CSE-234: Data Systems for Machine Learning, Winter 2025

5: GPUs and CUDA

Lecturer: Hao Zhang

Scribes: Sudhansh Peddabomma, Brandon Hsu, Melvyn Tan, Kuber Shahi, Vincent Thai, Ketaki Tank, Yi Li, Sarthak Kala, Nai-En Kuo, Keivan Rahmani, Allen Lu

1 Recap

We have seen how to make a general operator faster, and realized that we need accelerator hardware since the general CPU cores are not optimized. On the same note, we discussed the architecture of such special hardware and the market built around it.

1.1 Multiple Choice Questions

MCQ 1) What is the arithmetic intensity for the following function:

```
# A, B are 2-D matrices of shape [2, 2]
Load A
Load B
C = matmul(A, B)
```

A. 0.334

B. 2

C. 1

D. **1.334**

Correct answer: **D**
Given $A \in \mathbb{R}^{mxn}, B \in \mathbb{R}^{nxp}, C \in \mathbb{R}^{mxp}$, the number of I/O operations is $mn + np + mp$, and the number of compute operations is $2mnp$ since there are approximately mnp addition and multiplication operations. The arithmetic intensity is then $\frac{\#\text{compute operations}}{\#\text{I/O operations}} = \frac{2mnp}{mn+np+mp}$. Setting $m = n = p = 2$, results in $\frac{2x2x2x2}{2x2+2x2+2x2} = \frac{4}{3}$.

Note. The addition operation discussed in the previous lecture also has the same I/O operations. However, `matmul` is a denser operation that results in a higher arithmetic intensity. In practice, it takes the same time to execute matrix addition and multiplication on GPUs, which is why they are so powerful.

MCQ 2) 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]`

Correct answer: **F**
A strided representation allows us to manipulate and access our underlying data in zero-copy fashion, which is both efficient and flexible. However, it also causes noncontiguous data access, creating issues for certain operations that require contiguous memory layout. Consequently, these operations cannot benefit from a strided representation when the data layout is noncontiguous.

MCQ 3) If we have a tensor of shape $[2, 9, 1]$ stored contiguously in memory following a row major, what are its strides:

A. **(9, 1, 1)**

B. (2, 9, 1)

C. (1, 9, 2)

D. (9, 9, 9)

Correct answer: **A**
The strides of a tensor of shape $[2, 9, 1]$ stored in row-major order are $[9, 1, 1]$ and contain 18 elements.

MCQ 4) Which of the following is True for *Cache Tiling* in Matmul?

A. It saves memory allocated in Cache

B. It reduces the memory movement between Cache to Register

C. **It reuses memory movement between DRAM and Cache**

D. It increases arithmetic intensity because it makes the computation faster

Correct answer: **C**
Cache Tiling is a method for reducing data movement from slow DRAM to faster L_n-cache. The goal is to maximize cache allocation; therefore, option A is incorrect because Cache Tiling does not aim to minimize data movement to registers, making option B incorrect as well. While Cache Tiling is arithmetically efficient, its efficiency stems from improved Input/Output (I/O) performance rather than computational efficiency.

2 GPU and CUDA

We have seen that specialized cores offer much better performance over traditional CPUs. Consider the following basic architecture of a GPU:

Let us see the basic terminology for understanding the architecture:

- **Threads** - Smallest units of execution to process a chunk of data. Threads are the lowest-level workers in GPU computing and execute instructions concurrently within a warp.

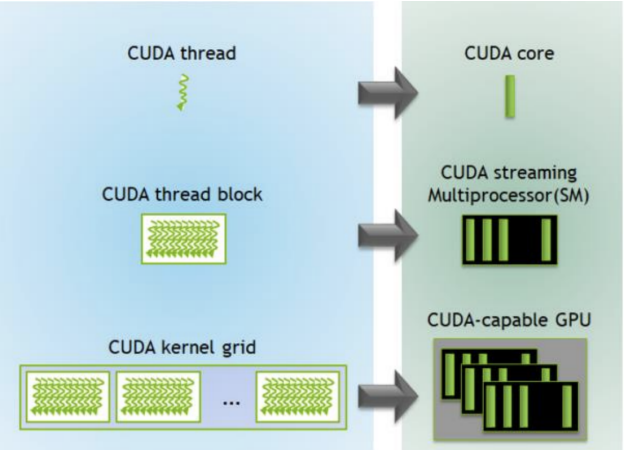

Figure 1: CUDA's hierarchical execution model.

- **Warps** - A group of 32 threads (usually on Nvidia GPUs) that execute instructions simultaneously in a Single Instruction, Multiple Thread (SIMT) model. Warps are the standard scheduling unit in a streaming multiprocessor.
- **Blocks** - A collection of threads that share memory. Each block is assigned to a single *streaming multiprocessor* (SM/SMP). The SM schedules and executes the threads in warps (usually 32 threads per warp).
- **Grid** - A collection of blocks that execute the same kernel. The grid defines the total structure of threads involved in the computation.
- **Kernel** - A function written in CUDA (or any other GPU programming framework) that is executed by many threads on the GPU in parallel. Each thread executes the kernel with a unique thread ID so that they can operate on different portions of data.

Characteristics of More Powerful GPUs

- **Higher FLOPS** - The ability to execute more floating-point operations per second improves the performance of GPU tasks.
- **More Steaming Multiprocessors** - Each SM can handle a set number of blocks and threads. Therefore, adding more SMs will result in higher computational throughput (throughput refers to the amount of computational work a system can complete in a given period of time; the faster the better).
- **More Cores per SM** - An increase in the number of cores in each SM allows more threads to execute simultaneously.
- **Improved Core Design** - Making the cores more powerful to allow for faster execution of tasks at lower energy consumption (currently, at a point of *diminishing rewards*).

NVIDIA, the largest GPU company, has released P100, V100, A100, H100, and B100 (Blackwell) for ML development. K80, P4, T4, and L4 were a lower tier of GPUs. Let's analyze how the compute has changed across these versions:

1. V100 (2019 -) - 80SMs, 2048 threads/SM - $3/hour.
2. A100 (2020 -) - 108SMs, 2048 threads/SM - $4/hour.
3. H100 (2022 -) - 144SMs, 2048 threads/SM - $12/hour.
4. B100 and B200 (2025 -) - Information about B100, Information about B200

The numbers are not doubling; then how has the performance doubled? The numbers are for mixed-precision performance... :(

2.1 CUDA

What is CUDA? It is a C-like language to program GPUs, first introduced in 2007 with NVIDIA Tesla architecture. It is designed after the grid/block/thread concepts.

CUDA code could be divided into two parts

1. **Host Code:** Host code refers to the portion of the program that runs on the computer's CPU. It is responsible for:
   - Managing the overall program flow
   - Allocating and deallocating memory on both the host (CPU) and device (GPU)
   - Transferring data between the host and device
   - Launching kernel functions to be executed on the GPU
   - Synchronizing CPU and GPU operations

   Host code is written in standard C or C++ and is compiled using a regular C/C++ compiler.
2. **Device Code:** Device code, on the other hand, is the part of the program that runs on the GPU. It consists of:
   - Kernel functions: Special functions that are executed in parallel on the GPU
   - Device functions: Helper functions that can only be called from within kernel functions or other device functions

   Device code is written using CUDA C/C++, which is an extension of standard C/C++ with additional keywords and constructs specific to GPU programming.

CUDA programs contain a hierarchy of threads. Consider the following host code:

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1); // 12
dim3 numBlocks(Ns/threadsPerBlock.x, Ny/threadsPerBlock.y, 1); // (3, 2, 1) = 6

// the following call triggers execution of 72 CUDA threads
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

The GPU models are associated with constants such as:

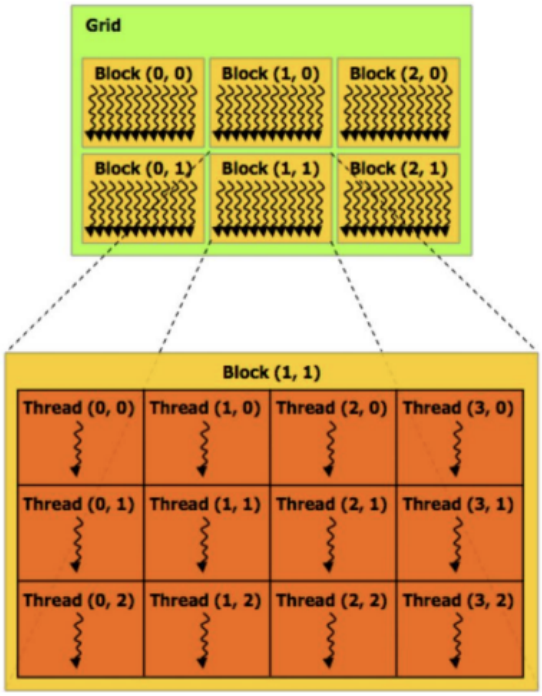

Figure 2: CUDA grid/block layout for a 12×6 problem using blocks of 4×3 threads, resulting in 6 blocks of 12 threads each (72 total).

- `GridDim` - Dimensions of the grid.
- `blockIdx` - The block index within the grid.
- `blockDim` - The dimensions of a block.
- `threadIdx` - The thread index within a block.

With these in mind, the CUDA kernel for the above code is designed as:

```
__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]);
}
```

The host code launches a grid of CUDA blocks, which then call the `matrixAdd` kernel. The function definition starts with `__global__` which denotes a CUDA kernel function that runs on the GPU. Each thread indexes its data using `blockIdx`, `blockDim`, `threadIdx` and executes the compute. It is the user's responsibility to ensure that the job is correctly partitioned and the memory is handled correctly.

The host code has a serial execution. However, the device code has SIMD parallel execution on the GPUs. When the kernel is launched, the CPU program continues executing without halting while the device code runs on the GPU. Due to this design, the device code mustn't have any return values—causes erroneous behavior. To get results from the GPU, `CUDA.synchronize` is used (an example will be shown later).

It is the developer's responsibility to map the data to blocks and threads. The blockDim, shapes, etc., should be statically declared. This is the reason why compilers like `torch.compile` require static shapes. The CUDA interface provides a CPU/GPU code separation to the users.

The SIMD implementation has a constraint for the control flow execution—it requires all ALUs/cores to process at the same pace. In a control flow, not all ALUs may do useful work, and it can lead to up to 8 times lower peak performance.

3 CUDA Programming Examples

3.1 Matrix Add

The last paragraph of Section 2.1 shows that CUDA code can be divided into two parts: host code and device code. Consider the following code.

The first part is the host code, which is executed serially on the CPU

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A,B,C);
```

The second part is the device code, which utilizes SIMD for parallel execution on GPUs.

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

Here, we assume that A, B, C are allocated $N_x \times N_y$ float arrays. We call an execution of $N_x \times N_y = 72$ threads. dim3 is a special CUDA datatype with 3 components .x, .y, .z each initialized to 1. Here, we specify

the .x and .y to 4 and 3. This means that we will have a block of 12 threads with 6 blocks in total. If we want to be more specific, the thread block would be of size 3×2 and the thread block would be of size 4×3

The host code launches a grid of CUDA thread blocks. Call returns when all threads have terminated.

As mentioned previously, the `__global__` keyword denotes a CUDA kernel function that is running on the GPU. For each thread, it indexes its data using the BlockIdx, BlockDim, and ThreadIdx to execute the compute.

The CPU program will continue execution immediately after launching the kernel. This is because CUDA kernel launches are asynchronous by default. When you call a CUDA kernel, the CPU doesn't wait for the GPU to finish - it continues executing the next line of code right away.

The kernel function is declared with `__global__` void, meaning it cannot directly return values. In CUDA, kernel functions must have a void return type because they're executed by multiple threads in parallel. Instead of return values, kernels typically write their results to pre-allocated GPU memory which can then be copied back to the CPU using `cudaMemcpy` when needed. The easiest way to know when the GPU has finished its work, is to call `cudaDeviceSynchronize()` to make the CPU wait for the GPU, but we want to minimize such calls to maximize compute.

One thing to note is that CUDA expects the user to be fully responsible for managing the correctness of the code. Threads are explicit and static in such programs, and developers are expected to provide the CPU/GPU code separation, statically declare blockDim and other shapes, and map data to blocks/threads. To ensure correctness, users should be extra careful when facing boundary checks, data mapping, and setting the shapes of blocks since these might affect the final result and performance of the CUDA code.

3.2 Handling Control Flow — Coherent vs. Divergent

As mentioned in Section 2.1, the SIMD implementation on CUDA Kernels has limitations. SIMD requires all the ALUs/cores to proceed at the same pace in a lockstep fashion. All ALUs or cores work in complete synchronization, executing the same instruction at the same time. This can be problematic when handling divergence or branching in the code as branching breaks the synchronization. While some ALUs execute one branch of the code, others must wait to execute their branch, making the execution serialized.

Let's illustrate this execution through a control flow example. Consider the following branching (`if-else` statement) kernel definition while running code on the GPU.

```
// kernel definition
__global__ void f(float A[])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = A[i];
    if (x > 0) {
        x = 2.0f * x;
    } else {
        x = exp(x, 5.0f);
    }
    A[i] = x;
}
```

To preface, coherent execution is a scenario in which all threads execute the same instruction in a lockstep fashion and divergent execution occurs if threads in the same warp take different branches of an `if-else`

statement. As show in the above CUDA kernel example, the kernel that checks whether `x > 0` can cause some threads to follow the "if" branch, while others follow the "else" branch. Under CUDA's execution model, all threads must sequentially execute both branches: first, the GPU hardware activates the threads that satisfy the "if" condition (while the others remain idle), then it activates the threads that satisfy the "else" condition (while the first group remains idle). This sequence introduces idle cycles—often called "bubbles"—and can diminish efficiency. As a result, it can lead to up to 8 times lower peak performance in a control flow.

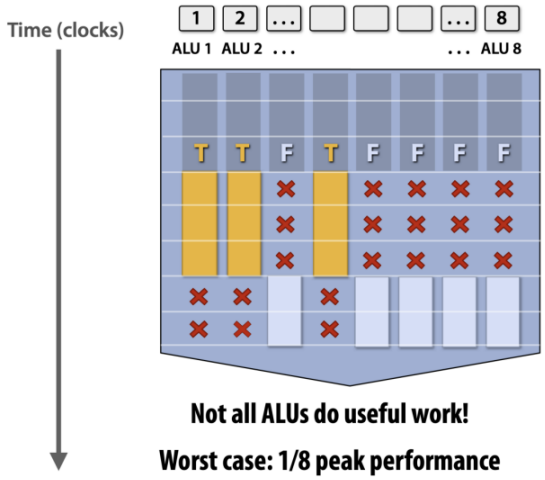

Figure 3: Control-flow divergence in a warp: threads follow different branches, causing some ALUs to remain idle and reducing overall GPU efficiency.

As an example, consider a warp of eight threads: three might satisfy `x > 0`, while the remaining five do not. The three threads that pass the condition will execute the instructions in the "if" block (with the other five idle), followed by execution of the "else" block for the remaining five threads (while the first three are idle). This sequential handling of both paths within a single warp leads to divergent execution, and minimizing such divergence is key to sustaining high performance.

A common situation where conditional checks arise is in language modeling tasks that use masking. For instance, causal masking discards future tokens by conditionally applying operations only to past tokens. In a naive CUDA implementation, this would require a conditional check in each thread, much like an `if-else` statement, which can create idle cycles whenever some threads are masked out and others are not.

To summarize, a coherent execution applies the same instructions to all data increasing GPU utilization and performance whereas divergent executions do the opposite and need to be minimized in CUDA programs.

4 CUDA Memory Model

4.1 Introduction

CUDA operates on GPUs using High Bandwidth Memory (HBM), a specialized memory technology that delivers significantly higher speed and bandwidth compared to traditional CPU memory, such as Dynamic Random Access Memory (DRAM).

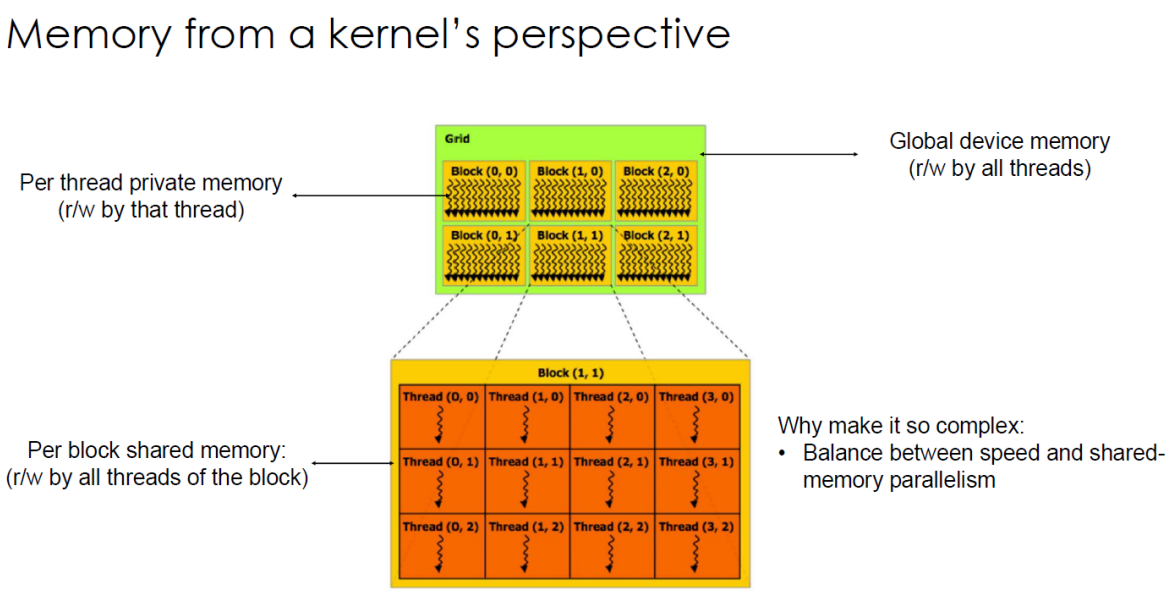

Figure 4: Memory From A Kernel's Perspective

In traditional CPU systems, host memory is managed by the operating system as pageable memory, which is divided into pages that can be indexed as needed. In contrast, GPU memory uses a memory pool model optimized for parallelism, allowing multiple threads to access memory concurrently. While the CPU executes instructions serially, relying on the host memory address space, the GPU executes Single Instruction, Multiple Data (SIMD) instructions, where thousands of threads operate in parallel, each performing operations on different data. This distinction underscores the separation of memory spaces in CUDA: the CPU uses host memory, while the GPU uses device memory, and neither can directly access the other's memory space without explicit data transfer.

Memory management in CUDA involves specific functions, such as `cudaMalloc` for memory allocation on the device and `cudaMemcpy` for transferring data between the host and device.

CUDA also introduces the concept of **pinned memory**, a dedicated portion of host memory optimized for high-speed data transfers between the CPU and GPU. Unlike regular pageable host memory, pinned memory is locked and non-pageable, allowing the GPU to access it more efficiently. Pinned memory is allocated and managed by NVIDIA drivers, which reserve a portion of the host memory to facilitate seamless data movement. Notably, certain NVIDIA APIs require the use of pinned memory to achieve optimal performance, as its locked nature eliminates the overhead associated with paging by the operating system.

Figure 4 shows the memory hierarchy from the perspective of the kernel. The CUDA memory model is hierarchical, with different memory types optimized for specific use cases.

- **Global memory (HBM)**: This resides in the GPU's device memory and is accessible to all threads. It offers the largest storage capacity but comes with higher latency and slower access times, making it suitable for data that is infrequently accessed or shared across the entire grid.
- **Shared memory**: This is a high-speed memory allocated per thread block, shared among all threads within the block. It provides significantly faster access than global memory but is limited in size. Shared memory is ideal for intermediate computations and data shared between threads in the same block.

- **Private memory**: Each thread has its own local private memory, which offers the fastest access speeds. However, private memory is limited to thread-local operations and cannot be accessed by other threads.

This hierarchy balances capacity, speed, and scope. Global memory provides large storage but higher latency, shared memory accelerates intra-block communication, and private memory delivers rapid access for thread-specific tasks.

4.2 Example

For example, consider the program for window averaging:

```
for i in range(len(input) - 2):
    output[i] = (input[i] + input[i + 1] + input[i + 1])/3.0
```

How can this be parallelized? Since every 3-element tuple reduction is independent, each reduction can be mapped to a CUDA core. So, each thread can compute the result for one element in the output array.

The host code:

```
int N = 1024*1024;
cudaMalloc(&devInput, sizeof(float)*(N+2)); // To account for edge conditions
cudaMalloc(&devOutput, sizeof(float)*N);

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

The device code:

```
#define THREADS_PER_BLK = 128

__global__ void convolve(int N, float* input, float* output) {
    int index = blockIdx.x *blockDim.x + threadIdx.x;
    float result = 0.0f; //thread-local variable
    result = input[index] + input[index + 1] + input[index + 2];
    output[index] = result /3.f;
}
```

This program can be optimized - each element is read thrice! Notice that the number of blocks assigned is much more than what a typical GPU has. This is a general practice in CUDA programming where the blocks are *oversubscribed*.

4.3 Shared memory

How to optimize? The memory hierarchy can be utilized.

The new device code:

```
#define THREADS_PER_BLK = 128

__global__ void convolve(int N, float* input, float* output) {
    int index = blockIdx.x *blockDim.x + threadIdx.x;

    __shared__ float support[THREADS_PER_BLK];
    support[threadIdx.x] = input[index];
    if(threadIdx.x < 2){
        support[THREADS_PER_BLK + threadIdx.x] = input[index + THREADS_PER_BLK];
    }

    __syncthreads();

    float result = 0.0f; //thread-local variable
    for(int i=0; i<3; i++)
        result += support[threadIdx.x + i];

    output[index] = result /3.f;
}
```

We introduced a synchronization primitive here. `__syncthreads()` waits for all threads in a block to arrive at this point. Another primitive `cudasynchronize()` that syncs between host and the device.

The code before `__syncthreads()` is using all the threads inside a threadblock to read data from the global "input" and store it into "support" which is shared among the threadblock. After a thread finishes reading its data, it needs to synchronize with the other threads before continuing computation.

By using shared memory, we can decrease number of memory operations. In this case, we use 128 threads to calculate 128 outputs which only cost us $128 + 2 = 130$ memory operations. However, without using shared memory, we need $128 \times 3 = 384$ memory operations to generate the same 128 results.

5 CUDA Compilation

A CUDA program also needs to be converted to low-level instructions that a processor can understand. A compiled CUDA device binary includes:

- Program text (instructions)
- Information about required resources. For example, a program might require - 128 threads per block, 8 types of local data per thread and 130 floats (520 bytes) of shared space per thread block.

Issue: Resource Management - An important issue is that different GPUs have different SMs or number of threads. This means that the amount of resources available differs between GPUs. If the user asks for a static (large) number of blocks, how can we handle this?

The first solution is that GPUs can have more (but ultimately limited) blocks to address demand. However, the number of blocks needed can usually be much greater than what any GPU today can allocate at once.

Solution: Dynamic Scheduling - A more feasible solution is that CUDA schedules the thread blocks to many cores (SMs), using a dynamic scheduling policy that respects the resource requirements. It assumes that the thread blocks are not dependent on each other, and can be executed in any order.

The scheduler works as follows: blocks are assigned according to the available resources, and the remaining ones are *queued*. Whenever a scheduled block is finished, the next block in the queue is scheduled in the finished block's place. The scheduler is dynamic because it can assign blocks to cores depending on the current situation, rather than working with a fixed, predetermined schedule.

6 Understanding a GPU

Consider a NVIDIA GTX 980 (2014) that has the following specs:

- 96KB of shared memory
- 16 SMs
- 2048 threads/SM
- 128 CUDA cores/SM

Note that the number of CUDA cores is not equal to the number of CUDA threads.

As the GPUs became better, NVIDIA tried to increase the shared memory per SM. This is similar to the SRAM which is very important for LLM inference. In particular, when NVIDIA GPUs had kilobytes of SRAM memory, a company called **Groq** put 230MB of SRAM per chip and obtained 3x higher performance than other GPUs. So, all these components are very important for GPU performance.

7 NVIDIA GPUs

NVIDIA is known to release a number of GPUs over time to cater to the growing demands for ML and AI applications. The earliest GPU: K80 GPU was released back in 2015 by NVIDIA. It then released P100 and P4 GPUS in 2016 and 2017 to signal advancements in computation power and energy efficiency tailored for ML tasks. V100 and T4 GPUs were then released in 2018-2019 where V100, which is based on NVIDIA's Volta architecture, became a flagship product for high performance ML.

The release of A100 GPU in 2020 set a new benchmark for large scale AI and ML training. It was based on the Ampere Architecture and performed well on inference tasks as well. In 2023, H400 and L4 GPUs with the Hopper Architecture representing the cutting edge of ML performance. These GPUs cater to:

- Generative AI
- Large Language Models
- Compute heavy applications

NVIDIA's journey over the past decade shows how much ML GPU technology has improved, mainly because of the growing need for GPUs that are faster, more scalable, and better suited for AI tasks.

7.1 Features of NVIDIA GPUs

NVIDIA has consistently been improving its GPUs to meet the increasing demands of machine learning and AI applications. Below is a comparison of some of their GPUs, including their key features and AWS on-demand pricing:

- **V100 (2018 - Present):** The V100 comes with 80 Streaming Multiprocessors (SMs), each supporting 2048 threads. It's been widely used for machine learning and AI workloads due to its solid performance. On AWS, the on-demand cost for the V100 is approximately **$3/hour**.
- **A100 (2020 - Present):** With 108 SMs and 2048 threads per SM, the A100 marked a significant improvement in performance and efficiency. It's ideal for handling more demanding AI tasks. The AWS on-demand cost for the A100 is about **$4/hour**.
- **H100 (2022 - Present):** The H100 represents the cutting edge in AI GPUs, featuring 144 SMs and 2048 threads per SM. Its performance is unmatched for large-scale AI and deep learning tasks. However, this level of performance comes at a higher cost of **$12/hour** on AWS.
- **B100 and B200 (2025 - Future):** These GPUs are part of NVIDIA's highly anticipated Blackwell architecture. NVIDIA's **Blackwell architecture supports models up to 740 billion parameters in size**, which is six times larger than what Hopper architecture could manage. This massive increase supports the development and operation of larger and more complex AI models. Although NVIDIA has not yet disclosed the exact number of SMs and threads per SM, early projections suggest that the B100 and B200 will provide significant advancements in computational performance for AI and machine learning workloads. Pricing details for AWS on-demand usage have not been officially announced. However, given their cutting-edge capabilities, they are expected to be priced higher than the H100. Industry estimates place the retail price of the B100 between **$30,000 and $35,000**, with the B200 likely commanding an even higher price point due to its enhanced features and performance.

Contributions

- Sudhansh Peddabomma: Provided the initial skeletal notes for all sections, code and math equations, and added section 1: Recap. Proofread and corrected typos in the document.
- Brandon Hsu: Added and revised content in MCQ. Proofread and corrected typos and formatting errors in the document.
- Melvyn Tan: Added and updated content in Section 2: GPU and CUDA.
- Allen Lu: Added and updated content in Section 2.1: CUDA and Section 3.1: Matrix Add
- Kuber Shahi: Added and revised content in section 3.2: Handling Control Flow. Proofread and revised content in sections 2.1, 3.1, and 3.2.
- Keivan Rahmani: Added and revised content in Section 3
- Nai-En Kuo: Added and updated content in Section 4.1: Shared memory.
- Vincent Thai: Added and revised content in Section 6: CUDA Compilation
- Ketaki Tank: Added and updated content in Section 7: NVIDIA GPUs and features.
- Yi Li: Added and updated content in Section 4.2, 4.3: CUDA Memory Model Example
- Sarthak Kala: Added and updated content in section 4.1: CUDA Memory Model Introduction