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

# 5: Accelerators, GPUs, and CUDA

*Lecturer: Hao Zhang*

*Scribe: Abdumannon Yovkochov, Vishruth Bharath, Kevin Nie, Cody Wang, Selina Xiong, Logan Ying, Zeyang Zhou, Oliver Kang*

## 1 Review MCQs

The lecture began with a series of multiple-choice questions reviewing core concepts from previous lectures on operator optimization and memory hierarchies.

### 1.1 Arithmetic Intensity of Matrix Multiplication

**Problem.** Given two $4 \times 4$ matrices $A$ and $B$, consider the following function:

```
func(matrix A, matrix B):
    Load A
    Load B
    C = matmul(A, B)
```

What is the arithmetic intensity?

**Solution.** We recall that for a matrix multiplication $C = A \times B$ where $A$ is $(m, n)$, $B$ is $(n, p)$, and $C$ is $(m, p)$, the number of floating-point operations is $2mnp$, since we have one multiply and one add per element of the dot product.

For $m = n = p = 4$:

- **FLOPs**: $2 \times 4 \times 4 \times 4 = 128$
- **Memory accesses**: We load $A$ ($4 \times 4 = 16$ floats) and $B$ ($4 \times 4 = 16$ floats), for a total of 32 floats $= 32 \times 4 = 128$ bytes.
- **Arithmetic intensity**: $\frac{128 \text{ FLOPs}}{128 \text{ bytes}} = 1$ FLOP/byte.

Thus, we have that the arithmetic intensity is $\boxed{1 \text{ FLOP/byte}}$.

### 1.2 Strided Representation Limitations

**Problem.** Which of the following tensor manipulations *cannot* benefit from a strided representation?

(A) `broadcast_to`

(B) `slice`

(C) `reshape`

(D) `permute dimensions`

(E) `transpose`

(F) `contiguous`

(G) `indexing like t[:, 1:5]`

(F) `contiguous.` This is because strided representations allow many operations, such as broadcasting (via zero strides), slicing and indexing (via offset adjustments), reshaping (when the tensor is already contiguous), permuting/transposing (via swapping strides). These are performed as metadata-only changes without copying data. The `contiguous` operation, however, explicitly forces a copy of data into a new contiguous memory layout, which is the opposite of what strided representations aim to avoid.

## 1.3 Computing Strides for a Contiguous Tensor

**Problem.** A tensor of shape $[2, 9, 1]$ is stored contiguously in memory in row-major order. What are its strides?

(A) $(9, 1, 1)$

(B) $(2, 9, 1)$

(C) $(1, 9, 2)$

(D) $(9, 9, 9)$

For a contiguous row-major tensor with shape $(d_0, d_1, d_2)$, the stride for dimension $i$ is the product of all subsequent dimensions. Formally, we have that:

$$\begin{aligned}\text{stride}_2 &= 1\\ \text{stride}_1 &= d_2 = 1\\ \text{stride}_0 &= d_1 \times d_2 = 9 \times 1 = 9\end{aligned}$$

So the strides are (A) $(9, 1, 1)$.

## 1.4 Cache Tiling in Matrix Multiplication

**Problem.** Which of the following is true for cache tiling in matmul?

(A) It saves memory allocated in cache.

(B) It reduces memory movement between cache and registers.

(C) It reuses memory movement between DRAM and cache.

(D) It increases arithmetic intensity because it makes computation faster.

(C) It reuses memory movement between DRAM and cache. Cache tiling works by loading a tile (sub-block) of the input matrices into cache and then performing as much computation as possible on that tile before evicting it. This reuses the data movement between DRAM and cache—once a tile is in cache, multiple computations reference it without additional DRAM fetches. This effectively increases arithmetic intensity by increasing the number of FLOPs performed per byte transferred from main memory, not by making individual operations faster (which also rules out option D).

# 2 Course Roadmap

At this point in the course, we have covered several layers of the deep learning systems stack from the bottom up:

- **Operator-level optimizations** (completed): vectorization, data layout transformations, and parallelization at the operator level.
- **Matrix multiplication optimizations** (completed): cache tiling, register blocking, and SIMD vectorization applied specifically to matmul.
- **GPUs and accelerators** (this lecture): understanding GPU hardware, the CUDA programming model, and the broader accelerator landscape.

The remaining topics in the course such as graph optimization, automatic differentiation, parallelization strategies (data/model/pipeline parallelism), and runtime scheduling, all build on top of these foundational layers.

# 3 Recap: Optimized Matrix Multiplication on CPUs

Before moving to GPUs, we reviewed the progression of matmul optimization from naïve to highly optimized CPU code.

## 3.1 Naïve Implementation

The starting point is the textbook triple-nested loop for computing $C = A \cdot B^T$ (equivalently, $C = \texttt{dot}(A, B.T)$):

```
float A[n][n], B[n][n], C[n][n];

for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) {
    C[i][j] = 0;
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[j][k];
    }
  }
```

This implementation suffers from poor cache utilization, as each element of $A$ and $B$ is loaded from DRAM many times across different iterations.

### 3.2 Tiled and Vectorized Implementation

The optimized version applies cache tiling and register blocking. The arrays are logically partitioned into tiles that fit in L1 cache, and within each tile, vector registers are used for the innermost computation:

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
  l1cache float a[b1/v1][n][v1] = A[i];
  for (int j = 0; j < n/b2; ++j) {
    l1cache float b[b2/v2][n][v2] = B[j];
    for (int x = 0; x < b1/v1; ++x)
      for (int y = 0; y < b2/v2; ++y) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n; ++k) {
          register float ar[v1] = a[x][k][:];
          register float br[v2] = b[y][k][:];
          C += dot(ar, br.T);
        }
      }
  }
}
```

The key optimizations here are (1) **cache tiling**, where the tiles of $A$ and $B$ are loaded into L1 cache and reused across many multiply-accumulate operations, (2) **register blocking** as small sub-tiles are kept in registers to minimize cache-to-register traffic, and (3) **vectorization**, as the innermost operations use SIMD vector instructions.

### 3.3 CPU Parallelization

The tiled matmul can be further parallelized across CPU cores using thread-level parallelism or using many concurrent cores. For simpler element-wise operations, this looks like:

```
#pragma omp parallel for
for (int i = 0; i < 64; ++i) {
    float4 a = load_float4(A + i*4);
    float4 b = load_float4(B + i*4);
    float4 c = add_float4(a, b);
    store_float4(C * 4, c);
}
```

This combines vectorization, where we process 4 floats per instruction via `float4` with parallelization, which is essentially just distributing loop iterations across CPU threads via OpenMP.

## 4 Single-Instruction Multiple-Data (SIMD)

The SIMD execution model is key to understanding both CPU vector units and GPU architectures. In SIMD, a single instruction is broadcast to multiple processing units (PUs), each of which applies the instruction to a different data element simultaneously.

For example, a scalar addition requires four separate instructions to compute $A_x + B_x = C_x$, $A_y + B_y = C_y$, $A_z + B_z = C_z$, $A_w + B_w = C_w$. A SIMD operation of vector length 4 computes all four additions in a single instruction cycle. Intel architectures currently support SIMD vector lengths of 4, 8, and 16 (via SSE, AVX2, and AVX-512, respectively).

This model is the foundation upon which GPU architectures are built. As we'll see moving forward, GPUs take the SIMD concept to an extreme scale with thousands of lightweight processing units (arithmetic logic units) executing the same instruction on all the processors at the same time "in parallel".

## 5 Chip Industry

Historically, system designers observed that increasing the number of processing units improves performance. This insight led to the widespread adoption of parallel architectures such as SIMD. However, chip design must operate within strict physical constraints, particularly limited chip area. As illustrated in the Figure 1 (CPU vs. GPU layout), a processor chip is divided into several key components:

- Control Units: Handle instruction flow (e.g., branching and decision-making).
- Caches (L1, L2, L3): Store frequently accessed data to reduce memory latency.
- ALUs (Arithmetic Logic Units): Perform computations and determine the level of data parallelism.

In CPU design (left side of the figure), a significant portion of the chip area is dedicated to control logic and multiple levels of cache (L1, L2, L3). This enables CPUs to efficiently handle complex control flows and diverse workloads.

In contrast, GPU design (right side of the figure) minimizes control logic and simplifies cache structure, allocating most of the chip area to a large number of ALUs. This design maximizes parallel computation, making GPUs highly effective for workloads with simple, repetitive operations (e.g., matrix and tensor computations). Advances in semiconductor technology have allowed engineers to shrink transistor sizes (currently on the order of a few nanometers), enabling more ALUs to fit within the same chip area. However, this scaling is approaching physical limits.

## 6 Limits of Scaling and Performance Plateau

The Figure 2 illustrates the historical improvement in processor performance. Early gains followed trends such as Moore's Law and Dennard scaling, where performance increased rapidly over time. However, as shown in the figure, this growth begins to plateau in recent years.

This plateau occurs due to fundamental physical constraints, such as power consumption limits, heat dissipation challenges and diminishing returns from transistor scaling. While demand for computational power continues to grow, the ability to increase performance through traditional scaling has slowed significantly.

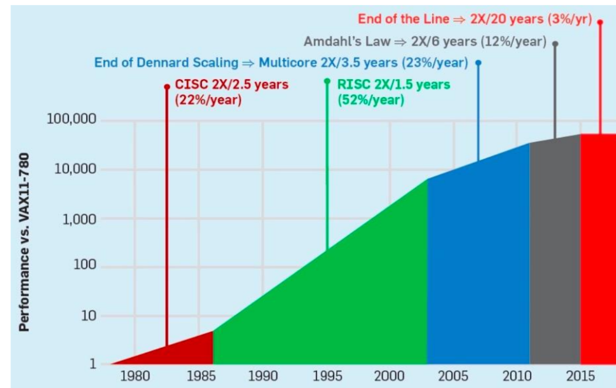


Figure 1: Historical trend of processor performance showing rapid growth followed by a plateau due to physical scaling limits.

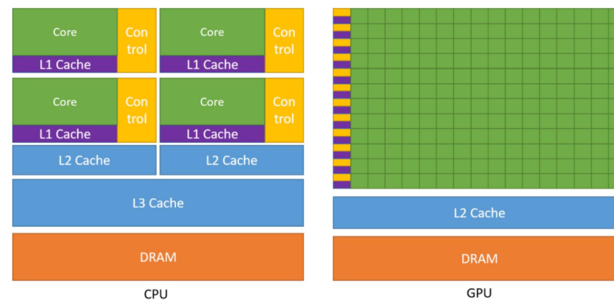


Figure 2: Comparison of CPU and GPU architectures highlighting control logic, cache hierarchy, and the higher density of ALUs in GPUs for parallel processing.

# 7 Specialized Hardware as a Solution

To address this slowdown, researchers and engineers are exploring alternative approaches. One of the research areas is quantum computing, although promising, it is still in early stages and not yet practical for most real-world applications. Another more popular approach is specialized hardware (accelerators). Unlike general-purpose CPUs, accelerators are designed for specific types of computations. GPUs are a prime example: by dedicating most of their chip area to ALUs and reducing control overhead, they excel at highly parallel tasks. These architectures are particularly effective for matrix and tensor operations, which are central to modern machine learning. Their widespread adoption was driven initially by graphics processing (notably by NVIDIA in the early 2000s) and later by breakthroughs such as AlexNet, developed by Geoffrey Hinton's lab, which demonstrated the use of GPUs for deep learning. Another example is the Tensor Processing Unit (TPU), developed by Google. TPUs are application-specific integrated circuits (ASICs) optimized specifically for tensor computations, further pushing efficiency for machine learning workloads.

# 8 What Does "Specialized" Mean in the Accelerator World?

As discussed, specialized hardware is a promising path forward when general-purpose CPU scaling stalls. But what does "specialized" actually mean in practice? There are several dimensions along which a hardware

accelerator can be specialized.

### 8.1 Functionality Specialization

The most direct form of specialization is restricting what computations the hardware can perform. Rather than supporting arbitrary programs, specialized accelerators may only support a limited set of operations such as dense matrix multiplication or matrix multiplication with sparsity. To compensate for the loss in generality, these chips can dedicate far more transistor area to executing those specific operations efficiently. In practice, many modern accelerators mix specialized cores for targeted workloads (e.g., tensor operations) with more versatile cores that handle general control flow.

### 8.2 Reduced Numerical Precision

Another powerful form of specialization is reducing the bit-width of floating-point or integer arithmetic. Standard scientific computing traditionally uses 32-bit floating-point (`fp32`), but modern ML accelerators support a spectrum of lower-precision formats:

- `fp32`: 1 sign bit, 8 exponent bits, 23 mantissa bits (standard precision)
- `fp16`: 1 sign bit, 5 exponent bits, 10 mantissa bits
- `bf16` (BFloat16): 1 sign bit, 8 exponent bits, 7 mantissa bits (same exponent range as `fp32`, less mantissa)
- `fp8`, `int8`, `int4`: progressively lower precision

Because hardware area and energy scale roughly with the square of the bit-width, halving the precision can approximately quadruple the number of arithmetic units that fit in the same silicon area. This is why lower-precision formats can yield dramatically higher peak FLOP/s on the same chip. The key observation that makes this work in ML is that neural network training and inference are surprisingly robust to reduced numerical precision: small rounding errors in individual operations tend to average out over millions of parameters and training steps.

### 8.3 Tuning the Component Distribution

Beyond computation, specialized hardware also re-balances the distribution of on-chip resources: the ratio of compute units, SRAM, cache, and registers is tuned specifically for the target workload rather than optimizing for general-purpose programs.

## 9 Case Studies in Hardware Specialization

The following three case studies illustrate how the specialization principles above are realized in actual products, each taking a different approach to the same underlying challenge.

## 9.1 NVIDIA Vera Rubin: Precision Scaling

NVIDIA's Vera Rubin GPU demonstrates specialization through reduced numerical precision. Figure 3 shows the throughput specifications across three configuration scales.

| | NVIDIA Vera Rubin NVL72 | NVIDIA Vera Rubin Superchip | NVIDIA Rubin GPU |
|---|---|---|---|
| **Configuration** | 72 NVIDIA Rubin GPUs \| 36 NVIDIA Vera CPUs | 2 NVIDIA Rubin GPUs \| 1 NVIDIA Vera CPU | 1 NVIDIA Rubin GPU |
| **NVFP4 Inference** | 3,600 PFLOPS | 100 PFLOPS | 50 PFLOPS |
| **NVFP4 Training[2]** | 2,520 PFLOPS | 70 PFLOPS | 35 PFLOPS |
| **FP8/FP6 Training[2]** | 1,260 PFLOPS | 35 PFLOPS | 17.5 PFLOPS |
| **INT8[2]** | 18 POPS | 0.5 POPS | 0.25 POPS |
| **FP16/BF16[2]** | 288 PFLOPS | 8 PFLOPS | 4 PFLOPS |
| **TF32[2]** | 144 PFLOPS | 4 PFLOPS | 2 PFLOPS |
| **FP32** | 9,360 TFLOPS | 260 TFLOPS | 130 TFLOPS |
| **FP64** | 2,400 TFLOPS | 67 TFLOPS | 33 TFLOPS |
| **FP32 SGEMM[3]** | 28,800 TFLOPS | 800 TFLOPS | 400 TFLOPS |
| **FP64 DGEMM[3]** | 14,400 TFLOPS | 400 TFLOPS | 200 TFLOPS |
| **GPU Memory \| Bandwidth** | 20.7 TB HBM4 \| 1,580 TB/s | 576 GB HBM4 \| 44 TB/s | 288 GB HBM4 \| 22 TB/s |
| **NVLink Bandwidth** | 260 TB/s | 7.2 TB/s | 3.6 TB/s |

Figure 3: NVIDIA Vera Rubin GPU specifications. Red boxes highlight FP16/BF16 (4 PFLOPS) and TF32 (2 PFLOPS) on a single GPU, illustrating the 2× throughput gain per precision step-down.

A single Rubin GPU delivers 2 PFLOPS at TF32, 4 PFLOPS at FP16/BF16, and 17.5 PFLOPS at FP8—roughly a 2× gain per step down in precision. This is consistent with the theoretical expectation that halving bit-width approximately doubles the number of arithmetic units that fit in the same silicon area. The reason this is practical in ML is illustrated in Figure 4: BFloat16 preserves the same 8-bit exponent as FP32 while cutting the mantissa from 23 to 7 bits. Neural networks tolerate the resulting rounding errors well, making lower-precision formats an effective lever for throughput.

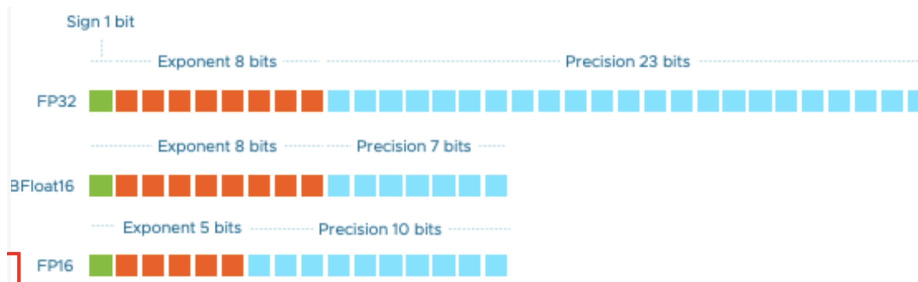


Figure 4: Bit-layout comparison of FP32, BFloat16, and FP16. BFloat16 shares FP32's exponent width, preserving dynamic range at the cost of mantissa precision.

### 9.2 Apple Silicon: Unified Memory Architecture

Apple's M-series chips illustrate specialization through architectural integration. Rather than a discrete GPU connected to the CPU via PCIe, Apple places CPU, GPU, and Neural Engine on a single die sharing one physical memory pool. The M3 MAX, for instance, pairs a 16-core CPU with a 40-core GPU and up to 128 GB of unified memory across 92 billion transistors.

The key benefit is eliminating CPU-to-GPU data transfers entirely. In conventional systems, moving a tensor from host DRAM to GPU VRAM is a major latency bottleneck; with unified memory, all processors read from and write to the same address space. This makes Apple Silicon particularly efficient for inference workloads where the model fits in the shared memory pool.

### 9.3 Groq LPU: On-Chip SRAM for Memory-Bound Inference

Groq takes the most radical approach: eliminating DRAM altogether for inference. Their benchmarks on Llama 3.3 70B show up to 1,678 output tokens/second, compared to 30–120 tokens/second on NVIDIA GPU-based cloud providers.

The insight behind this is that autoregressive LLM decoding is *memory-bandwidth bound*, not compute bound. Each generated token requires streaming the full model weights from memory, so the bottleneck is bandwidth rather than FLOP/s. Groq's Groq 3 LPU addresses this with 500 MB of on-chip SRAM running at 150 TB/s—far exceeding the 1–3 TB/s typical of off-chip DRAM. The chip is organized into specialized execution units all operating directly on this fast on-chip memory, with 96 chip-to-chip links at 112 Gbps each for multi-chip scaling. The tradeoff is capacity: 500 MB of SRAM is far smaller than tens of gigabytes of DRAM, so this approach suits inference workloads where weights can be fully staged on-chip or carefully orchestrated in a static schedule.

## 10 Motivation for GPUs and the CUDA Abstraction

### 10.1 From Accelerator Trends to GPUs

Hardware generations differ and specialized hardware is important for machine learning workloads. A broader question is: What is NVIDIA's moat? The key idea is that success in machine learning systems is not determined by hardware performance alone, but also by the surrounding software stack, programming model, and ecosystem adoption.

### 10.2 GPU Overview

A GPU is organized around many **streaming multiprocessors (SMs)** together with a memory hierarchy. Compared with CPUs, GPUs are built for much larger-scale parallel execution. Instead of relying on a small number of strong general-purpose cores, they use many execution units to support massive data parallelism.

At a high level, a GPU is organized around many **streaming multiprocessors (SMs)**, together with a memory hierarchy. Compared with CPUs, GPUs are designed for much larger-scale parallel execution. Instead of a small number of strong general-purpose cores, GPUs use many execution units to support massive data parallelism.

## 10.3 CUDA Execution Hierarchy

CUDA provides a structured programming model for expressing this parallelism. The four main concepts are:

- **Thread**: the smallest unit of execution.
- **Block**: a group of threads that can cooperate and share memory.
- **Grid**: a collection of blocks executing the same kernel.
- **Kernel**: a CUDA function launched in parallel on the GPU.

This hierarchy is the core CUDA abstraction: a programmer writes one kernel, launches it over a grid, and the work is divided among blocks and threads.

## 10.4 Threads, Blocks, and Grids

A **CUDA thread** is a lightweight execution unit designed for massive parallelism, unlike an **OS thread**, which is a heavier abstraction managed by the operating system.

A **thread block** is mapped to one SM. Threads in the same block can cooperate and use shared memory, so a block acts as both a programming unit and a hardware scheduling unit.

A **grid** is the full collection of blocks launched for one kernel. This abstraction allows the same CUDA program to scale across different GPUs even when the underlying hardware differs.

## 10.5 Hardware Trends

Several NVIDIA GPU generations are shown to illustrate a broader trend: machine learning hardware evolves quickly, and newer GPUs provide increasing parallel capacity and stronger support for ML workloads.

# 11 CUDA

CUDA is a C-like programming language designed specifically for programming GPUs. Introduced by NVIDIA in 2007 alongside the Tesla architecture, the language was designed to map to the underlying GPU hardware, requiring developers to explicitly structure their computations into a hierarchy of grids, thread blocks, and threads.

## 11.1 An Example CUDA Program: Matrix Addition

CUDA programs are divided into two parts: host code that runs serially on the CPU, and device code that utilizes SIMD parallel execution on the GPU. Let's look at an example of matrix addition using CUDA, starting with the host code:

```
1 const int Nx = 12;
2 const int Ny = 6;
3
```

```
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

We define `threadsPerBlock` as a 4x3 grid, meaning each block contains 12 threads. Next we calculate `numBlocks` by dividing the total array dimensions (12×6) by our block dimensions, resulting in a 3×2 grid, or 6 blocks. Finally, we launch the `matrixAddDoubleB` kernel using the `<<<numBlocks, threadsPerBlock>>>` syntax. Here, the host CPU instructs the GPU to asynchronously spawn a total of 72 threads (6 blocks × 12 threads).

```
__device__ float doubleValue(float x)
{
    return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

Once the kernel is launched, execution moves to the GPU. Here, `__global__` indicates a CUDA kernel function that runs on a GPU, and the `__device__` decorator defines a function (`doubleValue`) that executes on the GPU and can only be called by other GPU functions.

Because the GPU executes this kernel in a massively parallel SIMD fashion, all 72 threads execute the same piece of code simultaneously. Each thread uses its block index, block dimension, and thread index to determine which specific elements it is responsible for. Using the uniquely computed `i` and `j` values, each thread reads its specific target values for `A[j][i]` and `B[j][i]`, computes the sum, and writes the result to `C[j][i]`.

As element-wise operations like matrix addition are easily parallelizable, their resulting CUDA kernels are relatively straightforward. However, as operations grow more complex, introducing inter-thread dependencies and requiring strict management of the memory hierarchy, CUDA programming becomes significantly more difficult.

## 11.2 Asynchronous Kernel Execution

When the host code launches a kernel, the process becomes entirely asynchronous. The GPU begins its computation, but the CPU immediately proceeds to its next line of code without waiting for the GPU to finish. This asynchronous hand-off creates a common pitfall for developers building machine learning systems. If the computational results are accessed immediately after the kernel launch line, the data will be

incorrect as the GPU has not finished working yet, which can falsely appear as a logic bug. To resolve this, developers must explicitly call a synchronization API that forces the CPU to pause and wait for the GPU's results before moving forward.

## 11.3 Explicit and Static Thread Management

Beyond managing execution timing, CUDA requires developers to strictly separate their CPU and GPU code, statically declare block and thread sizes, and manually dictate exactly how the data maps to those structures. Because this data mapping is manual, developers must implement boundary condition checks inside the kernel. This ensures that threads do not attempt to access invalid memory locations, which would cause the program to crash.

Wile these boundary condition checks are essential for memory safety, they introduce a significant hardware conflict. Because GPUs consist of highly specialized cores built strictly for synchronized SIMD execution, they struggle to process diverging control flows like if/else statements. Managing this specific hardware limitation without sacrificing performance is a major optimization challenge in GPU programming.