Files**: Registers are extremely small and extremely fast storage locations located directly inside the compute pipeline. They hold temporary operands, partial sums, addresses, loop counters, and intermediate values during execution. The MXU and VPU continuously read from and write to register files while performing computations.
- **Scalar Unit (SU):** The scalar unit executes non-vector control operations such as loop management, address calculations, branching, instruction dispatch, and synchronization. While the MXU and VPU handle large tensor computations, the scalar unit orchestrates the overall execution flow.
- **Scalar Memory (SMEM):** SMEM is a small, low-latency on-chip memory attached to the Scalar Unit (SU). It stores scalar operands, loop counters, address metadata, synchronization state, and other control-related information required for orchestrating tensor execution.

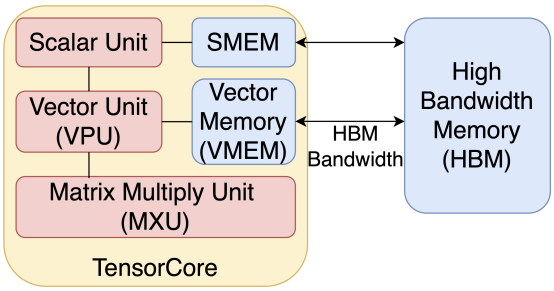


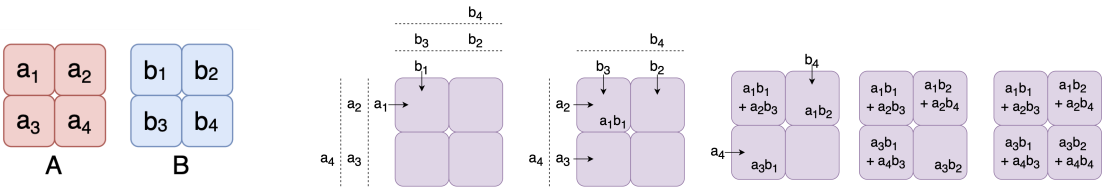
Figure 3.15: TPU architecture

- **Interconnect / Network-on-Chip (NoC):** A high-bandwidth on-chip communication network connects MXUs, VPUs, VMEM, scalar units, and memory controllers. In multi-chip TPU systems, dedicated inter-chip interconnects are also used for collective communication between TPUs.
- **Load/Store Units and DMA Engines:** These units are responsible for moving data between HBM, VMEM, and the compute units. DMA (Direct Memory Access) engines perform asynchronous tensor transfers so that data movement can overlap with computation, improving utilization of the MXU.

HBM (High Bandwidth Memory) is a large chunk of fast memory that store tensors for use by the TensorCore. It usually has capacity on the order of tens of gigabytes. For example, TPU v5e has an HBM of size 16 GB. When needed for a computation, tensors are streamed out of HBM through VMEM into the MXU and the result is written from VMEM back to HBM. The bandwidth between HBM and the TensorCore, (through VMEM) is typically called as HBM-bandwidth. It is usually arround 1–2 TB/s, and becomes pretty relevant for memory-bound workloads.

Generally, all TPU operations are pipelined and overlapped. To perform a matrix multiplication operation ($C = A \times B$), a TPU would need to first copy chunks of matrices A and B into Vector Memory (VMEM), then load them into MXU (chunks of size 8×128 for A and 128×128 for B). The result chunk is copied back to HBM. To do this efficiently, the compute is pipelined so that the copies to/from VMEM are overlapped with MXU work. This allows the MXU to continue working instead of waiting for memory transfers to complete.

A TPU chip typically (but not always) consists of two TPU cores which share memory and can be thought of as one larger accelerator with twice the FLOPs. Chips are arranged in sets of four on a tray connected to a CPU host via PCIe network. Similar to the case with GPUs, PCIe bandwidth is typically 20–100x slower than the HBM bandwidth.



(a) Input matrices

(b) Output matrix computation

Figure 3.16: Systolic array steps

3.7.2 Systolic Array

At the core of the TPU MXU is a hardware structure called systolic array which is can perform efficient multiplication every few cycles. For example, in TPU v5e, it is a 2D grid of $128 \times 128 = 16384$ ALUs, each capable of performing a multiply and add operation. These ALUs are also called processing elements. Weights are passed from above, while the inputs are passed from the left. To understand the working in detail, let us consider two matrices, of size 2×2 each, and the corresponding steps involved in their matrix multiplication by a systolic array (Figure 3.16).

- Rows of A enter from left, staggered by one clock cycle (a_1 enters first, a_3 one step later). Columns of B enter from the top, also staggered (b_1 first, b_2 one step later). The staggering is deliberate, it ensures each A element meets its correct B element at the right processing element (PE) at the right time.
- At clock tick 1, a_1 slides in from the left, and b_1 slides in from the top, they meet at PE(0,0). That PE multiplies them and stores $a_1 \times b_1$. All other PEs are idle because their data hasn't arrived yet.
- At tick 2, PE(0, 0) receives a_2 and b_3, adds $a_2 \times b_3$ to its running sum, and is now done. Meanwhile, a_1 has passed rightward to PE(0, 1) which meets b_2, and a_3 has passed downward to PE(1, 0) which meets b_1. Multiple PEs are now working simultaneously.
- By tick 4, every PE has received all the pairs it needs and finished accumulating. Each PE holds exactly one cell of the output matrix C. No communication between PEs was needed and the data just flowed through.

The key idea behind the systolic array is that data movement is minimized. Instead of repeatedly fetching operands from large memories like HBM, each value is reused locally as it propagates across neighboring PEs. This dramatically reduces memory bandwidth requirements and energy consumption, since moving data through memory hierarchies is often far more expensive than performing arithmetic itself. Once the pipeline is full, every PE performs a multiply-accumulate (MAC) operation every clock cycle, leading to extremely high utilization and throughput.

Another important property of systolic arrays is that computation and communication occur

simultaneously. Each PE performs three operations in lockstep: (i) receiving operands from neighbors, (ii) performing a multiply-add, and (iii) forwarding operands to adjacent PEs. For large matrix multiplications, matrices are divided into smaller tiles that fit into the systolic array dimensions. The MXU repeatedly streams these tiles through the array while partial sums are accumulated in high-speed accumulator buffers. Because thousands of MAC operations occur in parallel, TPUs achieve extremely high throughput for deep learning workloads such as transformer training and inference.

3.7.3 TPU Networking

Even though a TPU chip contains extremely powerful matrix multiplication hardware and high-bandwidth memory, the memory capacity and compute throughput of one chip are insufficient for training frontier-scale neural networks. As a result, modern TPU systems are designed as distributed computing systems in which thousands of TPU chips cooperate during training and inference. TPU networking is the mechanism that enables these chips to exchange tensors, gradients, activations, and parameters efficiently. Moreover, the networking philosophy of TPUs differs fundamentally from that of GPUs. As we saw before, GPU systems typically attempt to approximate full connectivity between accelerators using technologies such as NVLink, NVSwitch, and InfiniBand. These minimize hop counts and support highly flexible communication patterns, but they are expensive and difficult to scale efficiently. TPUs instead rely on nearest-neighbour communication. The hardware is simpler, cheaper, and easier to scale.

To understand TPU networking, consider a simple matrix multiplication distributed across multiple TPU chips. Suppose a large matrix is partitioned across several chips because it cannot fit into the memory of a single TPU. During computation, each chip may require tensor fragments stored on neighbouring chips. After the matrix multiplication is completed, partial outputs may also need to be aggregated across chips. This means that computation alone is not sufficient, tensor movement becomes equally important. TPU systems use two fundamentally different communication networks. The first network is called the Inter-Chip Interconnect (ICI), while the second is the Datacenter Network (DCN).

Inter-Chip Interconnect (ICI)

A collection of TPU chips connected through ICI is called a TPU pod. Each TPU chip contains dedicated high-speed communication links that connect it directly to nearby TPU chips. These links do not pass through PCIe or CPUs. Intead, TPU chips communicate directly through specialized networking hardware integrated into the TPU package itself. Rather than attempting to connect every chip directly to every other chip, TPUs only connect each chip to a small set of adjacent neighbours. Communication over long distances is achieved by forwarding packets through intermediate TPU chips. This design significantly reduces hardware complexity and power consumption. More importantly, it allows TPU systems to scale to thousands of chips while maintaining relatively constant communication bandwidth per chip.

Older TPU generations such as TPU v2 and TPU v3, as well as inference-oriented systems like TPU v5e and TPU v6e, use a two-dimensional torus topology. In this topology, each TPU chip is connected to four neighbouring chips: north, south, east, and west. As shown in Figure 3.17, the topology resembles a two-dimensional grid. However, the edges of the grid are connected together to form a torus, i.e., the left edge wraps around to the right edge, and the top edge wraps around to the bottom edge. For example in Figure 3.17, TPU A is connected to TPU I, and TPU A is also connected to TPU D. Without wraparound connections, a tensor traveling from one edge of the grid to the opposite edge would need to traverse the entire dimension length. With toroidal wraparound, the maximum distance between any two TPUs becomes approximately half the dimension size. This significantly lowers communication latency and improves collective operation performance. For TPU v5e and TPU v6e systems, the maximum torus size is typically a 16×16 topology, corresponding to 256 TPU chips.

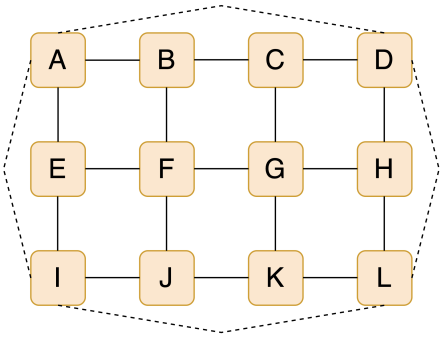


Figure 3.17: Two-dimensional torus topology (with only a few wraparound links shown)

Later TPU generations such as TPU v4 and TPU v5p extend this idea further by using a three-dimensional torus topology. In these systems, each TPU chip connects to six neighbouring chips instead of four. The chip communicates along three independent dimensions: X, Y, and Z. The addition of a third dimension substantially improves scalability. In very large systems, a two-dimensional topology would produce excessive communication distances and congestion. By introducing another dimension, the average number of hops between chips decreases significantly. A TPU v4 superpod, for example, can contain a $16 \times 16 \times 16$ topology, corresponding to 4096 TPU chips. TPU systems typically use deterministic dimension-order routing. In a three-dimensional torus, packets may first traverse the X dimension, then the Y dimension, and finally the Z dimension. This simplifies routing hardware and helps avoid deadlocks. Modern TPU systems also employ a twisted torus configuration. In a conventional torus, opposite edges connect directly. In a twisted torus, the wraparound connections are intentionally twisted, to reduce average communication distance even further and distribute traffic more evenly across the network.

Datacenter Network (DCN)

A single ICI-connected topology is called a TPU slice. A slice refers specifically to a directly interconnected subset of TPUs participating in one communication domain, whereas a TPU pod refers to the larger physical TPU supercomputer that may contain one or more slices. Extremely large workloads may require multiple slices or even multiple TPU pods. In such cases, communication occurs over the datacenter network (DCN). Unlike ICI communication, DCN transfers are not direct TPU-to-TPU operations. Instead, tensors must first move from TPU HBM through PCIe into the host CPU memory. The data is then transmitted over the datacenter network through network interface cards before finally being transferred into the destination TPU. This process introduces substantially higher latency and lower bandwidth. As a result, modern distributed training systems attempt to structure computations so that most communication remains within a single slice whenever possible.

In summary, TPU systems therefore exhibit a hierarchy of bandwidths. The fastest communication occurs inside the compute pipeline itself, followed by on-chip memory accesses, inter-chip communication, and finally datacenter networking. For TPU v5p systems, HBM bandwidth is approximately 2.8 TB/s per chip. ICI bandwidth is roughly 90 GB/s per axis, with three communication axes available per chip. DCN bandwidth, however, is only around 6.25 GB/s per TPU.

3.7.4 CPU vs. GPU vs. TPU

We have seen execution models on three types of hardware — CPU, GPU, and TPU. Let us now compare these three execution models on various axes.

Control Flow and Architecture

A **CPU** has 8–128 high-capability cores with out-of-order execution and branch prediction, optimized for low-latency serial workloads. A **GPU** exposes thousands of CUDA/Tensor cores organized into SMs under the SIMT model, built for throughput over latency. A **TPU** is a narrow compute engine whose execution model assumes that the vast majority of computation is matrix multiplication.

Memory Architecture and Bandwidth

CPUs rely on multi-level cache hierarchies (L1/L2/L3) that work well for programs with data locality, but suffer high cache-miss rates under the large, irregularly-accessed tensors. **GPUs** pair high-bandwidth memory (HBM2/3, up to 3.5 TB/s) with thread-level parallelism to hide latency. **TPUs** use a systolic array where operands flow between adjacent multiply-accumulate units, eliminating repeated off-chip access during core computation, but only if the access pattern is entirely regular.

Compiler and Software Stack

CPU compilers handle ML by vectorizing loops and scheduling AVX SIMD instructions. **GPU** compilers must manage thread-block partitioning, shared memory reuse, and kernel fusion to avoid excessive HBM traffic. **TPU** compilers (XLA) face the hardest task: padding tensors to multiples of 128, handling activations, and scheduling a continuous feed to the MXU.

Performance, Cost, and Energy Efficiency

CPUs are power-efficient per core but cannot compete at training scale; an 8-GPU server does in minutes what a 64-core CPU takes hours to finish. **GPUs** are the dominant training accelerators, bu demand a lot of power and are expensive. **TPUs** offer 2–10× better throughput-per-watt on transformer workloads but lose their advantage on sparse or irregular computation that the systolic array cannot efficiently execute.

Training vs. Inference Suitability

CPU inference is viable for small models and latency-critical, unbatchable requests. **GPUs** handle both large-batch training and high-throughput inference well; at low-latency single-query inference, they are often underutilized. **TPUs** excel at very large-model training via high-bandwidth inter-chip interconnects (TPU pods), and perform similarly to GPUs in batch inference.

Programmability

CPUs are fully general, and can execute any algorithm. **GPUs** benefit from a mature ecosystem (CUDA, cuDNN, Triton); writing custom kernels requires a bit of parallel programming expertise. **TPUs** expose little low-level control and are accessed primarily through JAX or TensorFlow/XLA; experimentation with custom operators is significantly harder.

Table 3.6 summarize these differences.

Category	**CPU**	**GPU**	**TPU**
Core count	8–128 powerful cores	Thousands of CUDA/Tensor cores	Systolic array (128×128 MXU)
Execution model	Complex components such as OOO execution	SIMT; warps hide stalls	Dataflow; operands pass between units
Memory bandwidth	50–100 GB/s	1–3.5 TB/s (HBM2/3)	2–3 TB/s
Compiler burden	Low	High	Very high
Peak power	5–30 W	400–700 W	200–450 W
Irregular ops	Handles easily	Handles well	Poor; systolic array stalls
Ecosystem	Universal; any language or framework	CUDA, cuDNN, Triton	JAX, TensorFlow/XLA; limited low-level access

Table 3.6: CPU vs. GPU vs. TPU comparison

3.8 Advanced Hardware Features

In modern machine learning systems, the arithmetic processing units, such as tensor cores, have become incredibly fast. The primary bottleneck is no longer compute power, but rather delivering the data to these processing units fast enough. To address this gap, recent GPU architectures introduce advanced hardware features like asynchronous memory transfer, specialized memory transfer accelerators, warpgroup-level operations, and warp specialization. These hardware and software innovations work together to ensure that the processing units are never left waiting for data, thereby improving performance.

Tensor Core Programming Model

Before diving into the newer hardware features, let us understand how to use tensor cores. Tensor cores operate on small matrices (e.g. $16 \times 16 \times 16$) in a single collective operation. The fundamental operation performed is the Matrix-Multiply-Accumulate (MMA):

$$D = A \times B + C$$

where A, B, C, and D are matrix fragments.

The most accessible way to program tensor cores is through the `nvcuda::wmma` C++ namespace. This API abstracts underlying PTX instructions into a high-level worklow as follows. Threads in a warp cooperate to load pieces of matrices from global or shared memory into specialized structures called fragments using `load_matrix_sync`. The `mma_sync` operation then triggers the tensor core to perform the multiplication and addition across the entire warp. The code snippet is shown below:

Code 3.19: Tensor Core Programming

```
#include <mma.h>
using namespace nvcuda;

wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;

wmma::load_matrix_sync(a_frag, smem_ptr_a, ldm);
wmma::load_matrix_sync(b_frag, smem_ptr_b, ldm);
wmma::fill_fragment(acc_frag, 0.0f);

wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
```

While the WMMA API is portable across different GPU generations, performance-critical kernels often use lower-level PTX MMA instructions (e.g., `mma.sync.aligned.m16n8k16`). These instruc-

tions provide finer control over register layout and data types.

However, as GPU architectures evolved, the overhead of manually moving data into registers for these instructions became a bottleneck. This has led to the development of the more advanced, asynchronous paradigms described below, which allow the hardware to manage data movement and computation concurrently.

Asynchronous Memory Transfer

Traditionally, when a GPU thread requires data from the main memory, it requests the data and then stalls. The thread cannot perform any further instructions until the data arrives. This synchronous behaviour wastes valuable computation cycles. Asynchronous memory transfer solves this problem by decoupling the memory request from the data arrival. A thread can issue an asynchronous command to load data from global memory into a fast, on-chip shared memory buffer and immediately move on to execute other independent instructions. The thread only needs to wait at a synchronization barrier when it absolutely requires the fetched data to proceed. By overlapping memory operations with active computation, the system effectively hides the high latency associated with accessing global memory.

Tensor Memory Accelerator

To further enhance asynchronous memory movement, modern hardware introduces the Tensor Memory Accelerator, commonly known as TMA. The TMA is a specialized hardware unit dedicated to fetching multi-dimensional arrays from global memory and placing them directly into shared memory. Before the introduction of the TMA, standard CUDA cores were responsible for computing memory addresses and routing the data through the register file before it reached shared memory. This traditional path consumed valuable registers and distracted the CUDA cores from their primary task of performing arithmetic. In contrast, the TMA takes over the entire burden of data movement. The programmer simply provides a base pointer and an offset, and the TMA hardware automatically calculates the memory addresses for bulk tensor transfers. It deposits the data into shared memory, completely bypassing the thread registers. Once the data transfer is complete, the TMA updates an asynchronous barrier to signal to the computing threads that the data is ready to be used.

```
// A single thread initiates an asynchronous bulk memory transfer using TMA
__shared__ float smem_buffer[TILE_SIZE];
cute::cp_async_bulk_tensor_to_shared(smem_buffer, tma_descriptor);
```

Warpgroup Matrix-Multiply-Accumulate

While the TMA solves the problem of moving data into shared memory, the next step is feeding that data into the tensor cores for calculation. This is where Warpgroup Matrix-Multiply-Accumulate, or WGMMA, comes into play. A warpgroup is a collection of four contiguous warps, totalling 128 threads. WGMMA allows this entire warpgroup to issue a single collective asynchronous instruction to perform matrix multiplication on the tensor cores. Another feature of WGMMA is its ability to

source operands directly from shared memory. In older designs, data had to be explicitly loaded from shared memory into individual thread registers before the tensor cores could consume it. WGMMA removes this intermediate step entirely by pulling data straight from the shared memory buffers and feeding them into the tensor cores asynchronously. Because WGMMA is non-blocking, the threads can immediately go back to managing other tasks, such as preparing the next memory load, while the tensor cores independently compute the matrix product.

Warp Specialization

With TMA handling memory movement and WGMMA handling the computation, developers need a way to coordinate these two asynchronous systems. This coordination is achieved through warp specialization. Instead of having every warp in a thread block take turns loading data and performing math, warp specialization assigns permanent, dedicated roles to specific warpgroups. Warpgroups are strictly divided into producers and consumers. The producer warpgroup is exclusively responsible for data movement. It is made extremely lightweight, utilizing very few registers, and its sole job is to issue TMA instructions to keep the shared memory buffers full. The consumer warpgroups, on the other hand, are given a massive allocation of registers and focus entirely on executing WGMMA math instructions.

A popular architectural pattern built on this concept is the Ping-Pong GEMM kernel. In a Ping-Pong kernel, there is one producer warpgroup and two separate consumer warpgroups. While the first consumer warpgroup is busy actively calculating a matrix multiplication using the tensor cores, the second consumer warpgroup finalizes its previous calculation and writes the results back to memory. Once the first consumer finishes, they swap roles. This continuous ping-pong effect guarantees that the tensor cores are highly utilized.

Programming Assignment 3.10: Matrix Multiplication Using Tensor Cores

In this assignment, you will implement a a high-performance matrix multiplication kernel in CUDA using NVIDIA tensor cores through the Warp Matrix Multiply-Accumulate (WMMA) API. The goal is to compute $C = A \times B$ for large matrices where the input matrices are stored in FP16 format and the output matrix is accumulated in FP32. You will first implement a helper kernel that converts FP32 matrices to FP16 on the GPU, then develop a tensor core matrix multiplication kernel in which each warp computes a single 16×16 output tile. For every 16×16×16 K-dimension tile, the kernel loads fragments of matrices A and B into WMMA fragments, performs warp-level matrix multiply-accumulate operations using tensor cores, and accumulates the results into FP32 accumulator fragments. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/matrix_multiplication_tensor_cores

References

Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym (2008). "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2, pp. 39–55. DOI: 10.1109/MM.2008.31.

NVIDIA (2025). *CUDA Programming Guide*. https://docs.nvidia.com/cuda/cuda-programming-guide.