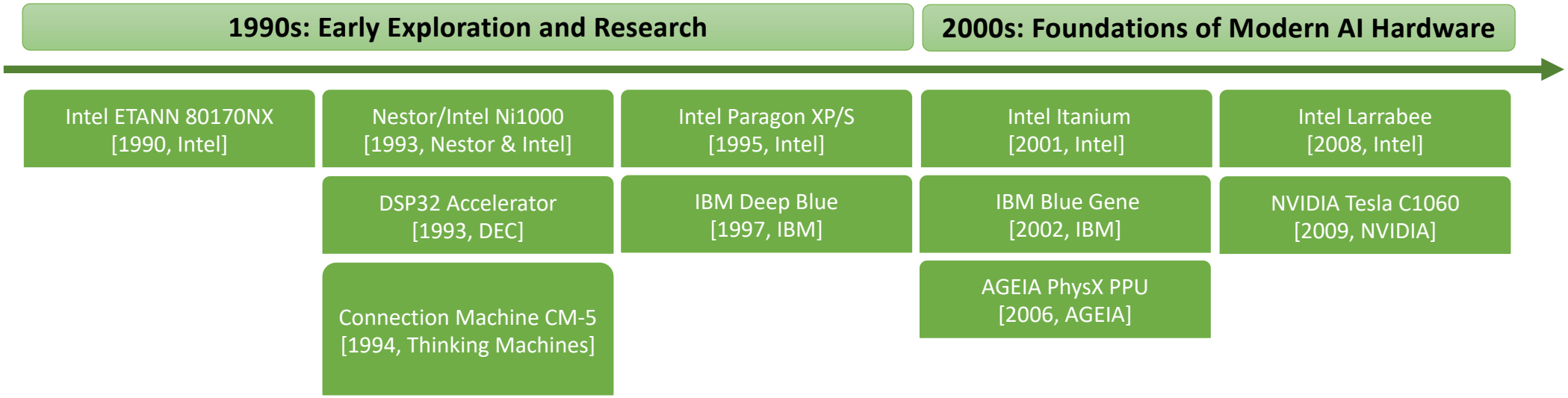
THE UNIVERSITY of EDINBURGH
informatics

Modern GPU Programming

Speaker: Yeqi Huang, Luo Mai

A brief history of AI accelerators (I): From general parallel to specific task



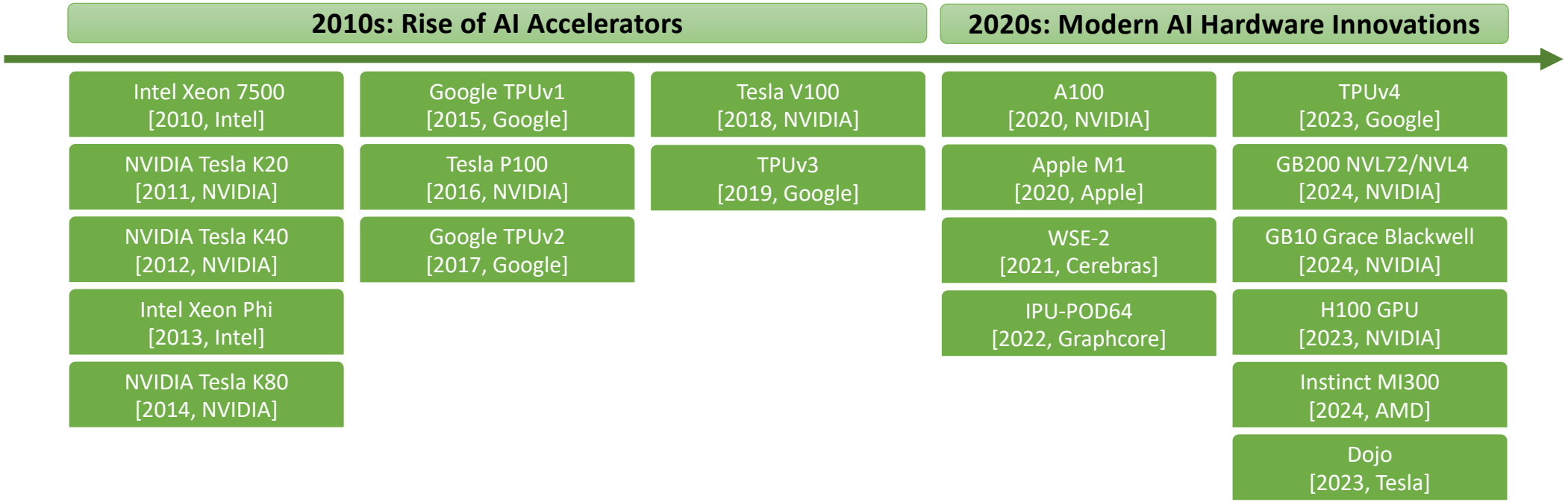
1990s: Early Exploration and Research

- **Focus on specialized processors** for AI tasks, transitioning from analog to digital.
- Introduction of **DSPs (Digital Signal Processors)** like Intel ETANN for neural network computations.
- Early **parallel computing** efforts with supercomputers like Connection Machine CM-5 and IBM Deep Blue for AI.

2000s: Foundations of Modern AI Hardware

- Emergence of **high-performance computing systems** (e.g., Intel Itanium, IBM Blue Gene) for AI simulations.
- Introduction of **general-purpose GPUs** like NVIDIA's Tesla C1060, marking the start of GPU involvement in AI.

A brief history of AI accelerators (II): Blooming of new architectures.



2010s: Rise of AI Accelerators

- Shift toward dedicated AI accelerators like Google's Tensor Processing Units (TPUs) for deep learning tasks.
- NVIDIA's GPUs (e.g., Tesla K80, P100) gaining prominence for AI and machine learning acceleration.
- Multi-core processors and parallel processing became key features for AI workloads (e.g., Intel Xeon Phi, NVIDIA Tesla series).

2020s: Modern AI Hardware Innovations

- There has been a significant increase in custom AI chips, including the Cerebras WSE-2 and Graphcore, which provide massive bandwidth.
- Custom hardware has also been developed to align with the features of AI models, such as Apple silicon and NVIDIA L20.
- As the capabilities of individual chips reach their limits, efforts are being directed towards improving strategies for chip connectivity.

Common Features of AI Accelerators

Key Trends

- Transition from general-purpose chips to specialized AI chips (FPGA/ASIC/ GPU).
- Focus on parallelism and scalability for complex AI tasks.
- Rise of energy-efficient AI accelerators.
- Increasing role of edge AI with chips designed for real-time, low-latency tasks.

Common Features of AI acclerators

Design computing units with high parallelism for specific tasks

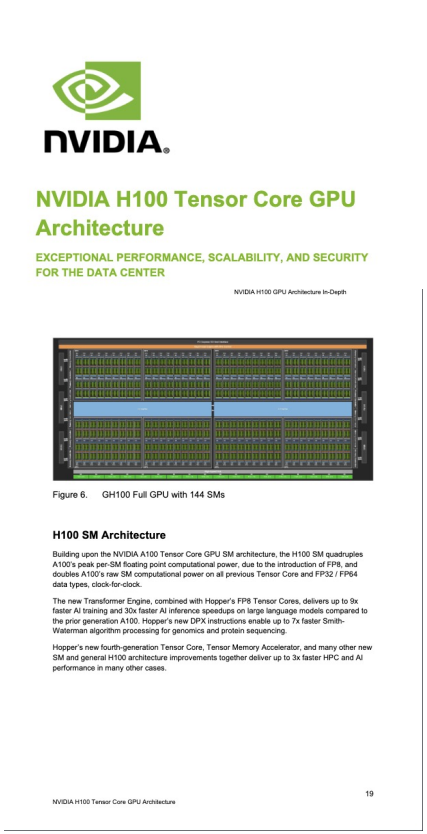
We use NVIDIA GPUs in this course for 3 reasons.

- GPUs are **suitable** for AI tasks
  - **Parallel Processing, Efficient Data Handling, Optimized for AI Libraries**
- NVIDIA GPUs are more **accessible**
  - Recent Market Share (2024):
    - NVIDIA holds a commanding 90% of the discrete GPU market share
    - AMD accounts for the remaining 10%
    - Intel has effectively dropped to 0% market share
- **User-friendly libraries** in the NVIDIA GPU ecosystem
  - Most AI chips are not programmable for common users, including the Intel Iris GPU, Apple A18 chip, and Qualcomm Snapdragon X series
  - NVIDIA CUDA has numerous related resources, and I will include some of them in the extra resources at the end of the slides

https://pcviewed.com/nvidia-vs-amd-discrete-gpu-market-share/

Deep Dive into Whitepaper

- H100 GPU Whitepaper: [NVIDIA H100 Tensor Core GPU Architecture Overview]



NVIDIA

NVIDIA H100 Tensor Core GPU Architecture

EXCEPTIONAL PERFORMANCE, SCALABILITY, AND SECURITY FOR THE DATA CENTER

NVIDIA H100 GPU Architecture In-Depth

Figure 6. GH100 Full GPU with 144 SMs

H100 SM Architecture

Building upon the NVIDIA A100 Tensor Core GPU SM architecture, the H100 SM quadruples A100's peak per-SM floating point computational power, due to the introduction of FP8, and doubles A100's raw SM computational power on all previous Tensor Core and FP32 / FP64 data types, clock-for-clock.

The new Transformer Engine, combined with Hopper's FP8 Tensor Cores, delivers up to 9x faster AI training and 30x faster AI inference speedups on large language models compared to the prior generation A100. Hopper's new DPX instructions enable up to 7x faster Smith-Waterman algorithm processing for genomics and protein sequencing.

Hopper's new fourth-generation Tensor Core, Tensor Memory Accelerator, and many other new SM and general H100 architecture improvements together deliver up to 3x faster HPC and AI performance in many other cases.

19

NVIDIA H100 Tensor Core GPU Architecture

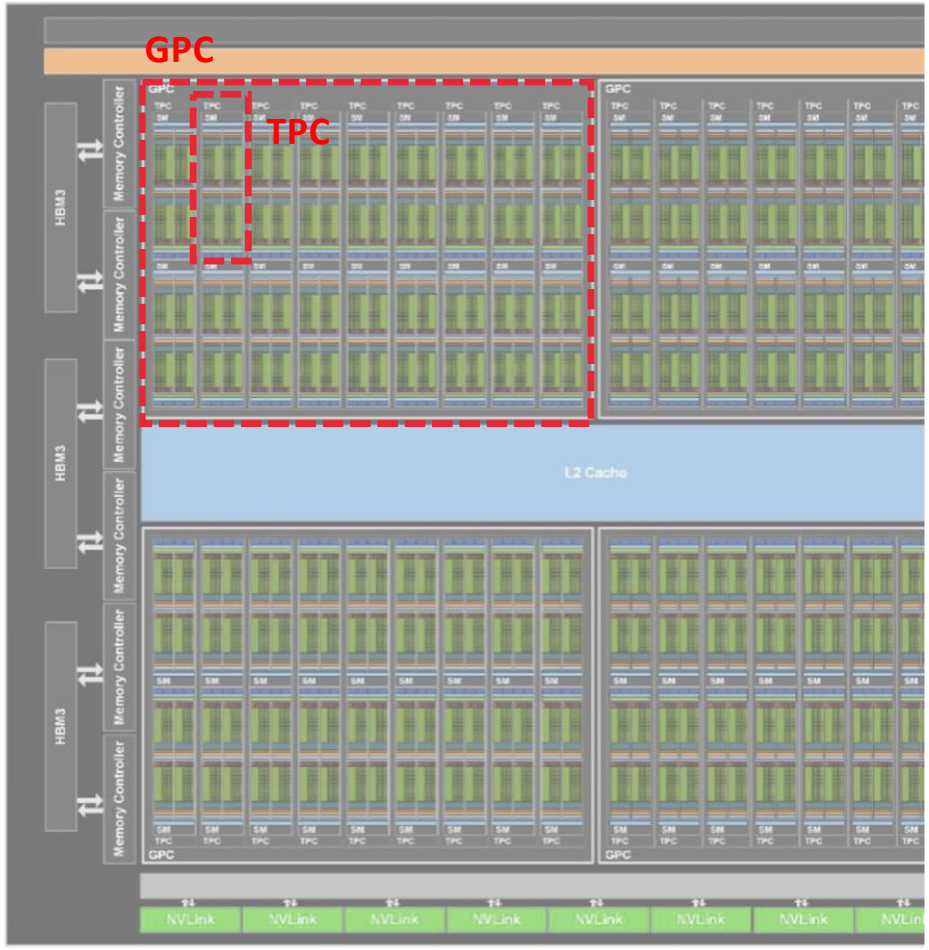
Deep Dive into Whitepaper

The full implementation of the GH100 GPU includes the following units:

- 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, **144 SMs per full GPU**
- 128 FP32 CUDA Cores per SM, **18432 FP32 CUDA Cores per full GPU**
- 4 Fourth-Generation **Tensor Cores** per SM, **576 per full GPU**
- **6 HBM3 or HBM2e stacks**, 12 512-bit Memory Controllers
- **60 MB L2 Cache**
- **Fourth-Generation NVLink and PCIe Gen 5**

You may be confused about those components, lets find them in the whitepaper.

Deep Dive into Whitepaper



The NVIDIA GH100 GPU is composed of multiple **GPU Processing Clusters (GPCs), Texture Processing Clusters (TPCs)**, Streaming Multiprocessors (SMs), L2 cache, and HBM3 memory controllers.

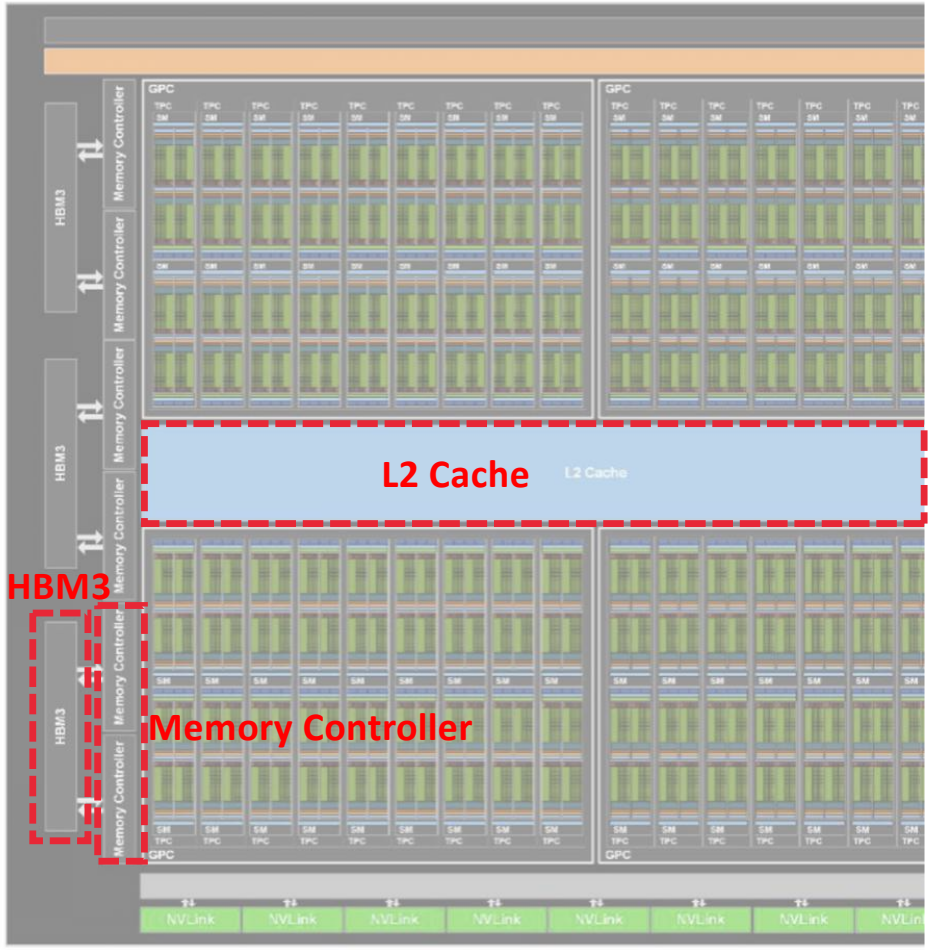
Modular Growth: By organizing the GPU into GPCs and TPCs, manufacturers can scale performance by adding more clusters. For instance, a GPU can have multiple GPCs, each containing several TPCs, allowing for increased parallelism without redesigning the entire architecture.

E.G.:
GH100 GPU:
- 8 GPCs, 72 TPCs (9 TPCs/GPC), 144 SMs per full GPU

H100 GPU with SXM5 board form-factor:
- 8 GPCs, 66 TPCs, 132 SMs per GPU

H100 GPU with a PCIe Gen 5 board form-factor
- 7 or 8 GPCs, 57 TPCs, 2 SMs/TPC, 114 SMs per GPU

Deep Dive into Whitepaper



HBM3 stands for **High Bandwidth Memory 3**, the third generation of HBM technology. HBM is a type of high-speed RAM used in GPUs and other high-performance computing applications.

Bandwidth Demands: Modern applications, especially those involving complex graphics rendering, AI training, and scientific computations, require extremely high memory bandwidth to feed data quickly to GPU cores. HBM3 meets these demands by offering unparalleled data transfer rates.

Memory Controllers are specialized circuits within the GPU that manage the flow of data between the GPU's processing units (like CUDA cores or shader units) and its memory subsystem (such as HBM3 stacks).

In short, they are used to copy data between the GPU and *external* devices (another GPU, DRAM, or other devices).

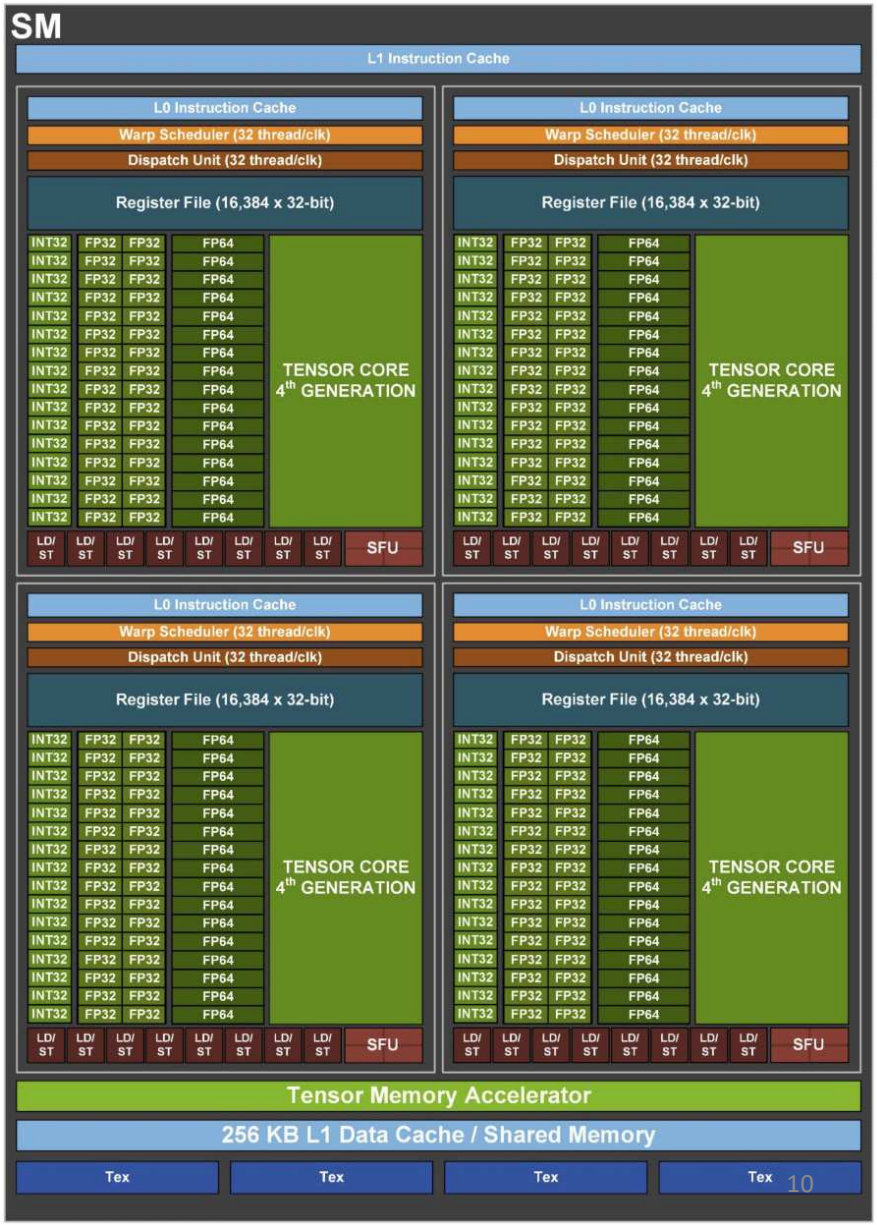
SM (Streaming Multiprocessor) Overview

When we program GPUs, we produce sequences of instructions for their **Streaming Multiprocessors to carry out**.

You can find many components in this large figure like:

- L1 Cache
- SFU
- LSU
- CUDA Core
- Tensor Core

We will introduce them one by one.



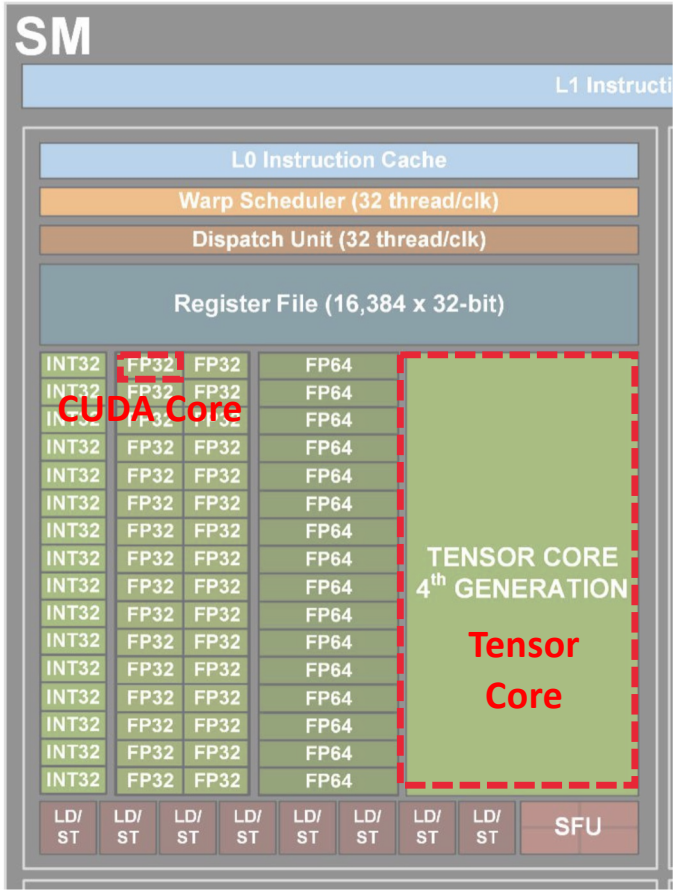
SM (Streaming Multiprocessor)

In every SM

- 128 FP32 **CUDA Cores**
- 4 Fourth-Generation **Tensor Cores**

CUDA Cores are GPU cores designed to execute scalar arithmetic instructions.

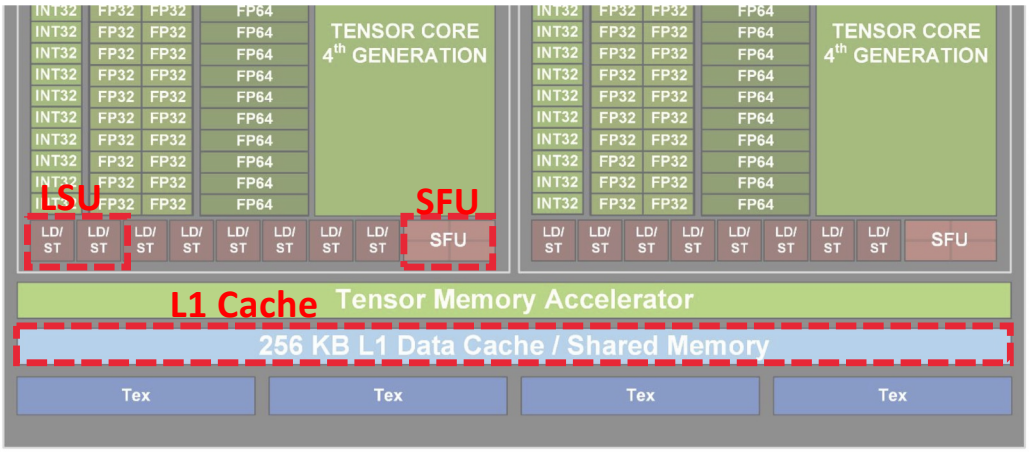
Tensor Cores are GPU cores that work on entire matrices with each instruction.



SM (Streaming Multiprocessor)

Special Function Units (SFUs) accelerate certain arithmetic operations, such as exp, sin, cos.

The Load/Store Units (LSUs) dispatch requests to load or store data to the memory subsystems of the GPU.



The L1 data cache is the private memory of the Streaming Multiprocessor (SM).

Let's focus on a simplified GPU hardware model

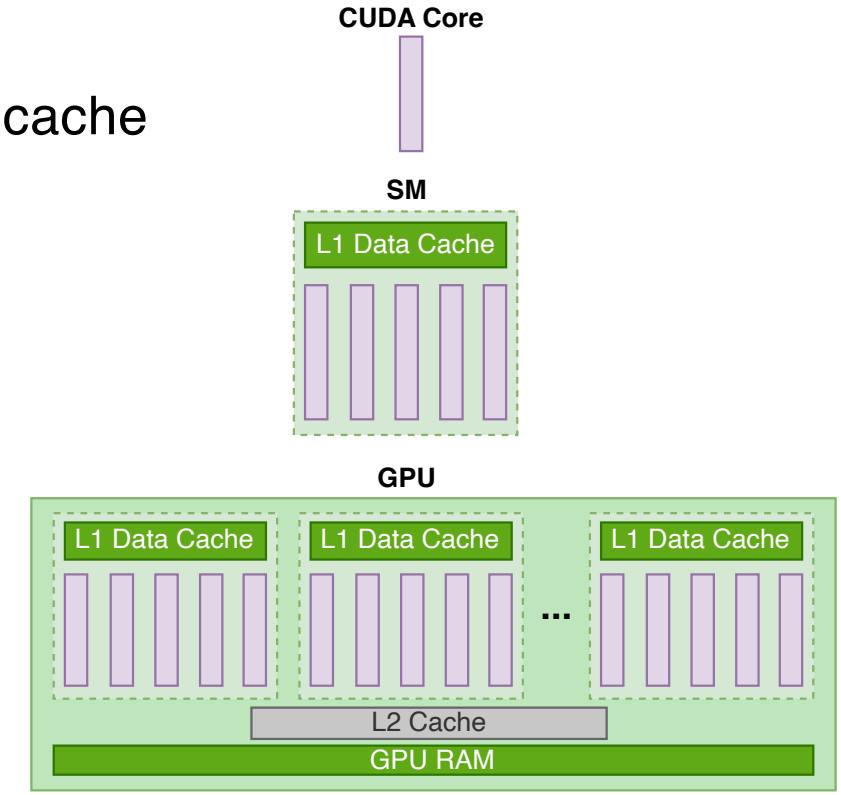
- Streaming Processor (SP) ≈ several cuda cores
- Streaming Multiprocessor (SM) ≈ multiple SPs + L1 cache
- GPU ≈ multiple SMs + L2 cache + RAM

Notice: In Fermi or some old architectures:

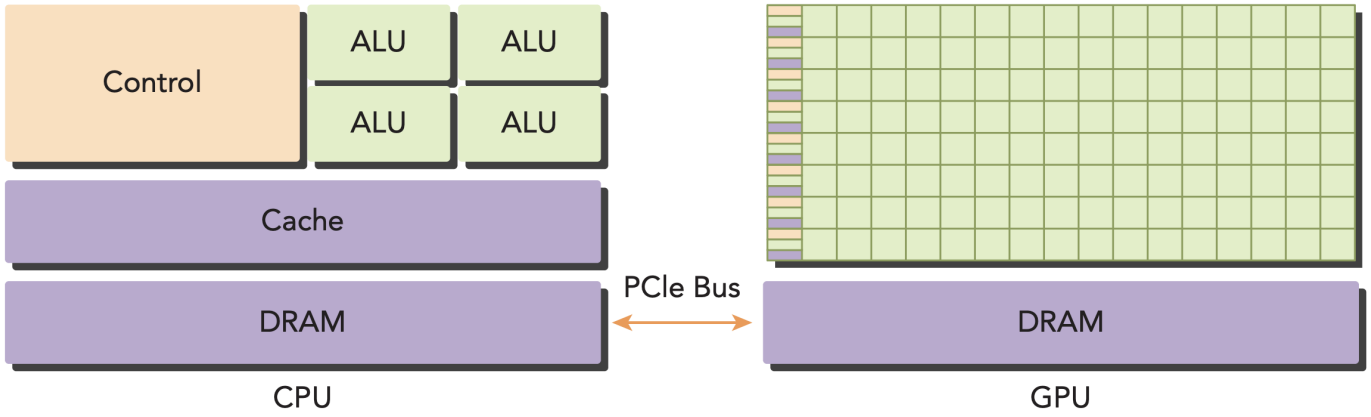
SP = 1 CUDA core.

In modern architectures like Kepler, Maxwell, Ampere, or Hopper,

SP = several CUDA cores + several tensor cores.



Why parallel on GPU not CPU?

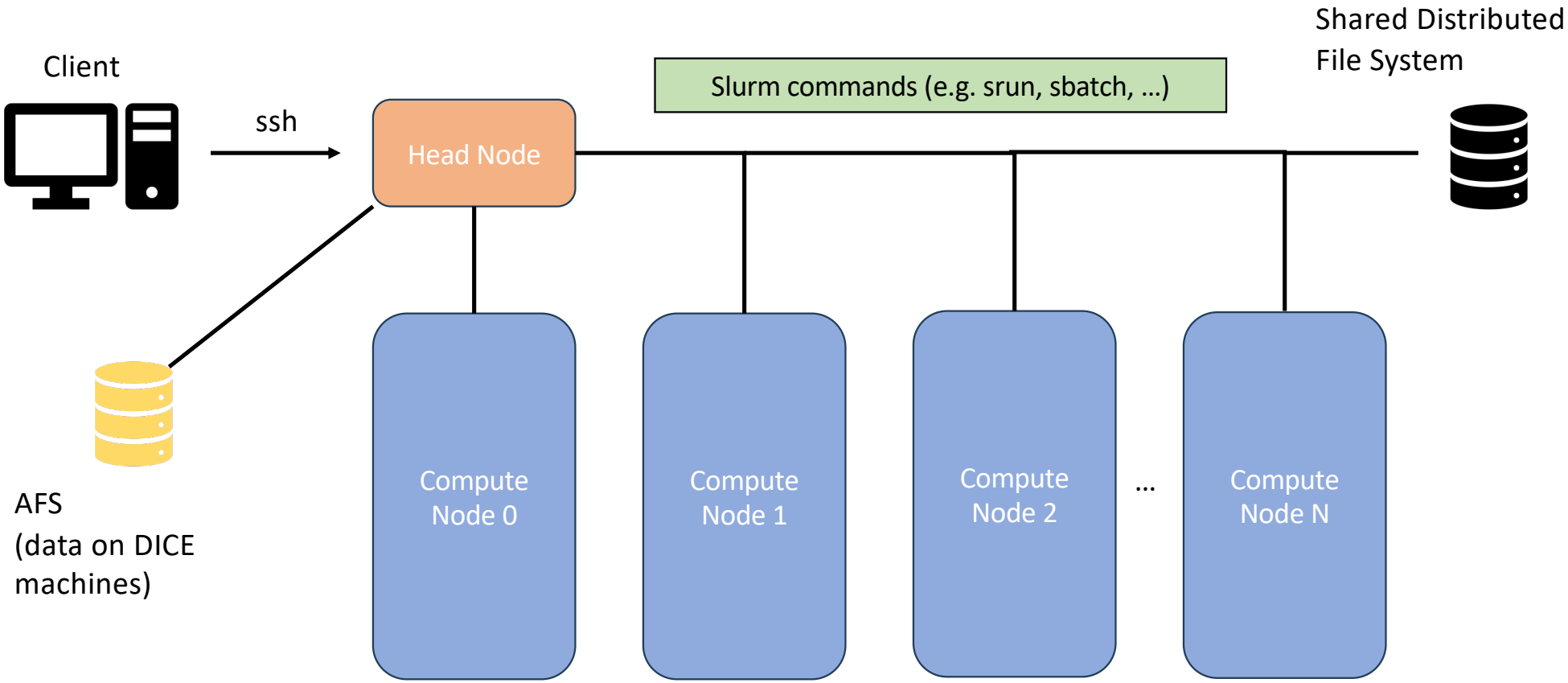


- GPU SMs can execute more threads in parallel.
- For comparison, an AMD EPYC 9965 CPU draws at most 500 W and has 192 cores, each of which can execute instructions for at most two threads at a time, for a total of **384** threads in parallel, running at about **1.25 W per thread**.
- An H100 SXM GPU draws at most 700 W and has 132 SMs, each of which has four Warp Schedulers that can each issue instructions to 32 threads (aka a warp) in parallel per clock cycle, for a total of **128 × 132 = 16,896 parallel threads** running at about **0.05 W** apiece. Note that this is truly parallel: each of the 17,000 threads can make progress with each clock cycle.

Example from https://modal.com/gpu-glossary/device-hardware/streaming-multiprocessor

Teaching GPU Cluster

Teaching Cluster Overview



Teaching GPU Cluster

Basic Info:

The teaching cluster has over 120 GPUs in 20 servers (landonia[01-25]) with:

- 100 NVIDIA GTX 1060 6GB GPUs,
- 8 NVIDIA RTX A6000 48GB GPUs,
- a small number of NVIDIA TITAN-X 12GB GPUs.

Usage:

Access through **ssh** and use **SLURM** commands.

1. Access the cluster's head node via:

   - Connect to the Informatics VPN or use the DICE machine and then run: `ssh <YOUR_UUN>@mlp.inf.ed.ac.uk`
   - You can remotely connect to your DICE machine using the command **below** if you are connected to the Informatics VPN or the eduroam Wi-Fi. Once you have accessed the DICE machine, run the command mentioned **above**.

     `ssh <YOUR_UUN>@ssh.inf.ed.ac.uk`

   - You can find information about how to connect to Informatics VPN at: https://computing.help.inf.ed.ac.uk/openvpn

Teaching GPU Cluster

Usage (continue):

After successfully connecting to the head node, your terminal will display the following:

```
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-130-generic x86_64)

This is a cluster head node please do not run compute intensive processes here
this node is intended to provide an interface to the cluster only
Please nice any long running processes
------------------------------------------------------------------------------
Looking for tips?  https://auth.computing.help.inf.ed.ac.uk/cluster-tips
Jan 2025: Note that some GRES have changed to swap dashes to underscores.


Last login: Mon Jan 13 15:45:11 2025 from openvpn-125-012.inf.ed.ac.uk
[uhtred]s2020153:
```

2. Run your code with SLURM command:

There are two ways you can run your code:

- Interactive job - allow you to interact directly with the allocated compute resources, good for debugging and testing code.
- Batch job - run the task automatically based on a pre-written script (.sh file) without interaction with the compute node and constant user attention, good for long-running tasks.

Teaching GPU Cluster

Usage (continue):

E.g. Let's say you want to run the code utilizing 1 GPU.

Interactive jobs:

```
srun -p Teaching -w saxa --gres gpu:1 --pty bash
```

After executing this command, run “nvidia-smi”, and you will see output similar to the following:

```
Thu Jan 22 04:24:57 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 570.195.03             Driver Version: 570.195.03     CUDA Version: 12.8     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA H200                    On  |   00000000:19:00.0 Off |                    On |
| N/A   30C    P0             75W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA H200                    On  |   00000000:3B:00.0 Off |                    On |
| N/A   30C    P0             77W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   2  NVIDIA H200                    On  |   00000000:4C:00.0 Off |                    On |
| N/A   28C    P0             78W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   3  NVIDIA H200                    On  |   00000000:5D:00.0 Off |                    On |
| N/A   29C    P0             76W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   4  NVIDIA H200                    On  |   00000000:9B:00.0 Off |                    On |
| N/A   30C    P0             76W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   5  NVIDIA H200                    On  |   00000000:BB:00.0 Off |                    On |
| N/A   29C    P0             78W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   6  NVIDIA H200                    On  |   00000000:CB:00.0 Off |                    On |
| N/A   30C    P0             75W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+
|   7  NVIDIA H200                    On  |   00000000:DB:00.0 Off |                    On |
| N/A   27C    P0             77W /  700W |      60MiB / 143771MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| MIG devices:                                                                            |
+------------------+--------------------------------+-----------+-----------------------+
| GPU  GI  CI  MIG |                    Memory-Usage |        Vol|         Shared        |
|      ID  ID  Dev |                      BAR1-Usage | SM     Unc| CE ENC  DEC  OFA  JPG |
|                  |                                |        ECC|                       |
|==================+================================+===========+=======================|
|  0    7   0   0  |              9MiB / 16384MiB    | 16      0 |  1   0    1    0    1 |
|                  |                0MiB / 32767MiB  |           |                       |
+------------------+--------------------------------+-----------+-----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Teaching GPU Cluster

Usage (continue):

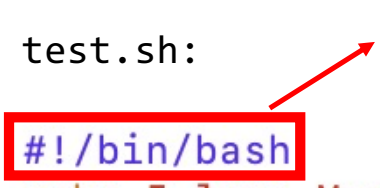
E.g. Let's say you want to run the code utilizing 1 GPU.

Batch jobs: `sbatch --gres=gpu:1 test.sh`

After executing this command, you will see "Submitted batch job [ID]".

The output file will be saved as `slurm-[ID].out` by default in the directory where you call sbatch. You can set the output directory and file names by using the `--output` option with the `sbatch` command.

test.sh:




```
#!/bin/bash
echo I love Machine Learning Systems.
pwd
```

```
[uhtred]s2020153: sbatch --gres gpu:1 test.sh
Submitted batch job 1945143
[uhtred]s2020153: cat slurm-1945143.out
I love Machine Learning Systems.
/home/s2020153
```

Teaching GPU Cluster

Usage (continue):

Since servers are equipped with different types of GPUs, you can specify the type of GPU you wish to use:

E.g. You want to have 1 NVIDIA Titan X GPU:

```
srun --gres=gpu:titan_x:1 --pty bash
sbatch --gres=gpu:titan_x:1 test.sh
```

```
[uhtred]s2020153: srun --gres=gpu:titan_x:1 --pty bash
srun: job 1944884 queued and waiting for resources
srun: job 1944884 has been allocated resources
[landonia08]s2020153: nvidia-smi
Mon Jan 13 15:46:28 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.127.08             Driver Version: 550.127.08     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce GTX TITAN X     Off |   00000000:07:00.0 Off |                  N/A |
| 22%   27C    P8             16W /  250W |       0MiB /  12288MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

You are allowed to request a maximum of 8 GTX 1060/4 Titan X/1 Titan X Pascal GPU/2 A6000 GPUs at a time.
This year we have 8 NVIDIA H200 for you! (so I guess nobody want to use 1060 any more)
You can get access with `-p Teaching -w saxa` after srun/sbatch command.

NOTE: Please allocate resources according to your specific requirements. Avoid over-allocating to ensure resources are available for others!

Teaching GPU Cluster

Usage (continue):

Other useful SLURM commands:

Check all available GPU types: `scontrol show node | grep gpu`

```
Gres=gpu:titan_x_pascal:1(S:0),gpu:titan_x:2(S:0),gpu:gtx_1060:2(S:1)
CfgTRES=cpu=12,mem=96000M,billing=12,gres/gpu=5
Gres=gpu:gtx_1060:8(S:0-1)
CfgTRES=cpu=12,mem=96000M,billing=12,gres/gpu=8
```

Check current SLURM job status: `squeue`

```
[uhtred]s2020153: squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           1563085 General_U collect_ s2263903 PD       0:00      1 (BadConstraints)
           1944830 PGR-Stand ilcc_pre s2148449 PD       0:00      1 (Resources)
```

Cancel a job: `scancel <job_id>`

For more details about the school cluster and running SLURM commands, please refer to: https://computing.help.inf.ed.ac.uk/teaching-cluster.

IMPORTANT: Please read the Files and Backups section and GPU cluster tips in this link carefully, as it contains crucial information about file storage and can help prevent data loss.

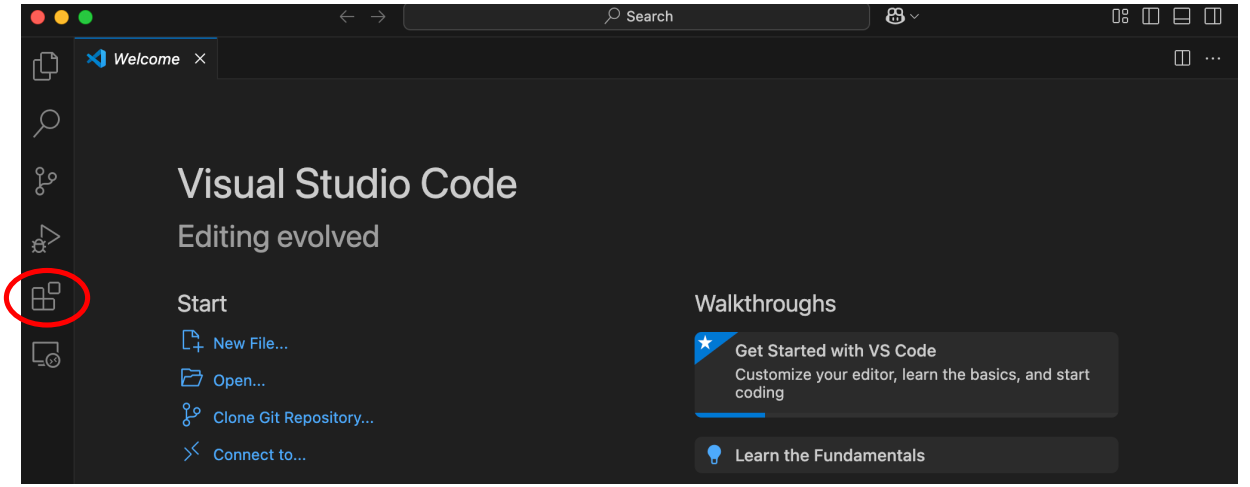
Teaching GPU Cluster

How to write codes in the teaching cluster?

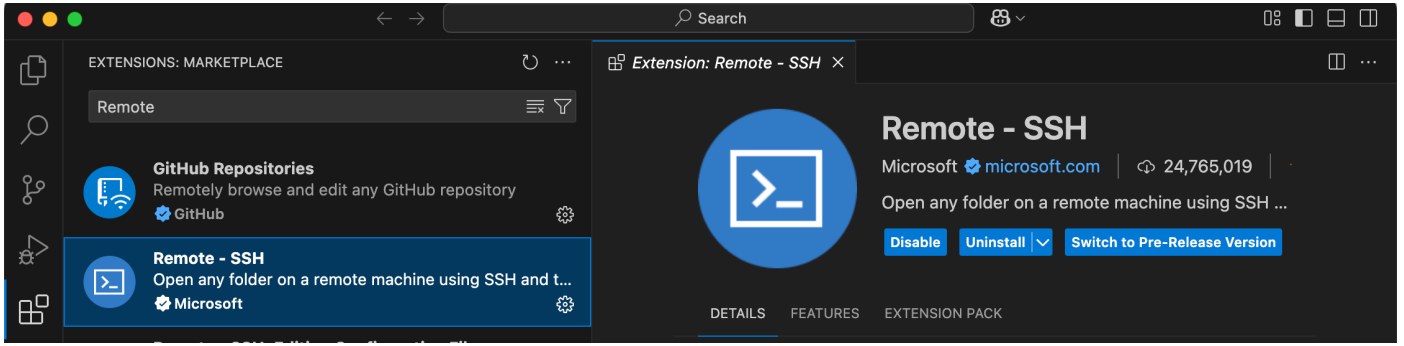
VSCode!

1. Open your VScode

2. Go to Extensions



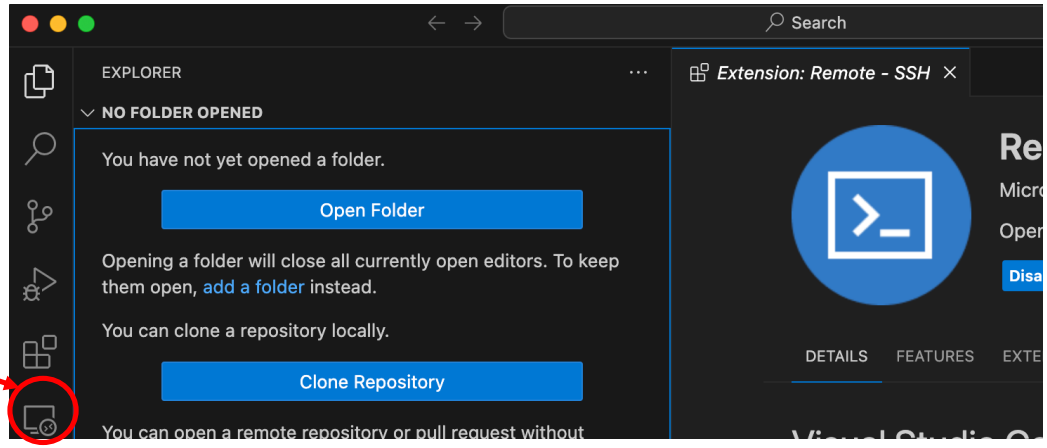
3. Search for the keyword "Remote" and install the "Remote - SSH" extension.



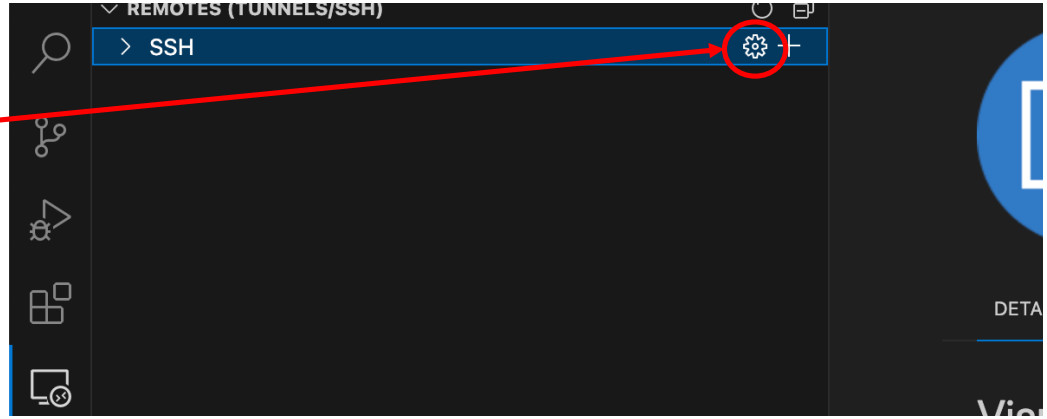
Teaching GPU Cluster

How to write codes in the teaching cluster? (Continue)

4. Once the extension is installed, the REMOTES icon will appear on the left sidebar.



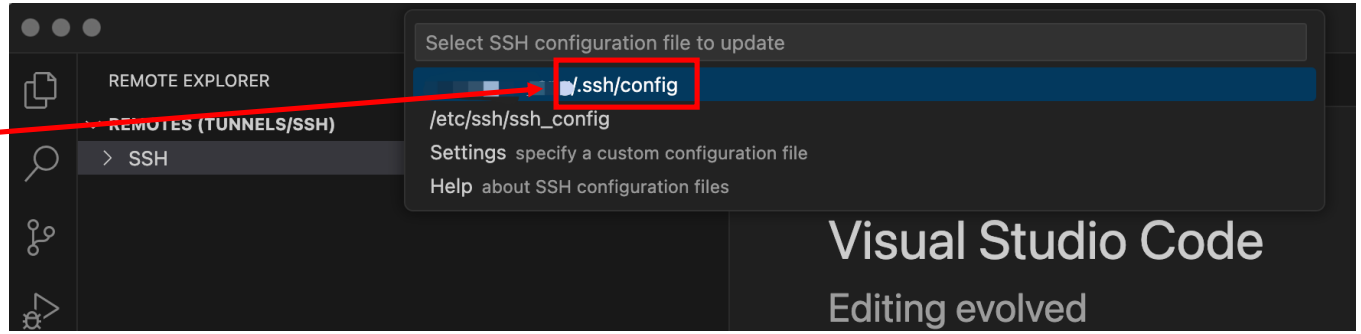
5. Click on the REMOTES icon, then select the Config icon located to the right of "SSH."



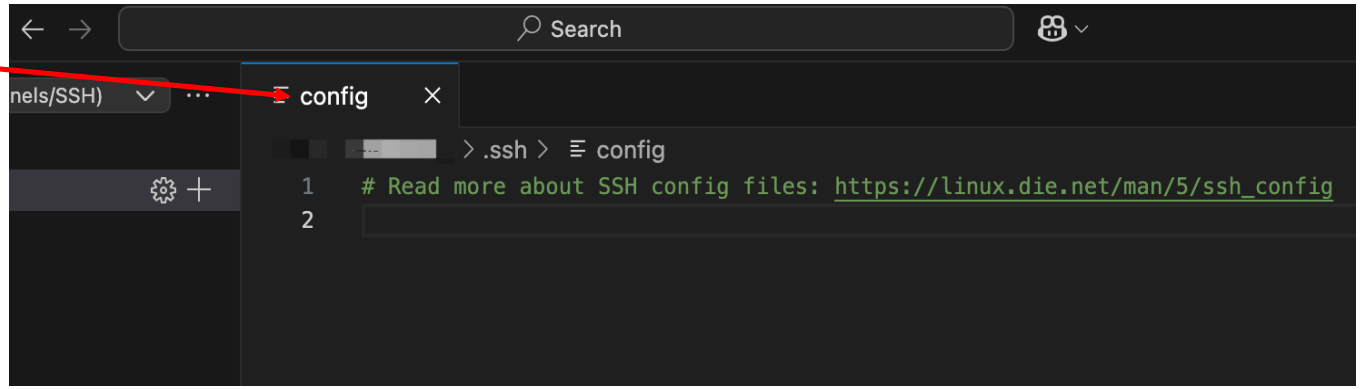
Teaching GPU Cluster

How to write codes in the teaching cluster? (Continue)

6. After clicking the Config icon, choose the configuration file with a path ending in .ssh/config.



7. You will see a page like this.



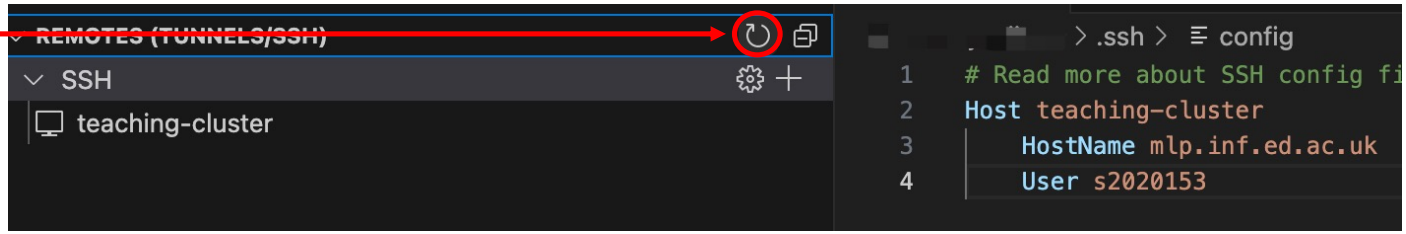
Teaching GPU Cluster

How to write codes in the teaching cluster? (Continue)

8. In this page, write the SSH configuration text and save it:

```
Host teaching-cluster
    HostName mlp.inf.ed.ac.uk
    User <YOUR_UUN>
```

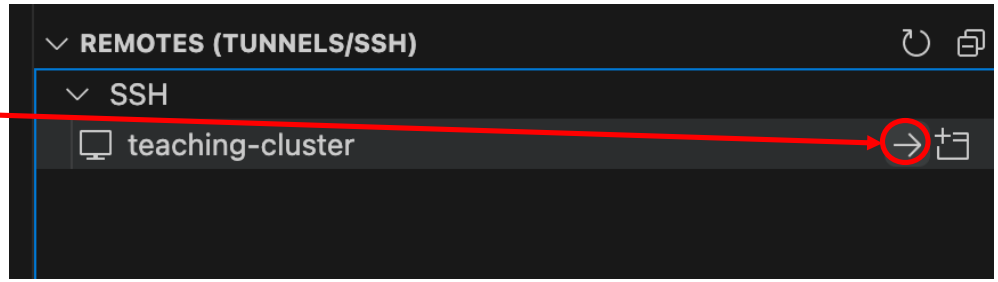
9. Refresh the REMOTES page, and a screen similar to this will appear.



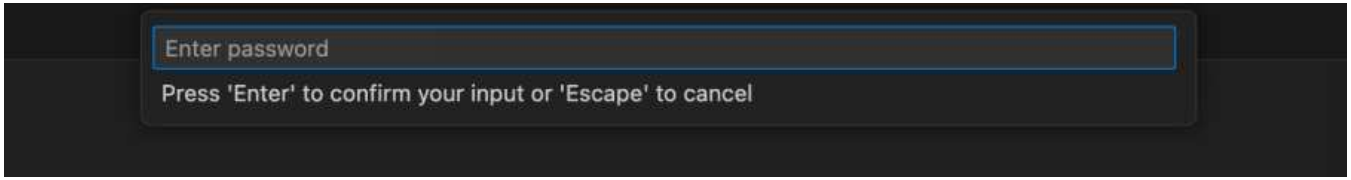
Teaching GPU Cluster

How to write codes in the teaching cluster? (Continue)

10. Connect to the Informatics VPN or Use the DICE machine, then click the right arrow icon to connect to the teaching cluster's head node:



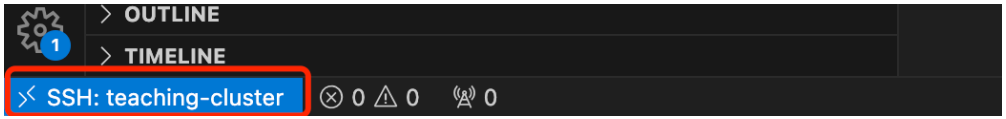
11. Enter the password of your DICE account



Teaching GPU Cluster

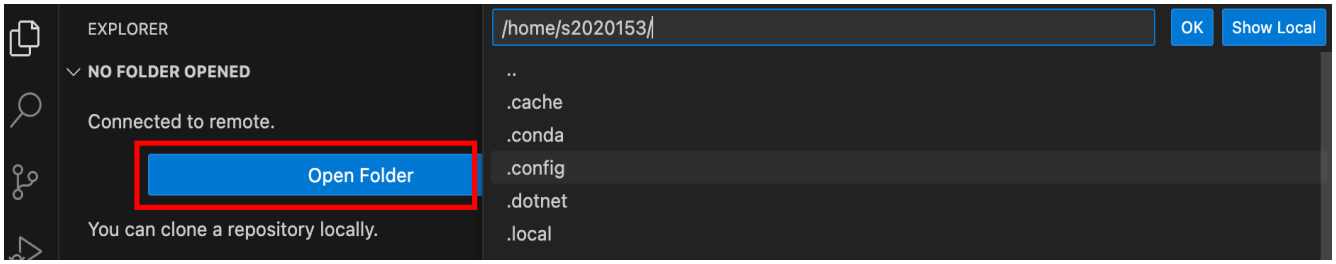
How to write codes in the teaching cluster? (Continue)

12. If this screen appears, congratulations! You have successfully connected to the teaching cluster's head node.



13. Go to your home directory by clicking ”Open Folder”, and select your home directory (/home/<YOUR_UUN>). Click OK.

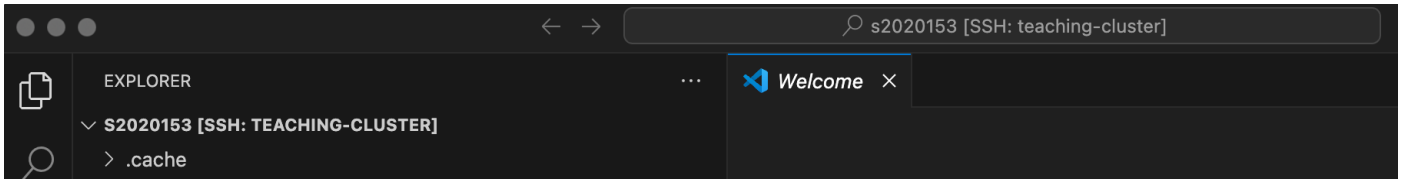
If password required, just type your password of the DICE account again.



Teaching GPU Cluster

How to write codes in the teaching cluster? (Continue)

14. If you can see this screen, you are now ready to write your code in the teaching cluster.



IMPORTANT NOTES:

1. Any changes you make in VSCode, such as writing code, creating or deleting files and folders, will automatically synchronize with the cluster. That's why we recommend it ☺. **Do not forget to save your changes**

2. Only install your environment and write your code on the head node. Actions such as Git operations and similar tasks should also be performed on the head node, as they will not work on the compute nodes.

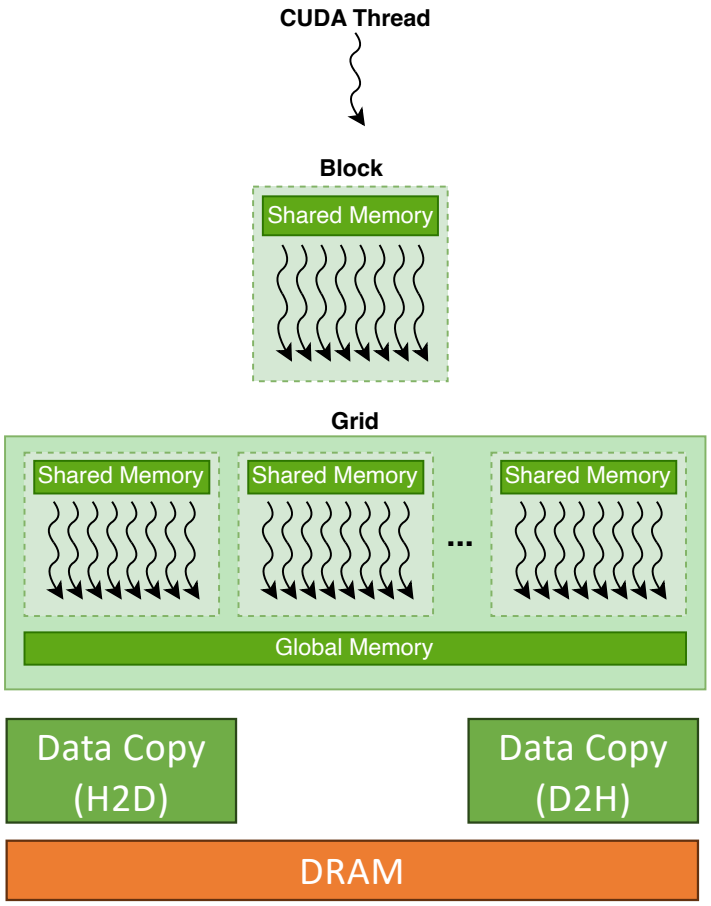
3. Only run your code on the compute nodes using `srun` or `sbatch`, as the head node has limited computing resources and does not have GPUs. Running `torch.cuda.is_available()` on the head node will return `False`.

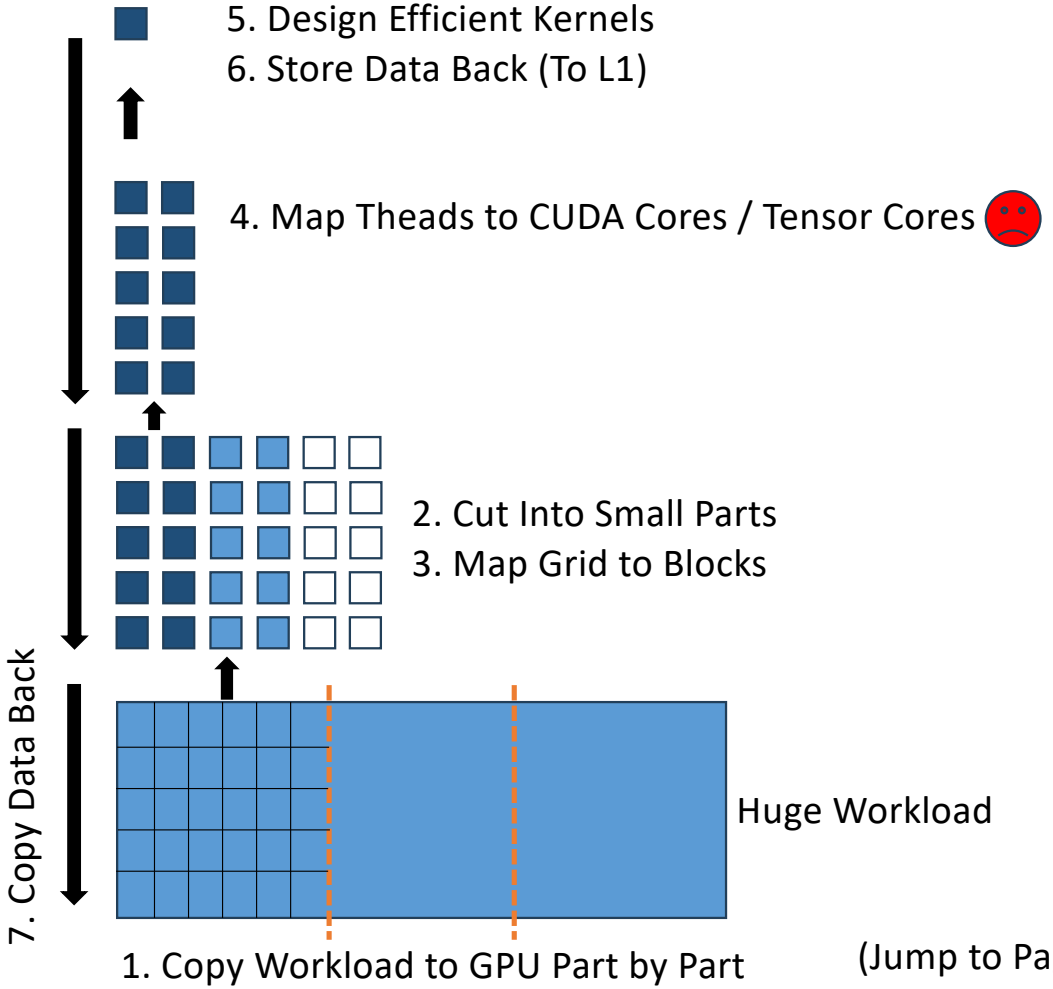
If you have further issues about accessing the teaching cluster, please contact the Computing Support Team: https://computing.help.inf.ed.ac.uk/

Let's start GPU programming.

- Choices are really complicated today.

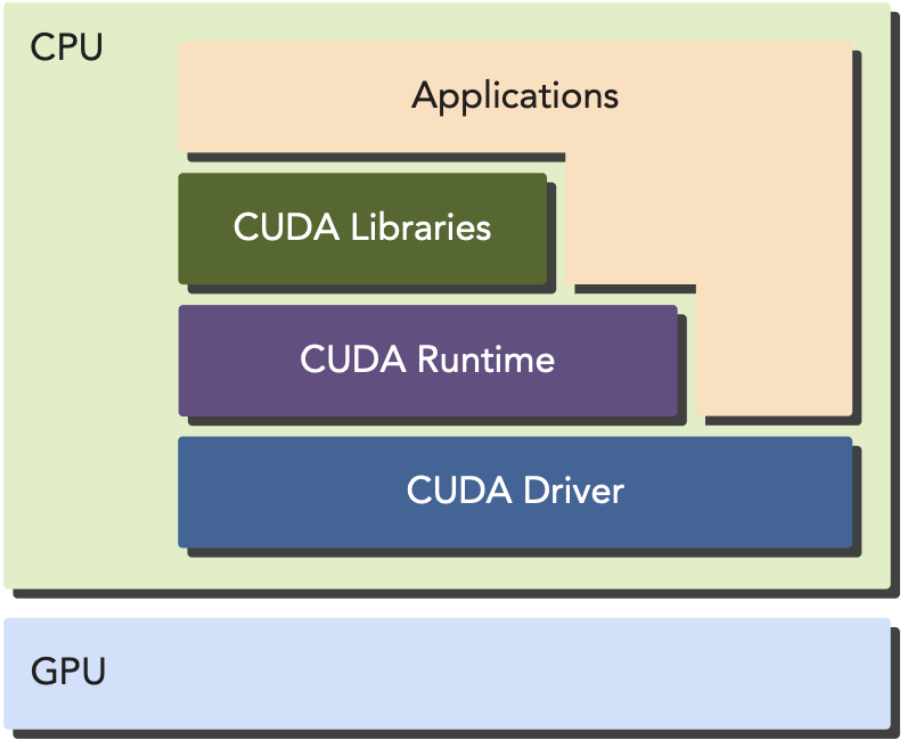
In the last year, students learned a tough lesson about **CUDA**.





(Jump to Page 38)

Understand Programming Model with CUDA.



CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform and programming model that enables dramatic performance increases in computing by utilizing NVIDIA GPUs (Graphics Processing Units). It allows developers to use GPU computing for general-purpose processing, accelerating computationally intensive applications in fields like scientific computing, AI, and data analysis.

NVIDIA's C++-like Programming Language

Quick Start with CUDA

```
__global__ void kernel() {
    printf("Block %d of %d, Thread %d of %d\n",
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x);
}

int main() {
    kernel<<<4, 3>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
Block 0 of 4, Thread 0 of 3
Block 0 of 4, Thread 1 of 3
Block 0 of 4, Thread 2 of 3
...
Block 1 of 4, Thread 1 of 3
Block 1 of 4, Thread 2 of 3
Block 3 of 4, Thread 0 of 3
Block 3 of 4, Thread 1 of 3
Block 3 of 4, Thread 2 of 3
```

- What is block?
- What is thread?
- What is <<<4, 3>>>
- What is cudaDeviceSynchronize();?

We will cover in the following slides.

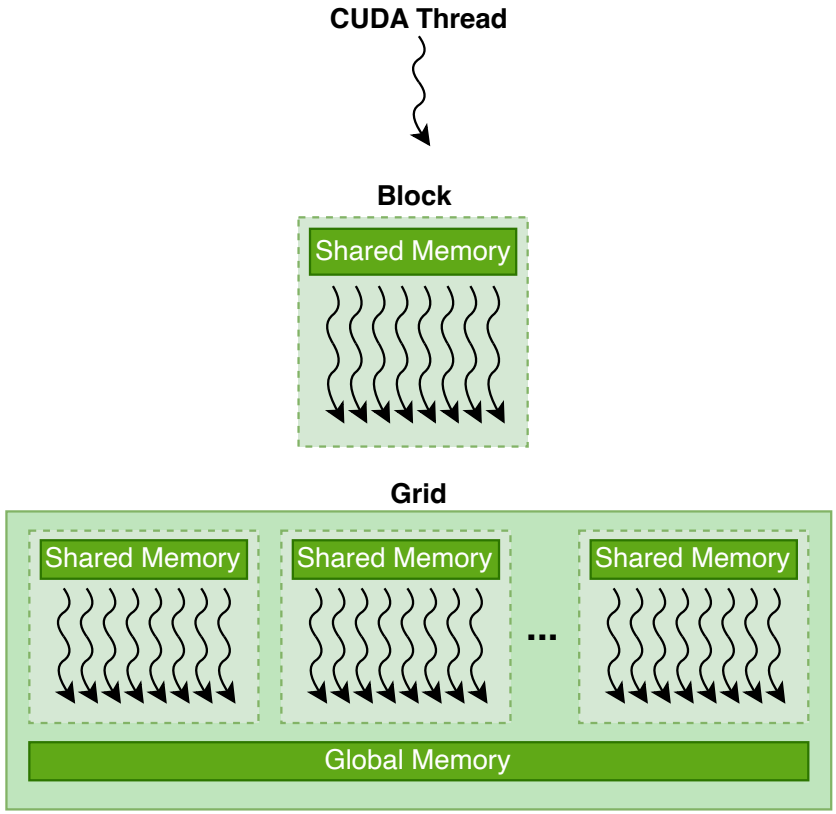
Kernel Code and Thread

CUDA Kernel Code:

- A kernel is a special function that runs on the GPU
- Called from CPU code but executed on GPU

CUDA Thread:

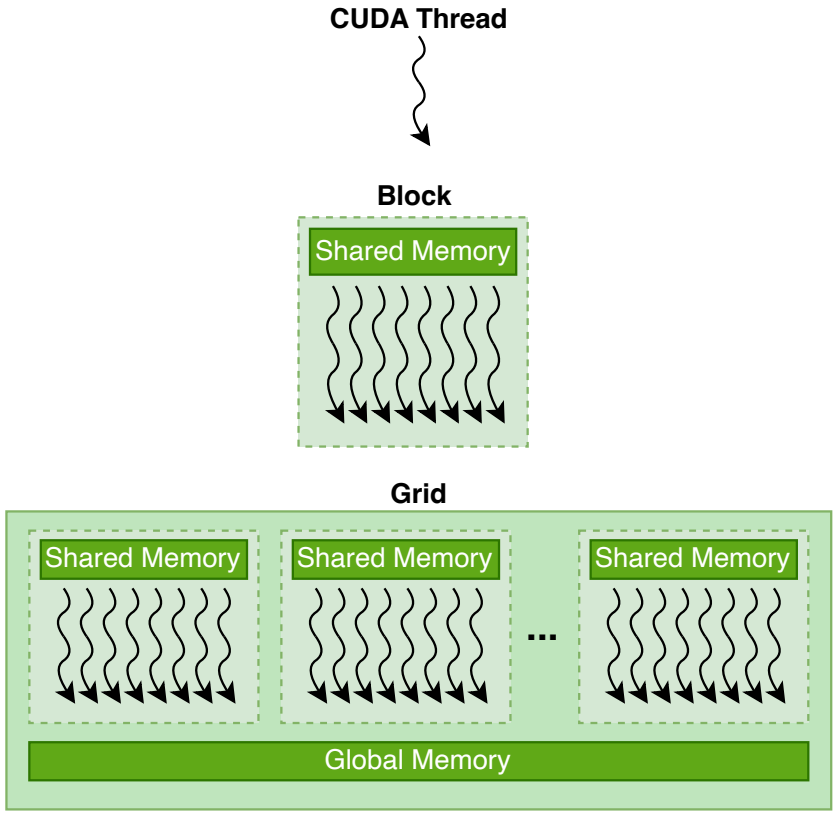
- A *thread of execution* (or "thread" for short) is the lowest unit of programming for GPUs.



Warps: A hidden level in Programming Model

Warps

- A warp is a group of threads that are scheduled together and execute in parallel. All threads in a warp are scheduled onto a single Streaming Multiprocessor (SM) . A single SM typically executes multiple warps, at the very least all warps from the same Cooperative Thread Array , aka thread block .
- Warps are the typical unit of execution on a GPU. In normal execution, all threads of a warp execute the same instruction in parallel — the so-called "Single-Instruction, Multiple Thread" or SIMT model. Warp size is technically a machine-dependent constant, but in practice it is 32.
- Warps are not actually part of the CUDA programming model 's thread group hierarchy.



Block and Grid

Block

A block (or thread block), is a group of warps.

Blocks are the smallest unit of thread coordination exposed to programmers.

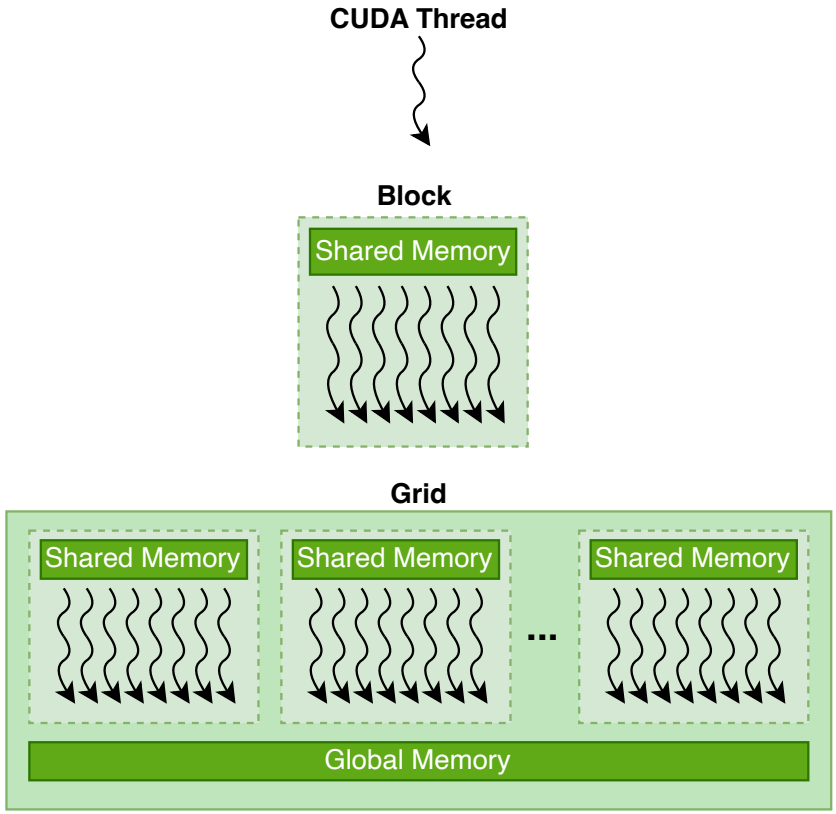
Blocks must execute independently, so that any execution order for blocks is valid, from fully serial in any order to all interleavings.

Grid

When a CUDA kernel is launched, it creates a collection of threads known as a grid (or thread block grid). Grids can be one, two, or three-dimensional.

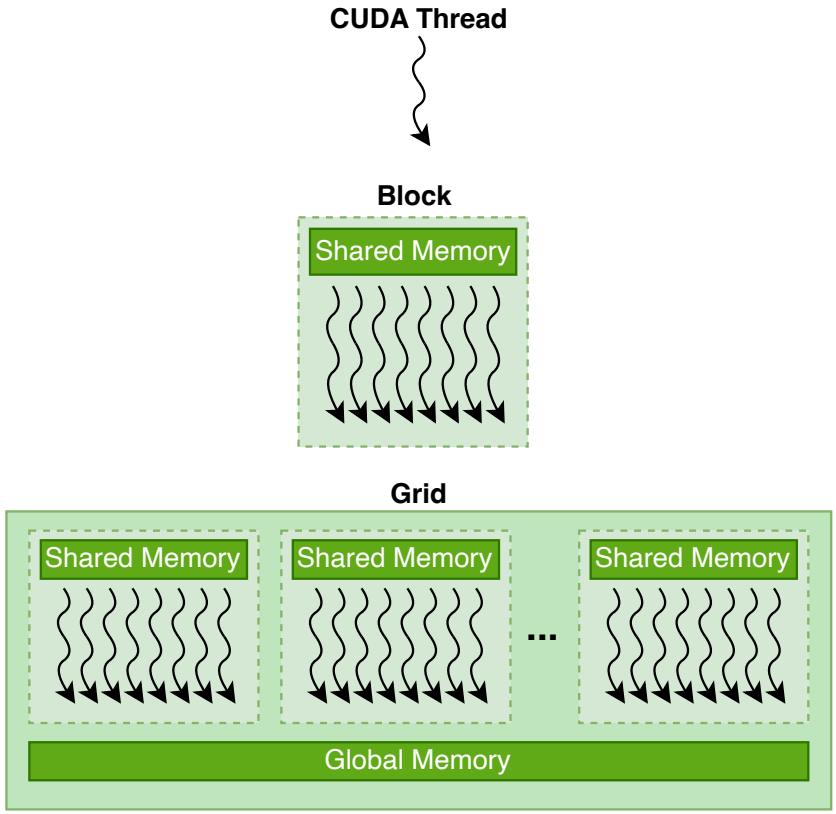
The matching level of the memory hierarchy is the global memory .

Block and Grid are both logical concept!

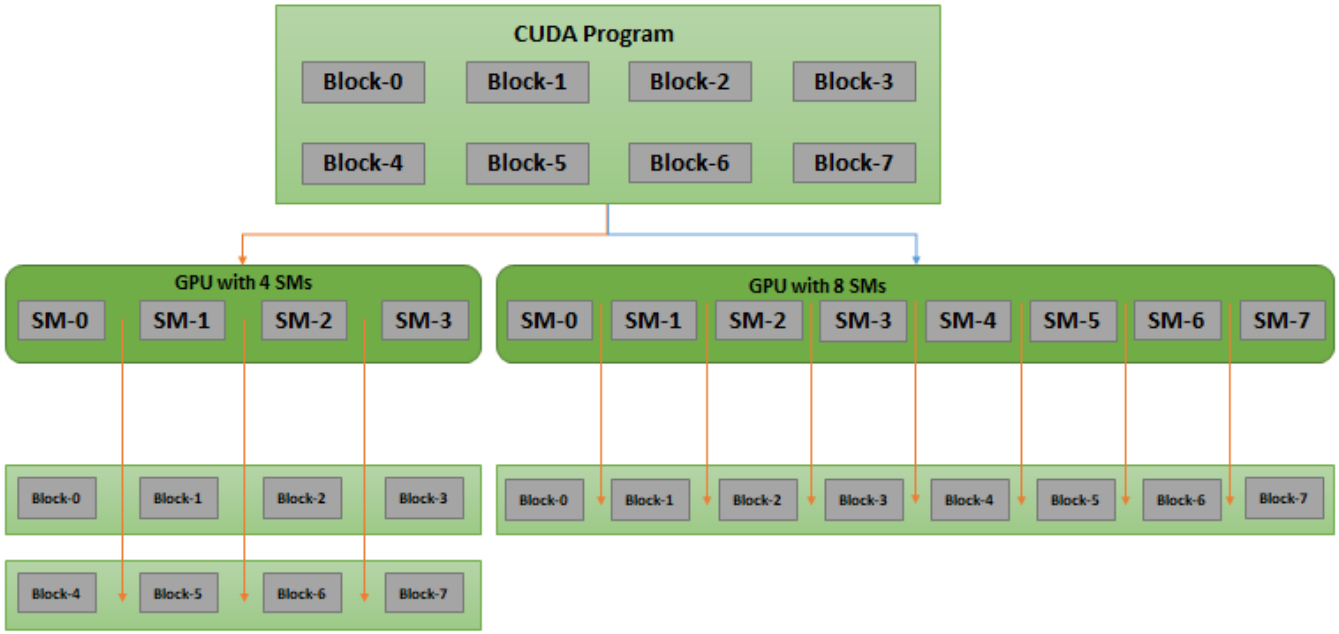


Logic Model on GPU = Block-Thread Parallel

- Block and Thread are 2 different levels of parallel.
- In <<<arg1, arg2>>>
- arg1 = number of blocks in grid
- = gridDim.x
- arg2 = number of threads in block
- = blockDim.x



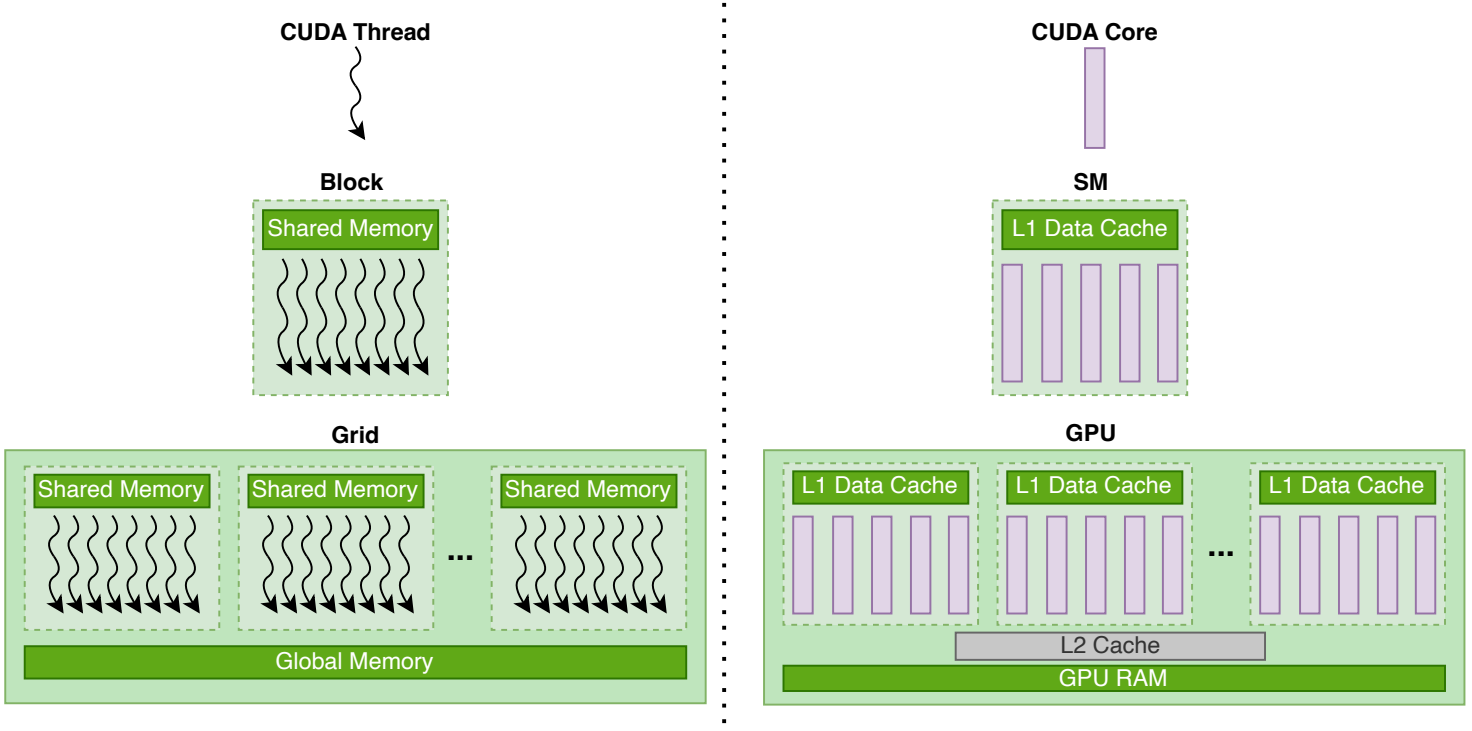
Block and Grid



- The compiled CUDA program has eight CUDA blocks. The CUDA runtime can choose how to allocate these blocks to multiprocessors as shown with streaming multiprocessors (SMs).

- For a smaller GPU with four SMs, each SM gets two CUDA blocks. For a larger GPU with eight SMs, each SM gets one CUDA block. This enables performance scalability for applications with more powerful GPUs without any code changes. **(We don't have to program twice for different GPUs!)**

Overview of GPU Architecture: Logic view (left) and Physical view (right)

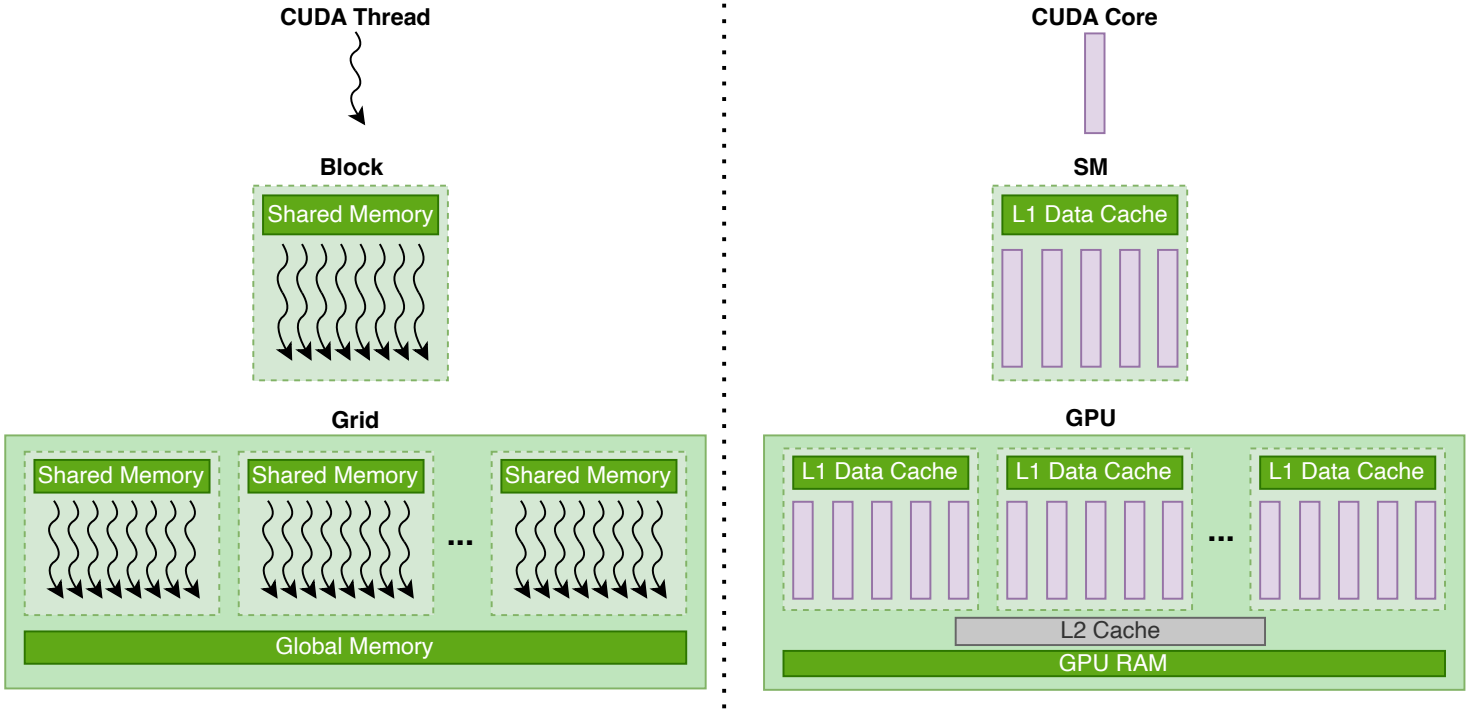


- SP(CUDA Cores) is used to handle threads.
- SM is used to handle blocks.

- Kernel runs in a Grid
- A Grid = some Blocks
- A Block = many Threads
- Kernel runs in parallel across many GPU threads finally

- Shared Memory are stored in L1 Data Cache.
- Global Memory are stored in GPU RAM.

Overview of GPU Architecture: Logic view (left) and Physical view (right)



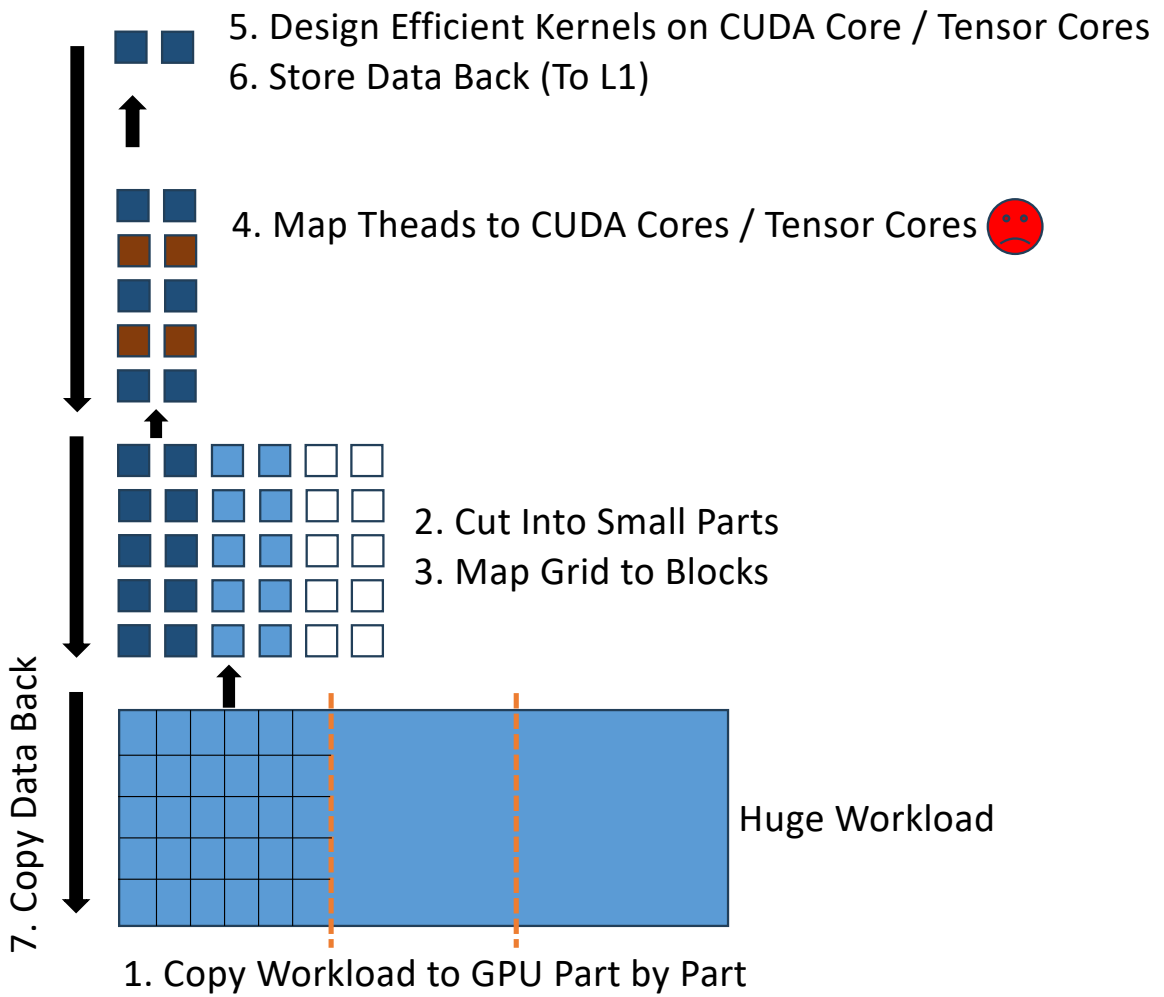
- SP(CUDA Cores) is used to handle threads.
- SM is used to handle blocks.

- **Kernel runs in a Grid**
- **A Grid = some Blocks**
- **A Block = many Threads**
- **Kernel runs in parallel across many GPU threads finally**

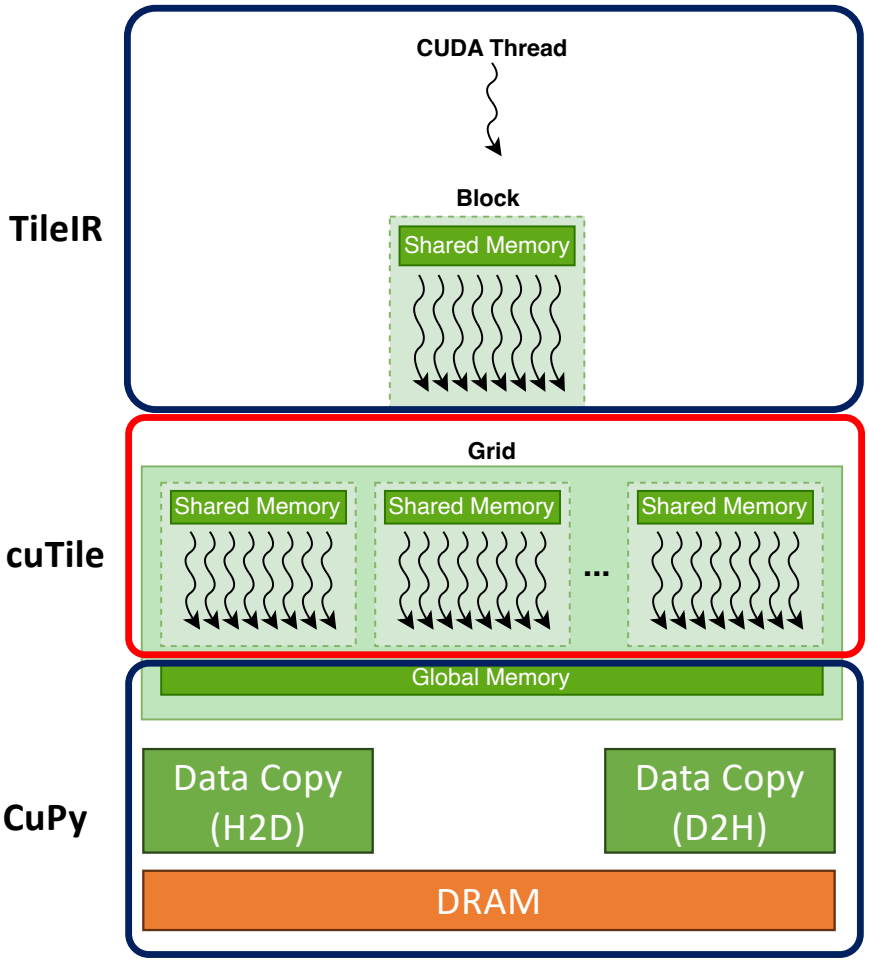
- Shared Memory are stored in L1 Data Cache.
- Global Memory are stored in GPU RAM.

But No More This Year …?

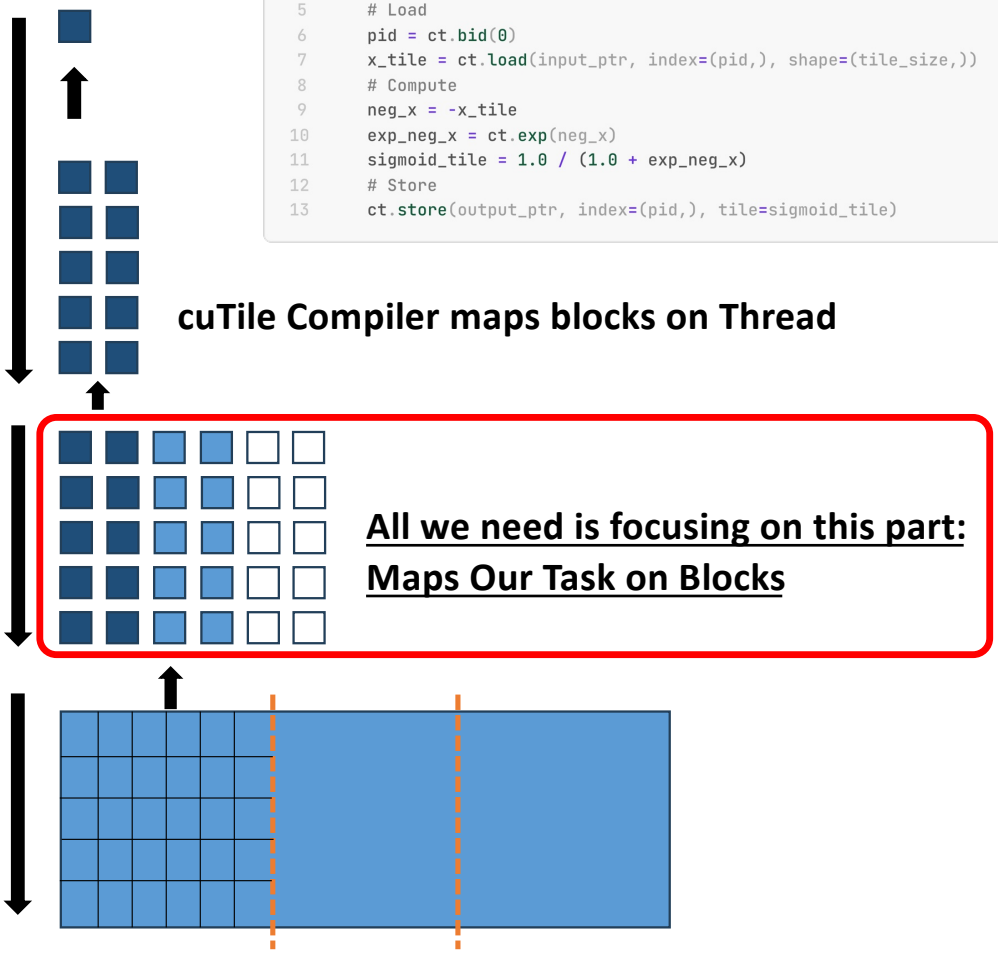
```python
from numba import cuda
import numpy as np
import math

@cuda.jit
def sigmoid_kernel(input_arr, output_arr, tile_size):
    # Load
    pid = cuda.blockIdx.x
    tid = cuda.threadIdx.x

    # Calculate global index
    idx = pid * tile_size + tid

    # Bounds check
    if idx < input_arr.size:
        # Compute
        neg_x = -input_arr[idx]
        exp_neg_x = math.exp(neg_x)
        sigmoid_val = 1.0 / (1.0 + exp_neg_x)

        # Store
        output_arr[idx] = sigmoid_va

n = 1024
tile_size = 256

x = np.random.randn(n).astype(np.float32)
y = np.zeros_like(x)
# Copy to device
d_x = cuda.to_device(x)
d_y = cuda.to_device(y)
# Launch kernel
blocks_per_grid = (n + tile_size - 1) // tile_size
threads_per_block = tile_size
sigmoid_kernel[blocks_per_grid, threads_per_block](d_x, d_y, tile_size)

```



cuTile Makes everything simple now




```python
import cuda.tile as ct

@ct.kernel
def sigmoid_kernel(input_ptr, output_ptr, tile_size: ct.Constant[int]):
    # Load
    pid = ct.bid(0)
    x_tile = ct.load(input_ptr, index=(pid,), shape=(tile_size,))
    # Compute
    neg_x = -x_tile
    exp_neg_x = ct.exp(neg_x)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)
    # Store
    ct.store(output_ptr, index=(pid,), tile=sigmoid_tile)
```



cuTile Compiler maps blocks on Thread

All we need is focusing on this part:
Maps Our Task on Blocks

CuPy helps us manage Memory in a unified method


THE UNIVERSITY of EDINBURGH
informatics
Learn GPU Programming by Examples
CUDA Version
Let's go to Page 85
INFORMATICS FORUM

Table of Content

Besides CUDA, what else can we use to work with GPUs in Python?

- **CuPy**
  - Open-source library for GPU-accelerated computing.
  - Provides a NumPy-like array object for seamless GPU integration.
  - Minimal code changes required for GPU acceleration.
  - Utilizes CUDA Toolkit libraries (cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN, NCCL) for optimal GPU performance.
- **PyTorch**
  - Popular open-source machine learning library.
  - Offers GPU acceleration for deep learning models.
  - Features dynamic computation graphs.
  - Widely used in AI research and production.
- **Triton**
  - Language and compiler for high-performance deep learning kernels.
  - Simplifies writing custom GPU kernels.
  - Focuses on speed and portability.
- **Numba**
  - Just-in-Time (JIT) compiler for Python.
  - Translates Python code into optimized machine code at runtime.
  - Enables GPU acceleration with minimal code changes.
- **TensorFlow**
  - Open-source library for machine learning and deep learning.
  - Comprehensive GPU acceleration support.
  - Efficient for training and deploying models.

In this course, we are going to use most widely used 3 libraries: **CuPy, PyTorch, and Triton**

Example 0: Vector add (CUDA version)

CUDA Vector Add

```
__global__ void vector_add(const float* a, const float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

// c = a + b
vector_add<<<gridSize, blockSize>>>(a, b, c, n);
```

Vector add is the most representative example of mapping an 1-dimensional parallel task on a GPU.

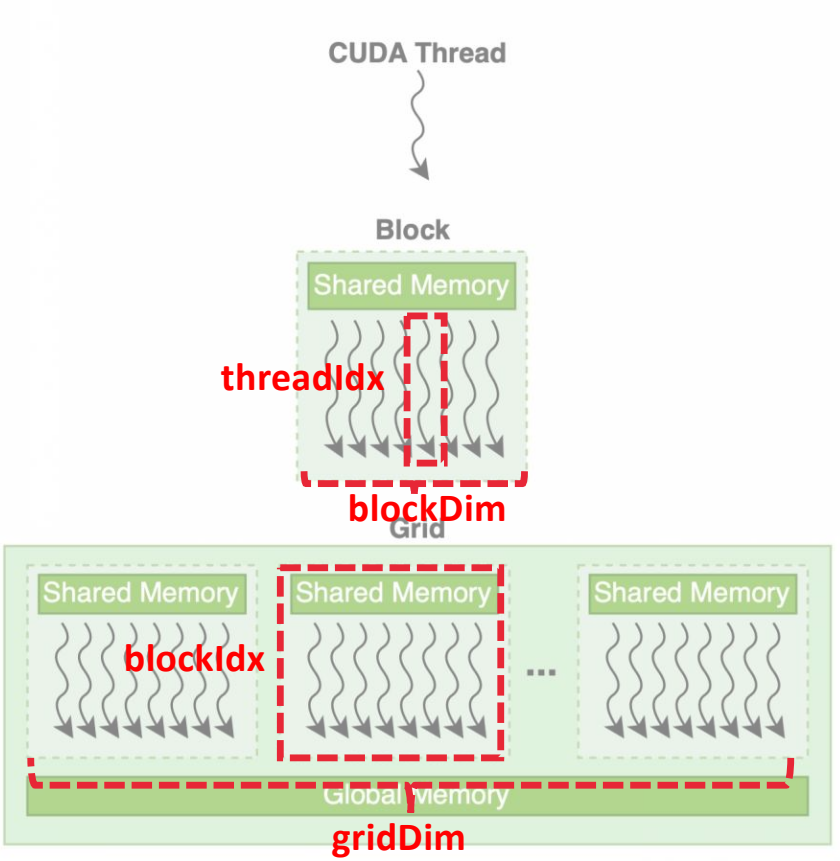
a, b and c are all 1-D array

Which core does a kernel thread get assigned to?

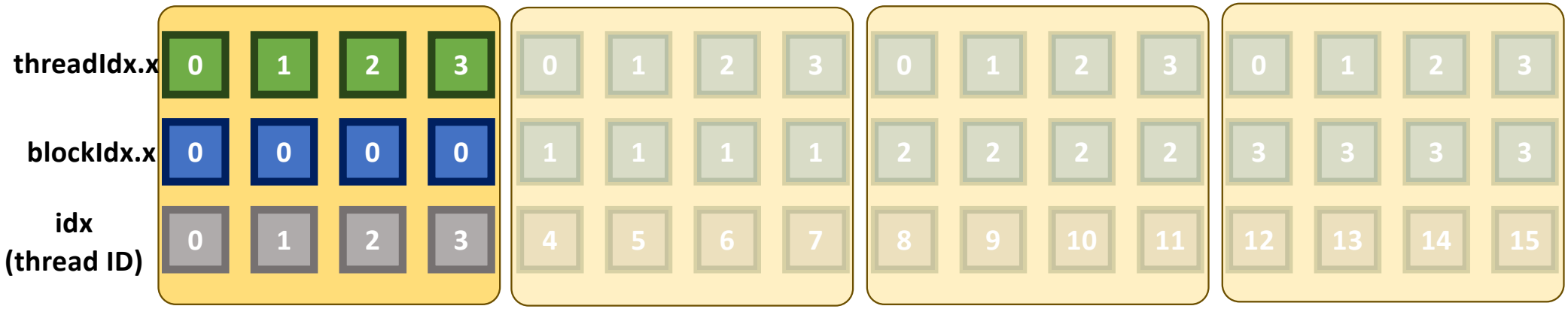
- When code is executed on GPU, we allocate our kernel function to a grid
- For each grid, we have *gridDim* blocks and use *blockIdx* to locate the block inside
- For each block, we have *blockDim* threads and use *threadIdx* to locate the thread inside

- Dim and Idx can be 1, 2, or 3 dimensions, but we typically use 1 dimension for 1D array and 2 dimensions for 2D-array(matrix).

In 1D array:
idx = blockId.x * blockDim.x + threadIdx.x

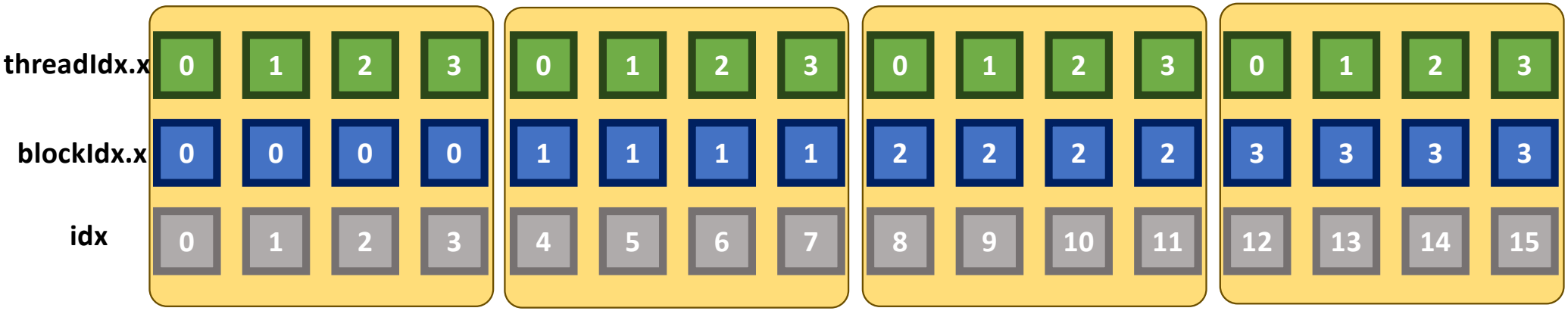


Which thread do I belong to (Example: BlockDim=4; GridDim=4)



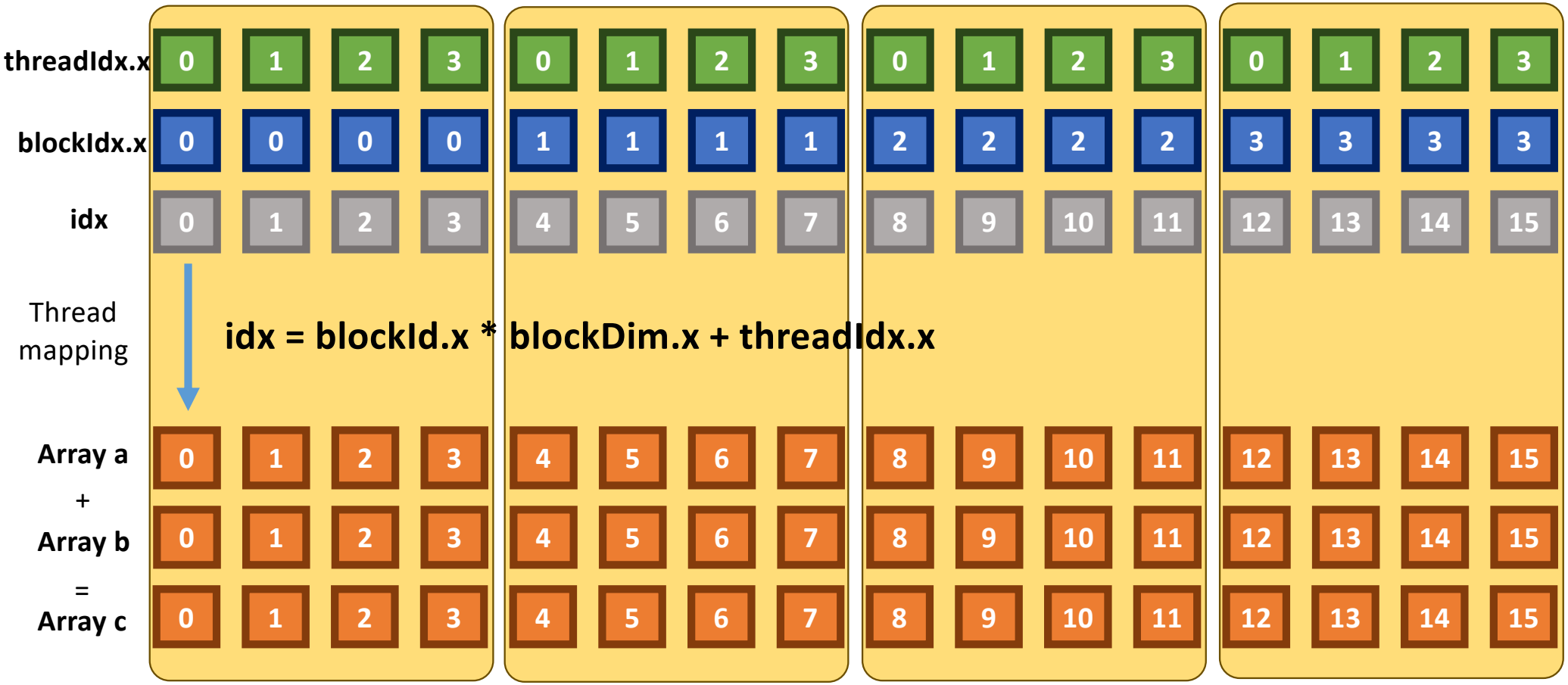
idx = blockId.x * blockDim.x + threadIdx.x

Which thread do I belong to (Example: BlockDim=4; GridDim=4)



idx = blockId.x * blockDim.x + threadIdx.x

Which thread do I belong to (Example: BlockDim=4; GridDim=4)



In programming, we typically set a static thread number in a block (blockDim.x) for convenience. And we allocate a large number of blocks (gridDim.x) for large workload.

We should use proper < gridDim, blockDim> to fully utilize GPU.

- Total thread number = gridDim * blockDim
- If gridDim or blockDim is too low, we cannot fully utilize GPU cores
- If blockDim is set too high, an excessive number of threads will overwhelm an SM's limited cores, potentially starving threads of computing or memory resources.
- Now the question is how to set proper gridDim/blockDim?

```
Device 0: "NVIDIA GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          11.5 / 11.5
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11178 MBytes (11721506816 bytes)
  (028) Multiprocessors, (128) CUDA Cores/MP:    3584 CUDA Cores
  GPU Max Clock rate:                            1582 MHz (1.58 GHz)
  Memory Clock rate:                             5505 Mhz
  Memory Bus Width:                              352-bit
  L2 Cache Size:                                 2883584 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

- In https://github.com/NVIDIA/cuda-samples/blob/master/Samples/1_Utilities/deviceQuery ,you can read device ability with NVIDIA-CUDA-Samples

- [nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf](https://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf)

Set grid/block size for kernel

Why set blockSize 256?

- **Fits GPU Warp Architecture**
  - 256 divides evenly by 32 (#threads in a warp), making 8 warps per block for fast processing.
- **Keeps GPU SM Busy**
  - 256 threads per block helps keep all GPU parts working without waiting.
- **Better L1 Memory Use**
  - Shared memory is used well, helping threads share data quickly.

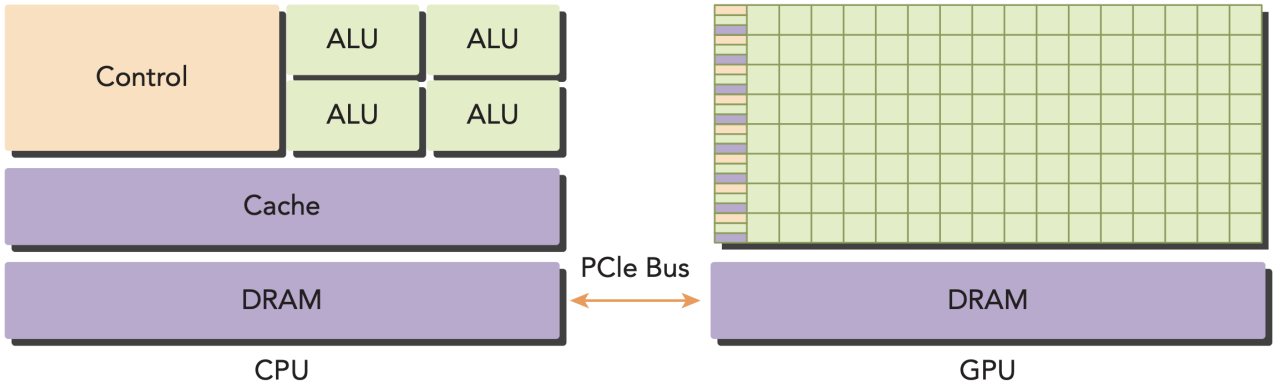
- Equivalent to $\lceil n/blockSize \rceil$

Grid/Block Size for Kernel

```
// Host memory
float *a, *b, *c;
a = (float*)malloc(size);
b = (float*)malloc(size);
c = (float*)malloc(size);

// Initialize host data
for(int i = 0; i < n; i++){
    a[i] = 1.0f;
    b[i] = 2.0f;
}

// Launch kernel
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;

vector_add<<<gridSize, blockSize>>>(a, b, c, n);

// print first 10 elements of c
for(int i = 0; i < 10; i++){
    printf("%f ", c[i]);
}
printf("\n");

// Clean up
free(a); free(b); free(c);
```

Error? Host memory and device memory

```
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.
000000 0.000000 0.000000
```



	Host Memory		Device Memory
Create Space	malloc	cudaMemcpy	cudaMalloc
Delete Space	free		cudaFree

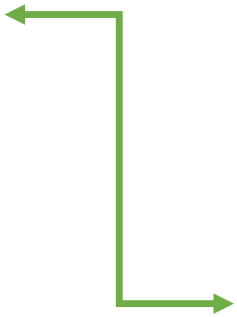
- Device can only visit device memory by address
- Host can only visit host memory by address

- We need CUDA memory management API to copy data between host and device

Final cuda version for vector add

FInal Cuda code for vector add

```
__global__ void vector_add(const float* a, const float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```



FInal Cuda code for vector add

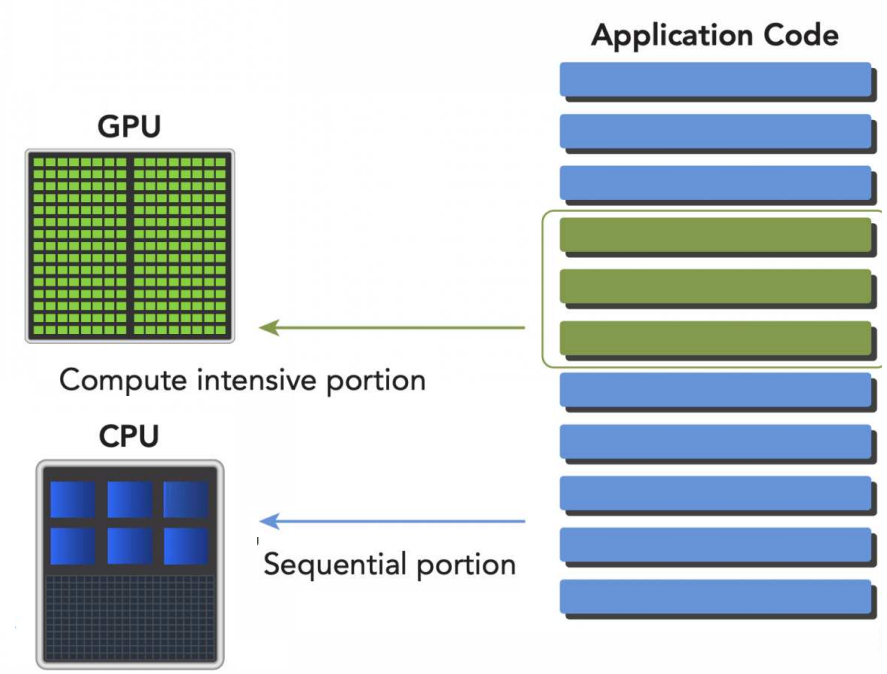
```
// Host memory
float *h_a, *h_b, *h_c;
h_a = (float*)malloc(size);
h_b = (float*)malloc(size);
h_c = (float*)malloc(size);

// Initialize host data
for(int i = 0; i < n; i++){
    h_a[i] = 1.0f; h_b[i] = 2.0f;
}

// Device memory
float *d_a, *d_b, *d_c;
cudaMalloc(&d_a, size);
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);

// Copy host to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch kernel
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;

vector_add<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy Result device to host
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
// Sync
cudaDeviceSynchronize();

// print first 10 elements of h_c
for(int i = 0; i < 10; i++){
    printf("%f ", h_c[i]);
}
printf("\n");

// Clean up
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
free(h_a); free(h_b); free(h_c);
```

Programming pattern on CUDA:

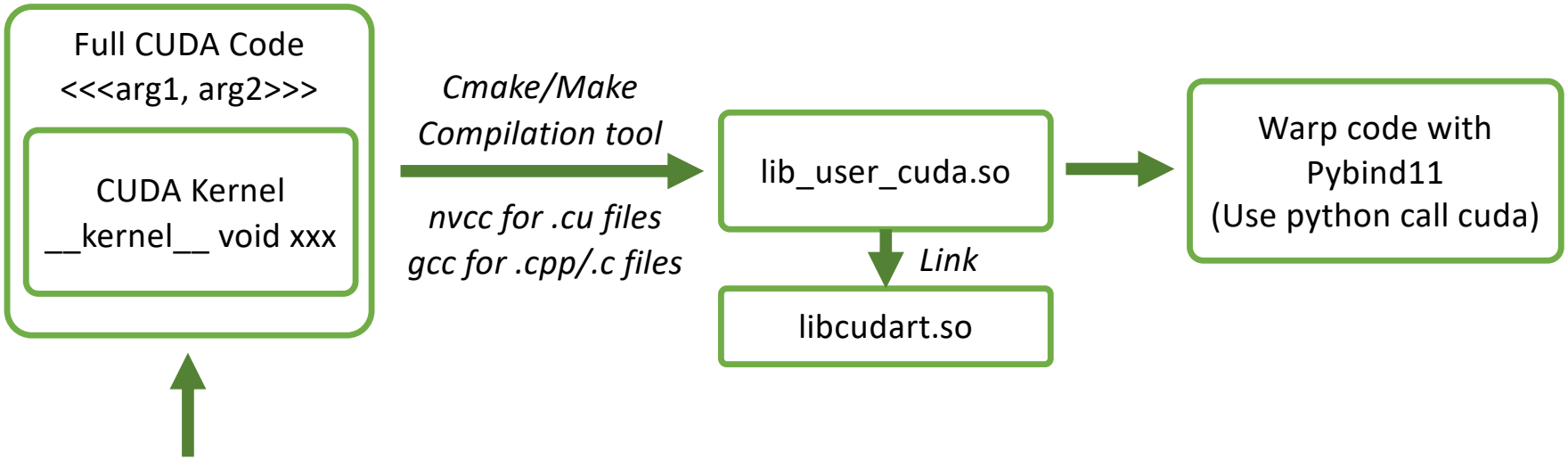
- Identify the sections that can be parallelized
- Copy the relevant data to the GPU
- Run the CUDA kernel
- Copy the results back to the CPU

Why `cudaDeviceSynchronize`

- kernel is called by CPU but run on GPU
- CUDA kernel launches are asynchronous
  - The CPU code continues executing without waiting for the GPU to complete
  - This means your CPU code might proceed before the GPU has finished its work
- cudaDeviceSynchronize is used to synchronized GPU and CPU



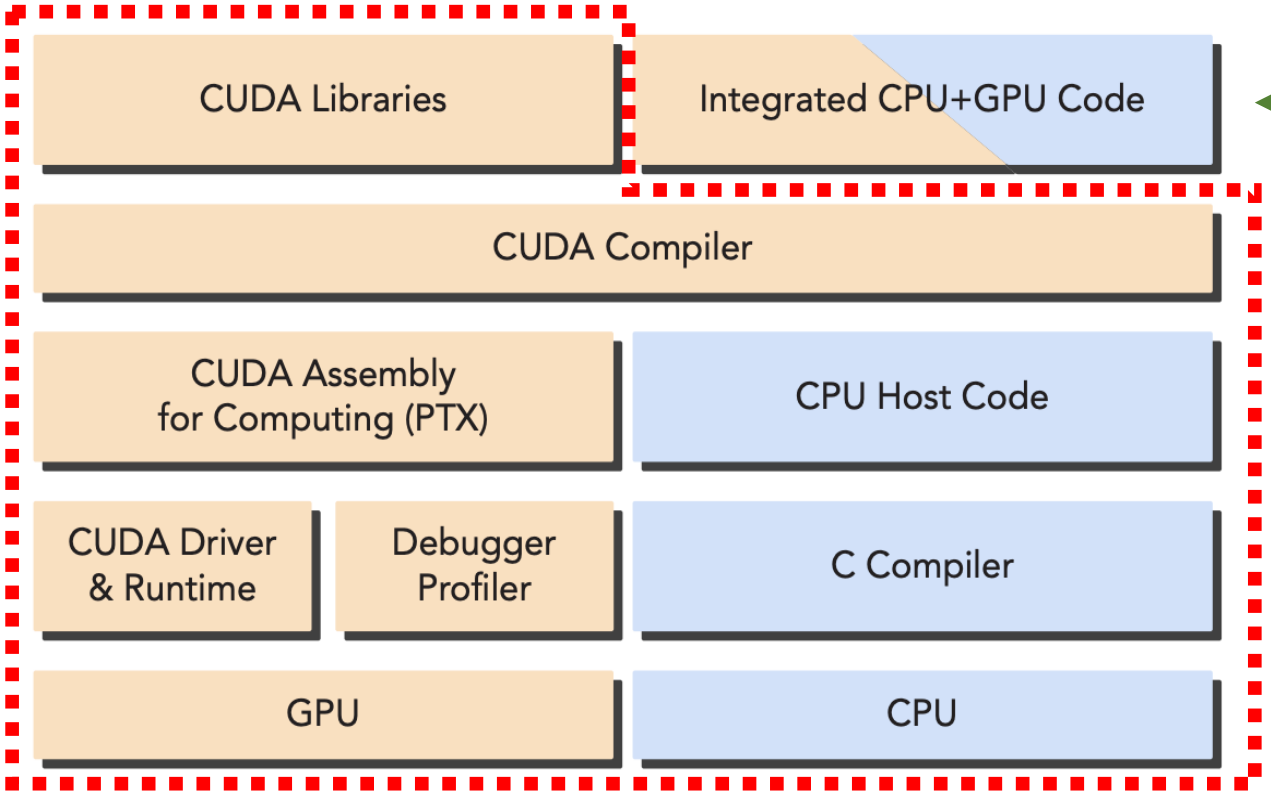
A simple workflow for compiling cuda and bind to python API



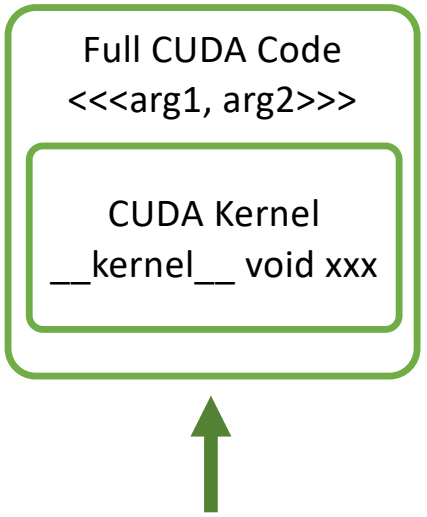
What we have learned just now.

1. We need a compilation tool like CMake or GNU Make to clarify file relationships.
2. .cu files should be compiled using nvcc, while .c/.cpp files should be compiled with GNU compilers like gcc/g++.
3. We need some linking tools to locate the correct CUDA libraries.
4. After that, we will have our library.
5. We need Pybind to assist us in building the final API to call the library and allocate resources on the GPU.

Though we know CUDA programming, we still have a long way to go.



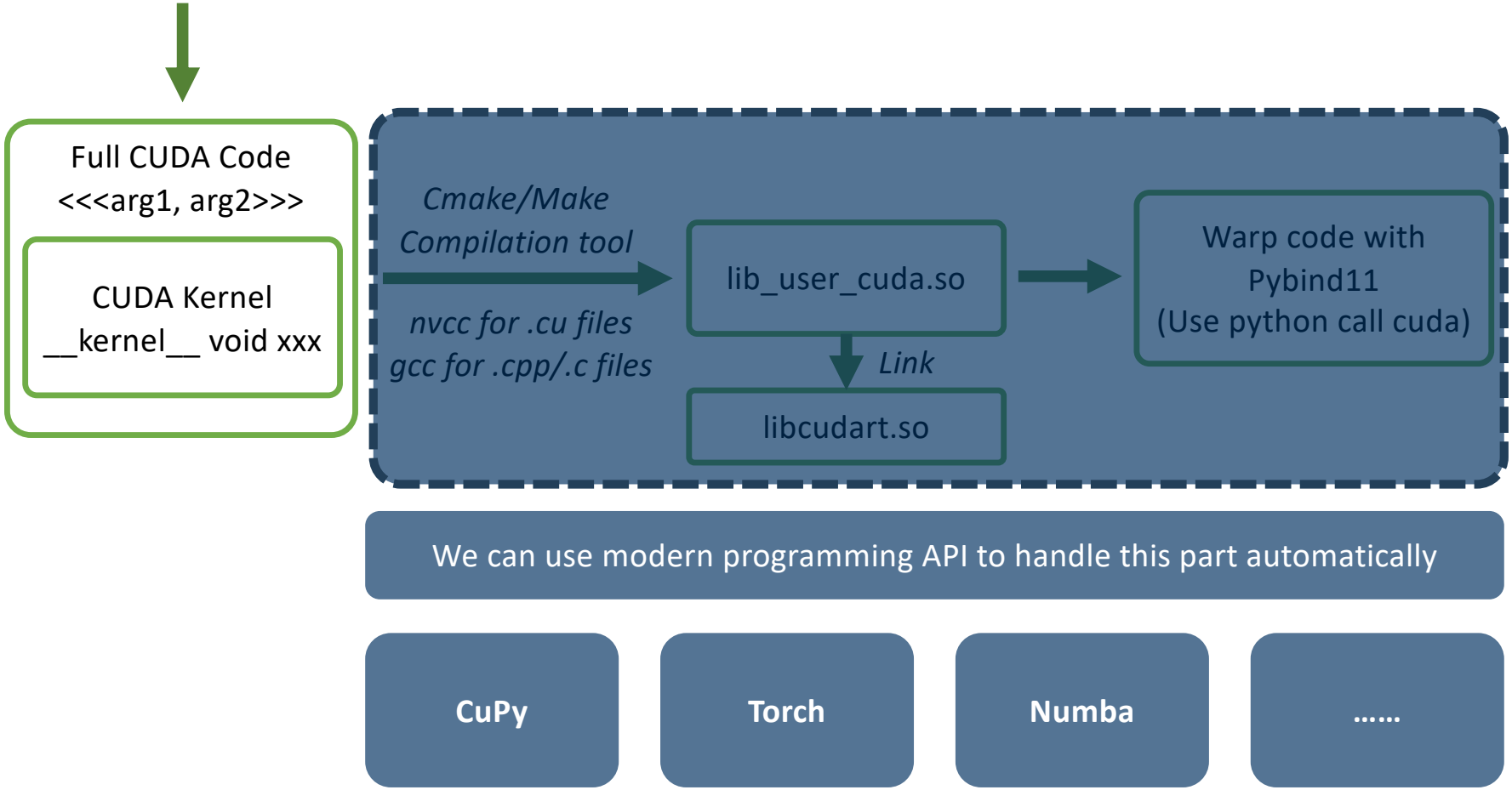
What we still don't know



What we have learned just now.

A simplified workflow with modern APIs

What we have learned just now.



From CUDA to modern programming API

FInal Cuda code for vector add

```cpp
// Host memory
float *h_a, *h_b, *h_c;
h_a = (float*)malloc(size);
h_b = (float*)malloc(size);
h_c = (float*)malloc(size);

// Initialize host data
for(int i = 0; i < n; i++){
    h_a[i] = 1.0f; h_b[i] = 2.0f;
}

// Device memory
float *d_a, *d_b, *d_c;
cudaMalloc(&d_a, size);
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);

// Copy host to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch kernel
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;

vector_add<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy Result device to host
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
// Sync
cudaDeviceSynchronize();

// print first 10 elements of h_c
for(int i = 0; i < 10; i++){
    printf("%f ", h_c[i]);
}
printf("\n");

// Clean up
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
free(h_a); free(h_b); free(h_c);
```

CUDA is complex

CuPy is easier

Cupy Vector Add

```python
import cupy as cp

def cupy_vector_add(a, b):
    return a + b  # Perform vector addition

# Create vectors on GPU
a = cp.random.rand(n, dtype=cp.float32)
b = cp.random.rand(n, dtype=cp.float32)

c = cupy_vector_add(a, b)
cp.cuda.Stream.null.synchronize()
```

Limitations of CuPy:

- Performance Overhead: Slower for small matrices due to kernel launch overhead.
- Structural Limitations: Less flexible for complex data structures compared to NumPy.
- Python only: CuPy is a Python only programming API

Example 0: Vector add (Torch & CuPy version)

Torch Vector Add

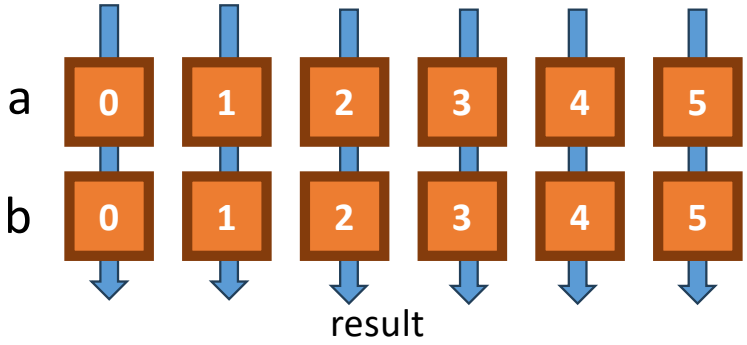
```
import torch

x = torch.rand(n, device="cuda", dtype=torch.float32)
y = torch.rand(n, device="cuda", dtype=torch.float32)
z = torch.empty_like(x)

z = x + y
```

Cupy Vector Add

```
import cupy as cp

def cupy_vector_add(a, b):
    return a + b # Perform vector addition

# Create vectors on GPU
a = cp.random.rand(n, dtype=cp.float32)
b = cp.random.rand(n, dtype=cp.float32)

c = cupy_vector_add(a, b)
cp.cuda.Stream.null.synchronize()
```

1. We cannot find cudaMemcpy in Python code
2. We just use '+' instead of complex kernel code

Common Operators on GPU



Elementwise: Every item executes the same pattern/operator

Example:
- result[i] = a[i] + b[i]
- result[i] = a[i] * k + a[i] * a[i]

Reduction: a reduction operation on a large dataset, typically to compute a single output value from multiple input values.

Example:
- Result = max(a[i])
- Result = sum(a[i])

CuPy can automatically set the block/thread in the following cases.

- Elementwise kernels
- Reduction kernels
- Widely-used functions like GEMM or GEMV

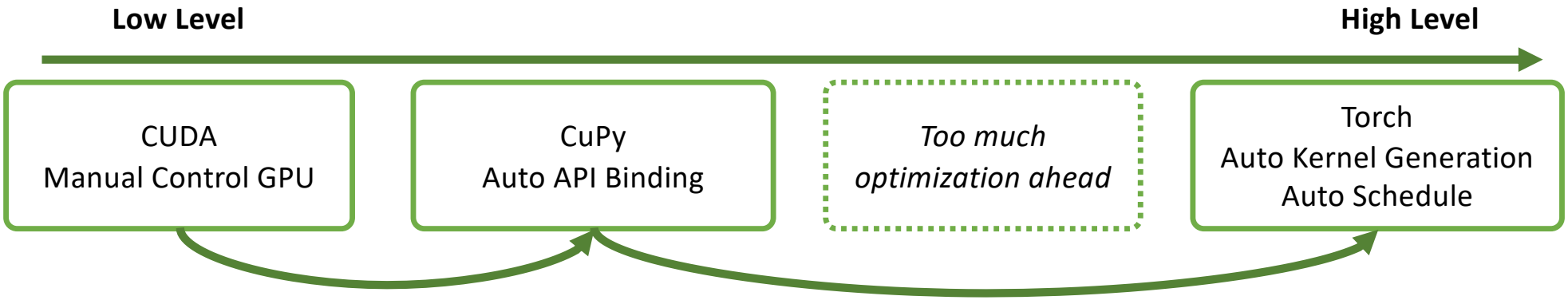
- When use other types of kernels, they still need to set proper grid size and block size

- An raw kernel example is given on the right.

Cupy Raw Kernel

```
add_kernel = cp.RawKernel(r'''
extern "C" __global__
void my_add(const float* x1, const float* x2, float* y) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
}
''', 'my_add')
x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments
print(y)

# Result:
# array([[ 0.,  2.,  4.,  6.,  8.],
#        [10., 12., 14., 16., 18.],
#        [20., 22., 24., 26., 28.],
#        [30., 32., 34., 36., 38.],
#        [40., 42., 44., 46., 48.]], dtype=float32)
```

Difference between APIs



- Elementwise/Reduce Kernel auto-generation
- CUDA Block/Thread auto-tuning
- Compilation options for optimization
- Python API warp

- AI Compilers + Torch Execution Engine
- Memory Management
- Auto block/thread tuning
- Kernel code generation
- Many other optimizations

Example 1: CUDA Stream

- What is CUDA Stream?
  - A CUDA *stream* is a queue of GPU operations that are executed in a specific order.

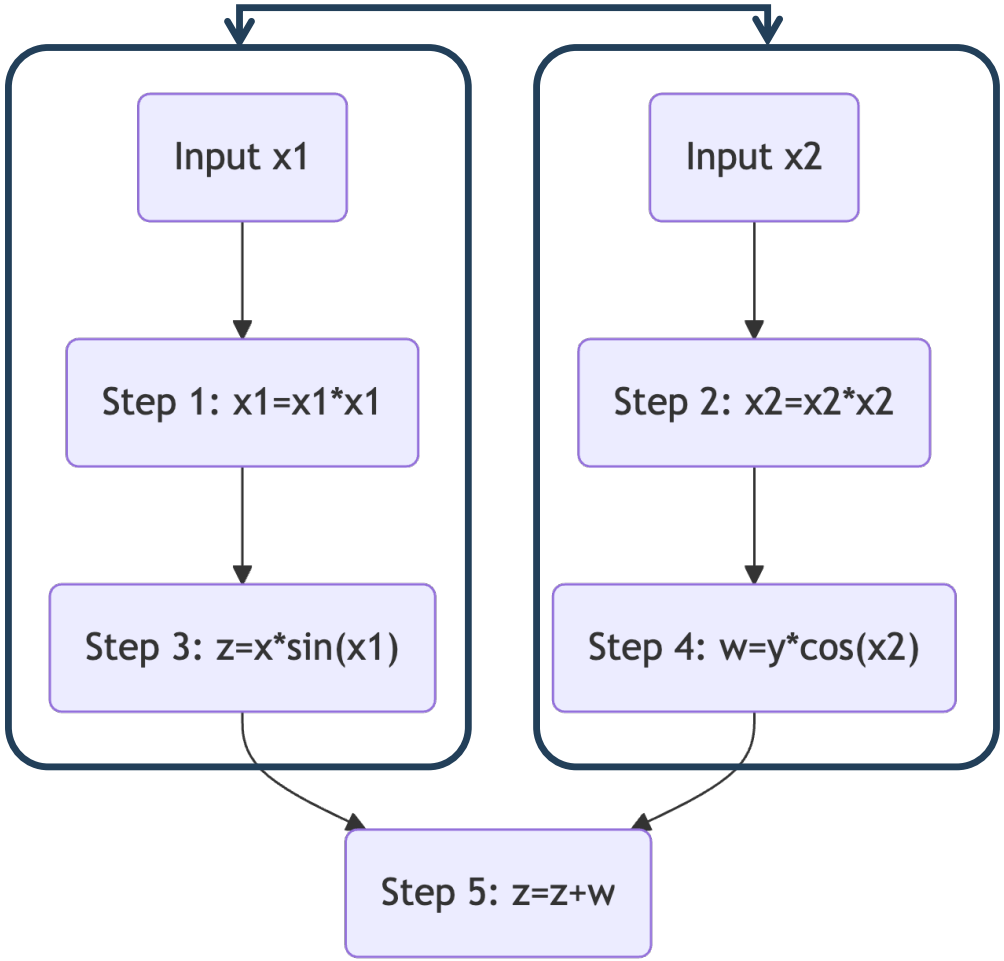
- When we need Stream
  - Concurrent Kernel Execution
  - Pipeline operators
    (usually refer to overlap data transfer and computation)

[Slide 1](https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf)

A Stream Async Example

```
# Step 1: x = x1 * x1
# Step 2: y = x2 * x2
# Step 3: z = x * sin(x1)
# Step 4: w = y * cos(x2)
# Step 5: z = z + w
```

Step 1 and Step 3 are dependent on each other.
Step 2 and Step 4 also have a dependency.
However, [1, 3] and [2, 4] can be executed in parallel.



CuPy Version Stream

- CuPy:
- With streams: 84.057091 ms.
- Without streams: 223.951874 ms.

Cupy Stream

```
def compute_with_streams(x1, x2):
    # Create CUDA streams
    stream1 = cp.cuda.Stream()
    stream2 = cp.cuda.Stream()

    # Step 1 & 2: x = x1 * x1 and y = x2 * x2 (run in separate streams)
    # No need to immediately synchronize
    with stream1:
        x = square_kernel(x1)
        # Step 3: z = x * sin(x1) (stream3, depends on `x`)
        z = x * sin_kernel(x1)
    with stream2:
        y = square_kernel(x2)
        # Step 4: w = y * cos(x2) (stream4, depends on `y`)
        w = y * cos_kernel(x2)

    # Step 5: z = z + w (sync all before this step)
    cp.cuda.Stream(null=True).synchronize()  # Synchronize all streams
    z = z + w

    return z
```

Create Streams

Stream 1

Stream 2

Torch Version Stream

- Torch:
- With streams: 97.108994 ms.
- Without streams: 233.461761 ms.

Torch Stream

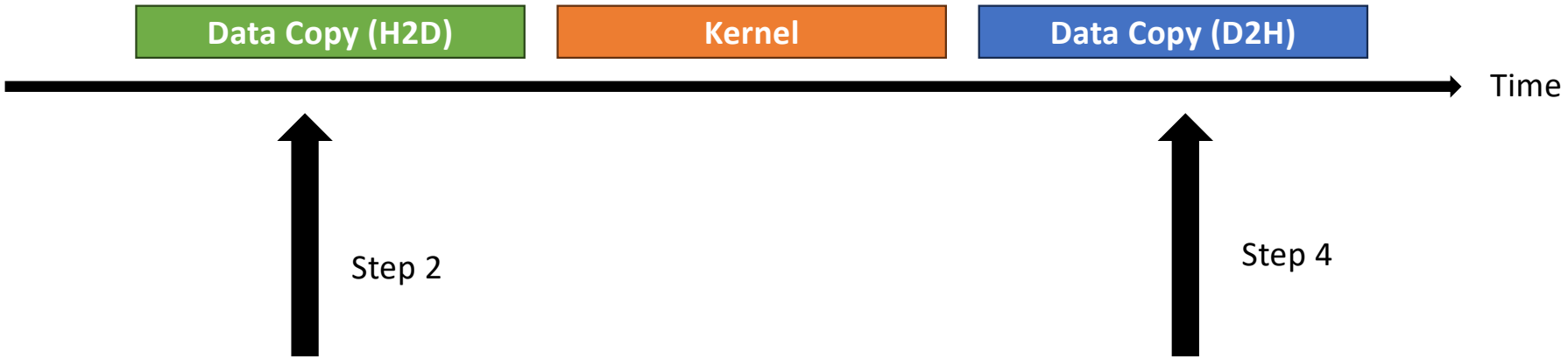
Create Streams

Stream 1

Stream 2

```
def compute_with_streams(x1, x2):
    # Create CUDA streams
    stream1 = torch.cuda.Stream()
    stream2 = torch.cuda.Stream()

    # Allocate tensors for the intermediate results
    z = torch.cuda.FloatTensor()
    w = torch.cuda.FloatTensor()

    # Stream1 operations
    with torch.cuda.stream(stream1):
        x = square_kernel(x1)  # Step 1
        z = x * sin_kernel(x1)  # Step 3: z = x * sin(x1)

    # Stream2 operations
    with torch.cuda.stream(stream2):
        y = square_kernel(x2)  # Step 2
        w = y * cos_kernel(x2)  # Step 4: w = y * cos(x2)

    # Wait for all streams to finish before proceeding
    torch.cuda.synchronize()

    # Step 5: z = z + w (requires results from both streams)
    z = z + w

    return z
```

Another Scenario when we need CUDA Stream

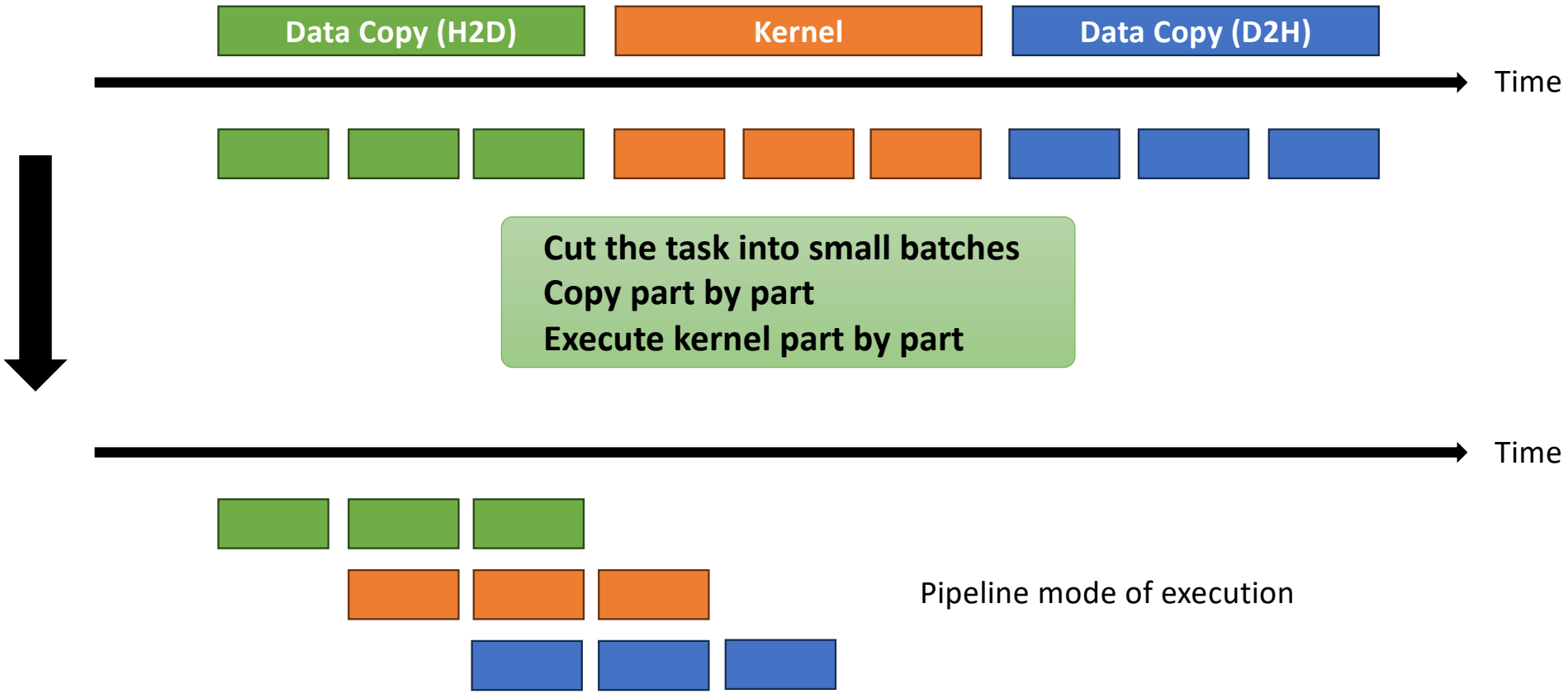


Programming pattern on CUDA:

1. Identify the sections that can be parallelized
2. **Copy the relevant data to the GPU**
3. Run the CUDA kernel
4. **Copy the results back to the CPU**

The GPU remains idle while awaiting data transfer. When the kernel is executed multiple times, significant time is lost during the data readiness period.

Another Scenario when we need CUDA Stream



CuPy Stream Example

- Example:
- Z = (x * y) * (y + sin(x))

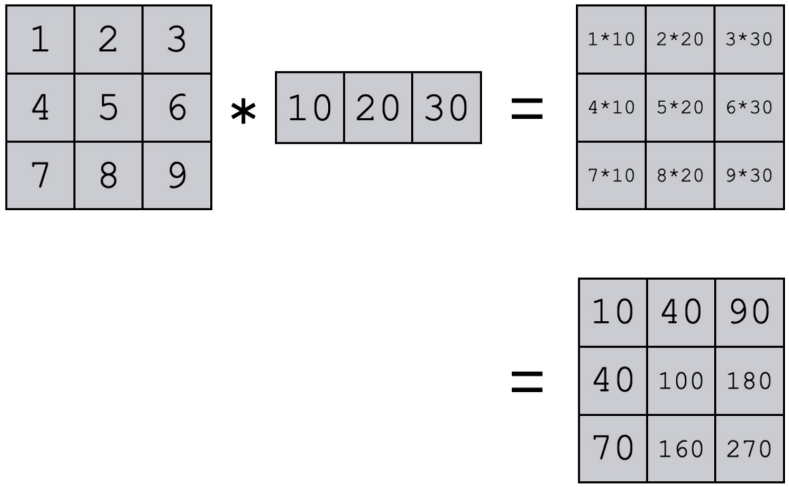
- Stream 1: copy x and y from host to device
- Stream 2: compute z
- Stream 3: copy z from device to host

- With streams example took **1973.514282** s.
- Without streams example took **2574.597900** s.

Stream Data Copy

```
stream1 = cp.cuda.Stream()
stream2 = cp.cuda.Stream()
stream3 = cp.cuda.Stream()

batch_num = 8

# Split arrays into batches
batch_size = len(x) // batch_num
batches = [(i * batch_size, min((i + 1) * batch_size, len(x))) \
           for i in range(batch_num)]

# Initialize output array on device instead of host
z_host = np.empty_like(x)

for start, end in batches:
    with stream1:
        x_d = cp.asarray(x[start:end])
        y_d = cp.asarray(y[start:end])

    with stream2:
        # Need to wait for stream1 to complete
        stream2.wait_event(stream1.record())
        z = x_d * y_d
        w = y_d + sin_kernel(x_d)
        z = z * w

    with stream3:
        # Need to wait for stream2 to complete
        stream3.wait_event(stream2.record())
        z_host[start:end] = cp.asnumpy(z)
```

Example 2: Hierarchy Memory (CuPy)




```python
import numpy as np

N = 2560000
M = 512

A = np.random.rand(N).astype(np.float32)  # Input vector A
B = np.random.rand(M, N).astype(np.float32)  # Input matrix B

C_cpu = np.multiply(A, B) # Hadamard product on CPU
```

Test on CPU with numpy

CPU: AMD EPYC 7453
Time: 751.023459 ms

Example 2: Hierarchy Memory (CuPy)

Test on GPU with CuPy

GPU: RTX A5000
Time: 25.477409 ms

```
import cupy as cp

A_cu = cp.asarray(A)  # Transfer A to GPU
B_cu = cp.asarray(B)  # Transfer B to GPU
C_cu = cp.empty_like(B)  # Allocate memory for the output C

cp.multiply(A_cu, B_cu, out=C_cu)
device.synchronize()  # Ensure all GPU computations finish
```

751ms -> 25ms, improved by moving from CPU to GPU

Example 2: Hierarchy Memory (CuPy)

Test on GPU with CuPy

GPU: RTX A5000
Time: 20.907259 ms

```
from cupyx import jit

@jit.rawkernel()
def my_multiply(A_cu, B_cu, C_cu):

    global_idx = jit.threadIdx.x + jit.blockIdx.x * jit.blockDim.x
    global_idy = jit.threadIdx.y + jit.blockIdx.y * jit.blockDim.y

    C_cu[global_idy, global_idx] = A_cu[global_idx] * B_cu[global_idy, global_idx]

Db = (256, 2, 1)
Dg = (N//Db[0], M//Db[1], 1)
my_multiply[Dg,Db](A_cu, B_cu, C_cu)
device.synchronize()  # Ensure all GPU computations finish
```

25ms -> 21ms, improved by optimising block and grid structure manually

CuPy can set block/grid size automatically, but the strategy is various on different GPU type.

Example 2: Hierarchy Memory (CuPy)

Test on GPU with CuPy

GPU: RTX A5000
Time: 17.024183 ms

21ms -> 17ms, improved by shared memory

multiply_sharedMem_1: Optimised by compiler

multiply_sharedMem_2: Manually load to shared memory

```
from cupyx import jit

@jit.rawkernel()
def multiply_sharedMem_1(A_cu, B_cu, C_cu, bulk_y):

    global_idx = jit.threadIdx.x + jit.blockIdx.x * jit.blockDim.x
    global_idy = jit.threadIdx.y + jit.blockIdx.y * jit.blockDim.y

    bulk_y_u32 = cp.uint32(bulk_y)
    offset_y = global_idy * bulk_y_u32

    for i in range(bulk_y_u32):
        C_cu[offset_y+i, global_idx] = A_cu[global_idx] * B_cu[offset_y+i, global_idx]

@jit.rawkernel()
def multiply_sharedMem_2(A_cu, B_cu, C_cu, bulk_x, bulk_y):

    global_idx = jit.threadIdx.x + jit.blockIdx.x * jit.blockDim.x
    global_idy = jit.threadIdx.y + jit.blockIdx.y * jit.blockDim.y

    shared_mem = jit.shared_memory(cp.float32, 512)

    bulk_x_u32 = cp.uint32(bulk_x)
    bulk_y_u32 = cp.uint32(bulk_y)

    offset_x = global_idx * bulk_x_u32
    offset_y = global_idy * bulk_y_u32

    block_offset = jit.threadIdx.x * bulk_x_u32

    for i in range(bulk_x_u32):
        shared_mem[block_offset+i] = A_cu[offset_x+i]
        jit.syncthreads()

    for i in range(bulk_y_u32):
        for j in range(bulk_x_u32):
            C_cu[offset_y+i, offset_x+j] = shared_mem[block_offset+j] * B_cu[offset_y+i, \\
offset_x+j]

bulk_x = 1
bulk_y = 128
Db = (256, 2, 1)
Dg = (N//Db[0], M//Db[1], 1)
multiply_sharedMem_1[Dg,Db](A_cu, B_cu, C_cu, bulk_y)
device.synchronize()  # Ensure all GPU computations finish
```

Why we need to optimize shared memory?

- Dummy strategy on CuPy

We assume that shared memory size = 4 vectors' size

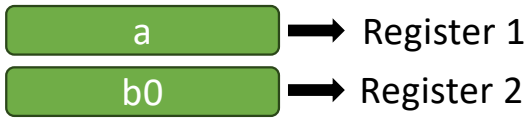
Step 1: load **a** and a line of **b**[0], **a** miss, **b**[0] miss

simplified np.multiply

```
for i in range(n):
    # a is a vector, b and c are matrices
    c[i] = a * b[i]
```

Step1

Shared memory



Cost = 2 * load time

Why we need to optimize shared memory?

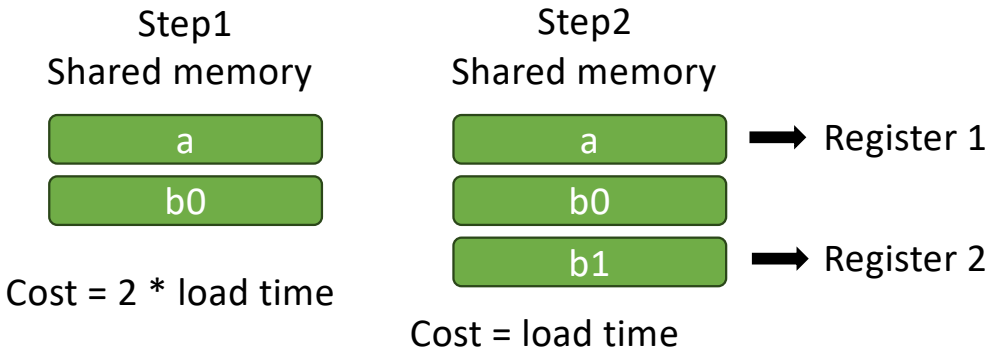
- Dummy strategy on CuPy

We assume that shared memory size = 3 vectors' size

Step 1: load **a** and a line of **b**[0], **a** miss, **b**[0] miss
Step 2: load **a** and a line of **b**[1], **a** hit, **b**[1] miss

simplified np.multiply

```
for i in range(n):
    # a is a vector, b and c are matrices
    c[i] = a * b[i]
```



Why we need to optimize shared memory?

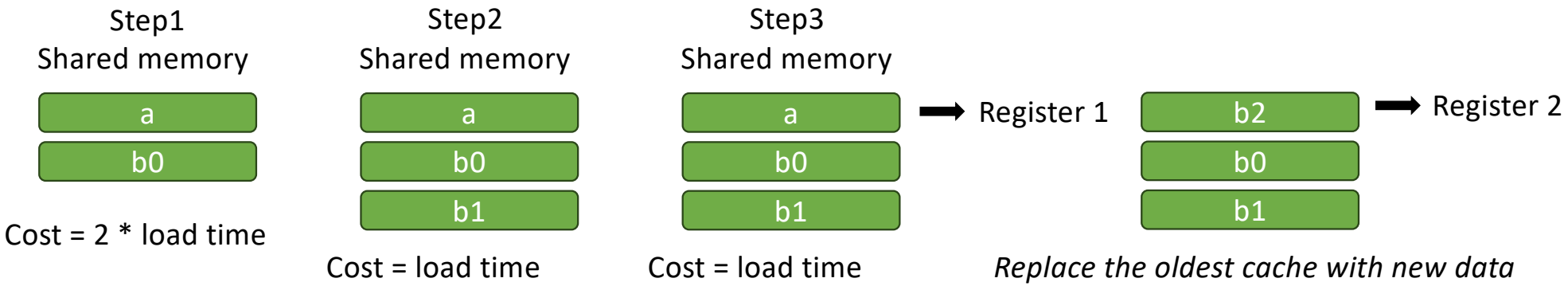
- Dummy strategy on CuPy

We assume that shared memory size = 3 vectors' size

Step 1: load **a** and a line of **b**[0], **a** miss, **b**[0] miss
Step 2: load **a** and a line of **b**[1], **a** hit, **b**[1] miss
Step 3: load **a** and a line of **b**[2], **a** hit, **b**[2] miss.

simplified np.multiply

```
for i in range(n):
    # a is a vector, b and c are matrices
    c[i] = a * b[i]
```



Why we need to optimize shared memory?

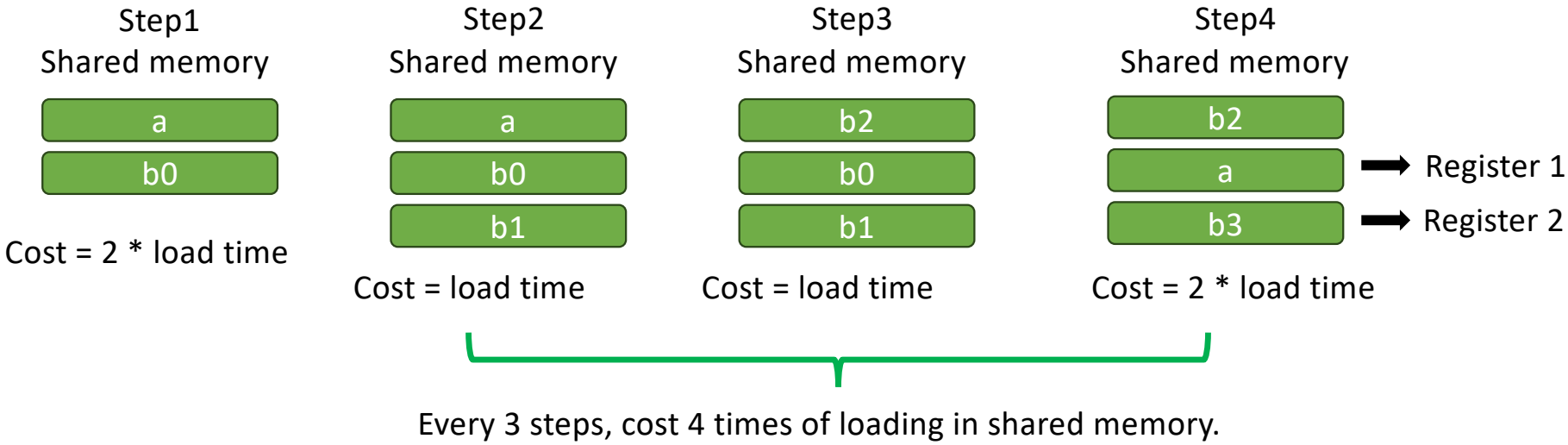
- Dummy strategy on CuPy

We assume that shared memory size = 3 vectors' size

Step 1: load **a** and a line of **b**[0], **a** miss, **b**[0] miss
Step 2: load **a** and a line of **b**[1], **a** hit, **b**[1] miss
Step 3: load **a** and a line of **b**[2], **a** hit, **b**[2] miss.
Step 4: load **a** and a line of **b**[3], **a** miss, **b**[3] miss.

simplified np.multiply

```
for i in range(n):
    # a is a vector, b and c are matrices
    c[i] = a * b[i]
```



Why we need to optimize shared memory?

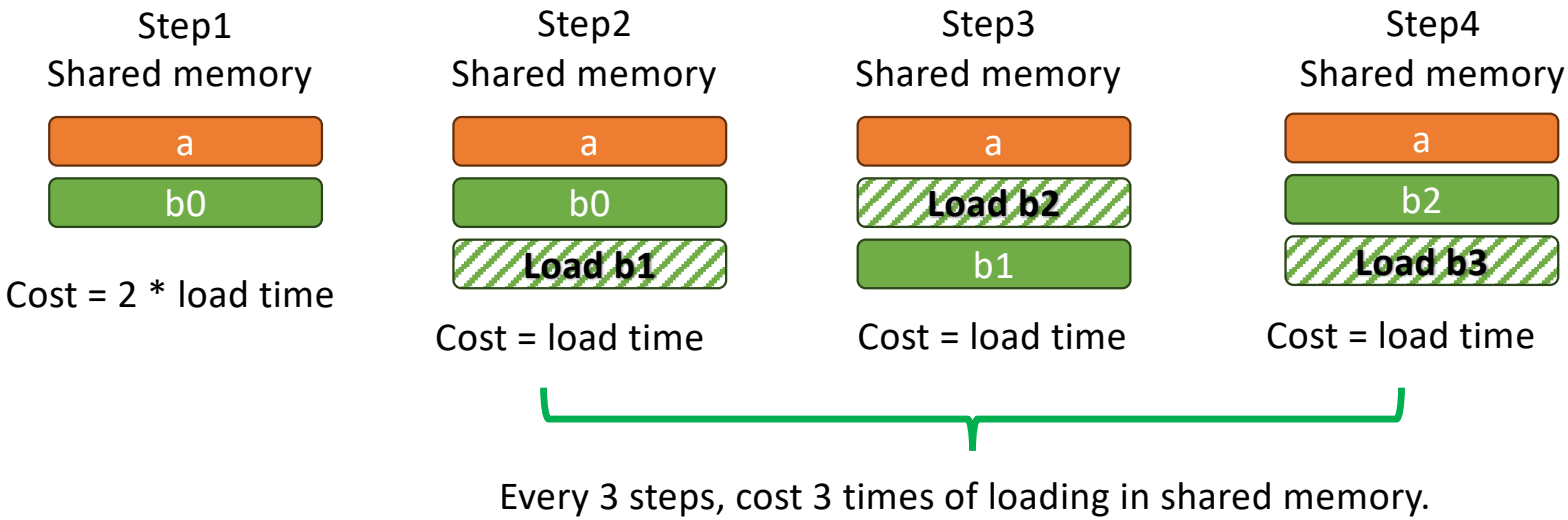
- Dummy strategy on CuPy

simplified np.multiply

```
for i in range(n):
    # a is a vector, b and c are matrices
    c[i] = a * b[i]
```

We assume that shared memory size = 3 vectors' size

If **we lock a in shared memory manually**, cost in each step is always 1 load time.

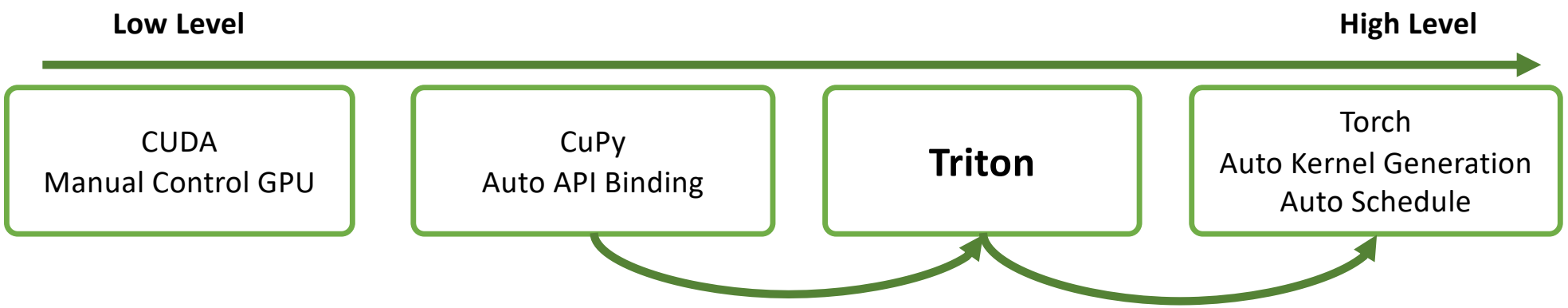


CuPy API for shared memory

- Declare the array with
- `jit.shared_memory`

```
    shared_mem = jit.shared_memory(cp.float32, 512)

    bulk_x_u32 = cp.uint32(bulk_x)
    bulk_y_u32 = cp.uint32(bulk_y)

    offset_x = global_idx * bulk_x_u32
    offset_y = global_idy * bulk_y_u32

    block_offset = jit.threadIdx.x * bulk_x_u32

    for i in range(bulk_x_u32):
        shared_mem[block_offset+i] = A_cu[offset_x+i]
        jit.syncthreads()

    for i in range(bulk_y_u32):
        for j in range(bulk_x_u32):
            C_cu[offset_y+i, offset_x+j] = shared_mem[block_offset+j] * B_cu[offset_y+i, \\
offset_x+j]
```

Example 3: What is Triton?



- Use triton API warp memory management
- Provide fine-grained memory API
- Auto streaming

- Developers are using Triton refactor some slow kernels or applications in PyTorch

Example 3: Triton vector add

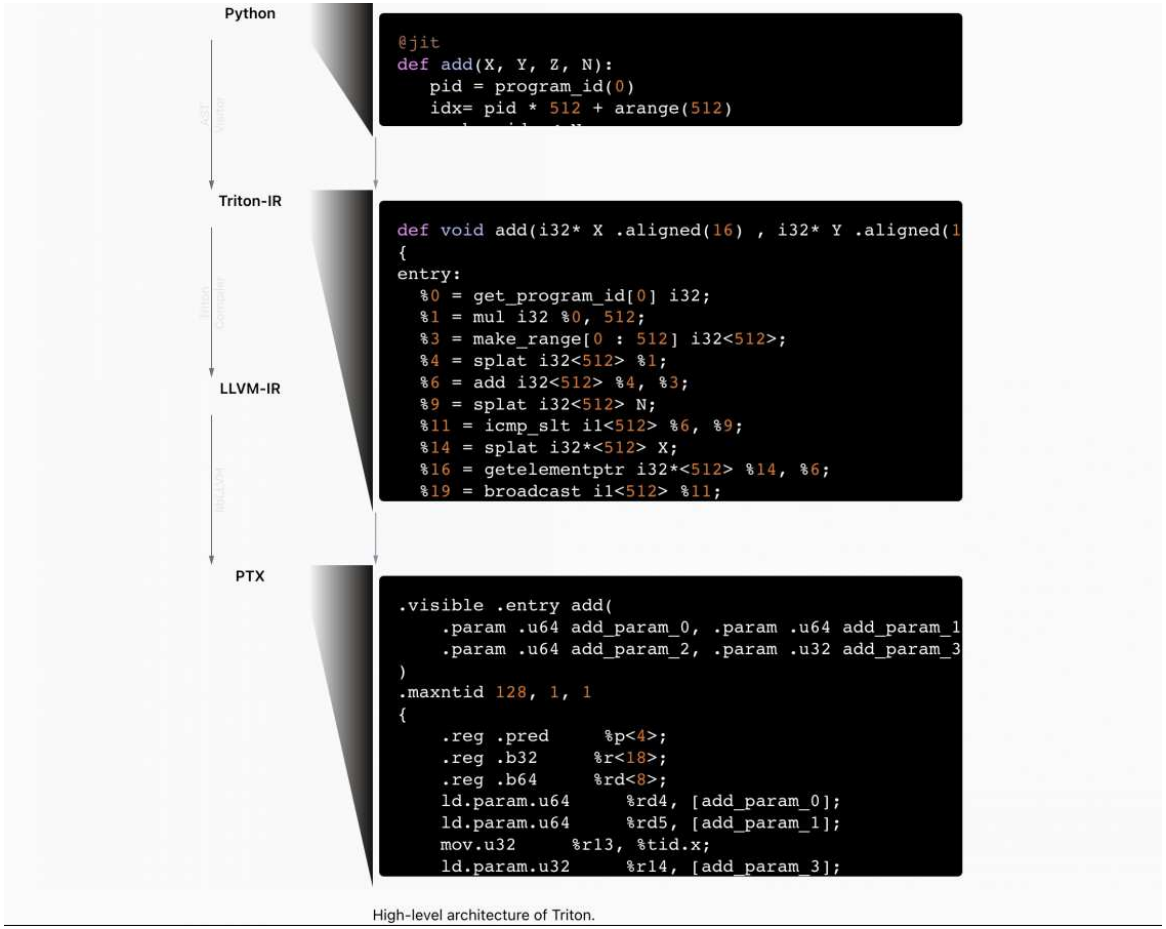
Triton Add

```python
import triton
import triton.language as tl
import torch
import time
import sys

@triton.jit
def vec_add_kernel(X_ptr, Y_ptr, Z_ptr, N, BLOCK: tl.constexpr):
    # Get the program ID (range depends on grid size)
    pid = tl.program_id(0)
    # Compute the block of indices handled by this program
    offsets = pid * BLOCK + tl.arange(0, BLOCK)
    # Mask to handle out-of-bounds memory accesses
    mask = offsets < N

    # Load inputs from device memory
    x = tl.load(X_ptr + offsets, mask=mask, other=0.0)
    y = tl.load(Y_ptr + offsets, mask=mask, other=0.0)
    # Perform the vector addition
    z = x + y
    # Store the result
    tl.store(Z_ptr + offsets, z, mask=mask)
```

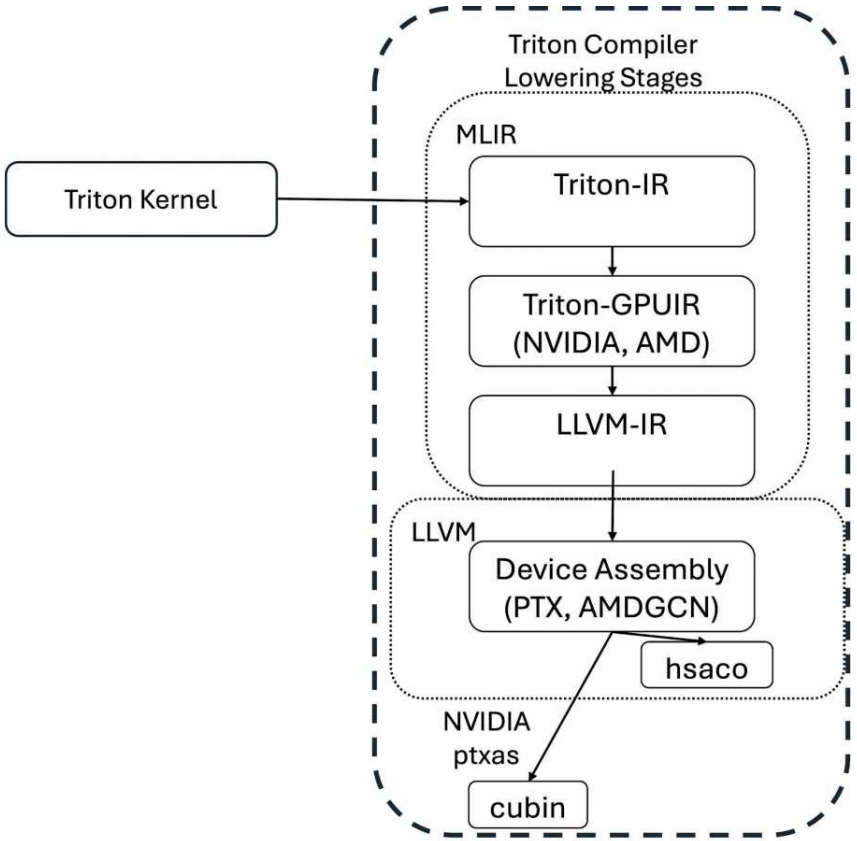
Triton Vector Add

```python
def triton_vector_add(n, repeat):
    # Create inputs on GPU (using PyTorch tensors for convenience)
    x = torch.rand(n, device="cuda", dtype=torch.float32)
    y = torch.rand(n, device="cuda", dtype=torch.float32)
    z = torch.empty_like(x)

    # Grid configuration: divide the work over blocks
    BLOCK = 1024
    # grid = (n + BLOCK - 1) // BLOCK
    grid = lambda meta: ( (n + BLOCK - 1) // BLOCK, )

    # Warm-up (ensure Triton compiles the kernel before timing)
    vec_add_kernel[grid](x, y, z, n, BLOCK)

    torch.cuda.synchronize()  # Ensure all GPU computations are complete
```

Triton and AI Compilers



High-level architecture of Triton.



[Triton Kernel Compilation Stages | PyTorch](https://pytorch.org/blog/triton-kernel-compilation-stages/)

Example 3: Triton Stream

Triton Vector Add

```
def compute_with_triton(x1, x2, n_elements):
    BLOCK_SIZE = 1024  # Set block size

    # Step 1 & 2: x1^2 and x2^2
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    square_kernel[grid](x1, x1_square, n_elements, BLOCK_SIZE)
    square_kernel[grid](x2, x2_square, n_elements, BLOCK_SIZE)

    # Step 3: z = x1^2 * sin(x1)
    trig_kernel[grid](x1, x1_sin, 0, n_elements, BLOCK_SIZE)  # sin
    elementwise_mul_kernel[grid](x1_square, x1_sin, z_part, n_elements, BLOCK_SIZE)

    # Step 4: w = x2^2 * cos(x2)
    trig_kernel[grid](x2, x2_cos, 1, n_elements, BLOCK_SIZE)  # cos
    elementwise_mul_kernel[grid](x2_square, x2_cos, w_part, n_elements, BLOCK_SIZE)

    # Step 5: Final z = z + w
    add_kernel[grid](z_part, w_part, result, n_elements, BLOCK_SIZE)
```

Triton Vector Add

```
# Triton Kernel for square operation
@triton.jit
def square_kernel(X, Y, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)  # Get program ID
    block_start = pid * BLOCK_SIZE  # Starting point for this block
    offsets = block_start + tl.arange(0, BLOCK_SIZE)  # Offsets for this thread
    mask = offsets < N  # Ensure we don't go out of bounds
    x = tl.load(X + offsets, mask=mask, other=0.0)  # Load inputs
    x = x * x  # Square the values
    tl.store(Y + offsets, x, mask=mask)  # Store the results

# Triton Kernel for sin and cos operation
@triton.jit
def trig_kernel(x_ptr, out_ptr, trig_type: tl.constexpr, n_elements, BLOCK_SIZE:
tl.constexpr):
    pid = tl.program_id(0)  # Get program ID
    block_start = pid * BLOCK_SIZE  # Starting point for this block
    offsets = block_start + tl.arange(0, BLOCK_SIZE)  # Offsets for this thread
    mask = offsets < n_elements  # Ensure we don't go out of bounds
    x = tl.load(x_ptr + offsets, mask=mask, other=0.0)  # Load inputs
    if trig_type == 0:  # sin
        out = tl.math.sin(x)
    elif trig_type == 1:  # cos
        out = tl.math.cos(x)
    tl.store(out_ptr + offsets, out, mask=mask)  # Store output

# Triton Kernel for elementwise multiplication
@triton.jit
def elementwise_mul_kernel(a_ptr, b_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)  # Get program ID
    block_start = pid * BLOCK_SIZE  # Starting point for this block
    offsets = block_start + tl.arange(0, BLOCK_SIZE)  # Offsets for this thread
    mask = offsets < n_elements  # Ensure we don't go out of bounds
    a = tl.load(a_ptr + offsets, mask=mask, other=0.0)  # Load inputs
    b = tl.load(b_ptr + offsets, mask=mask, other=0.0)  # Load inputs
    out = a * b  # Multiply
    tl.store(out_ptr + offsets, out, mask=mask)  # Store output

# Triton Kernel for addition
@triton.jit
def add_kernel(a_ptr, b_ptr, out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)  # Get program ID
    block_start = pid * BLOCK_SIZE  # Starting point for this block
    offsets = block_start + tl.arange(0, BLOCK_SIZE)  # Offsets for this thread
    mask = offsets < n_elements  # Ensure we don't go out of bounds
    a = tl.load(a_ptr + offsets, mask=mask, other=0.0)  # Load inputs
    b = tl.load(b_ptr + offsets, mask=mask, other=0.0)  # Load inputs
    out = a + b  # Add
    tl.store(out_ptr + offsets, out, mask=mask)  # Store output
```

(Just ignore kernel implementation on the right part)

After this course

- I understand that many of you are using Python in your other coursework. Some of you may have access to GPU resources, and your tasks could range from Machine Learning to Image Processing.

- However, what you've learned in this course is the simplest way to implement your tasks on a GPU.

- You are CUDA Hero now.

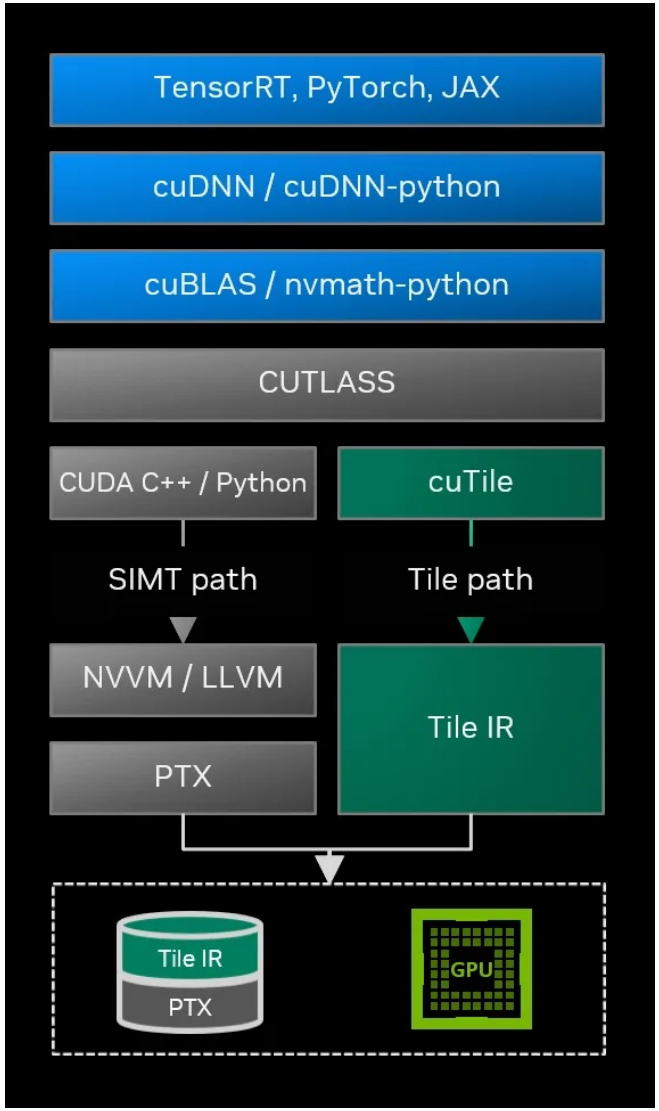

cuTile: New Era of GPU Programming
INFORMATICS FORUM

Why we need cuTile and TileIR

CUDA 13.1 introduced CUDA Tile, enabling higher-level, tile-based programming for GPUs by abstracting away details of specialized hardware like tensor cores.

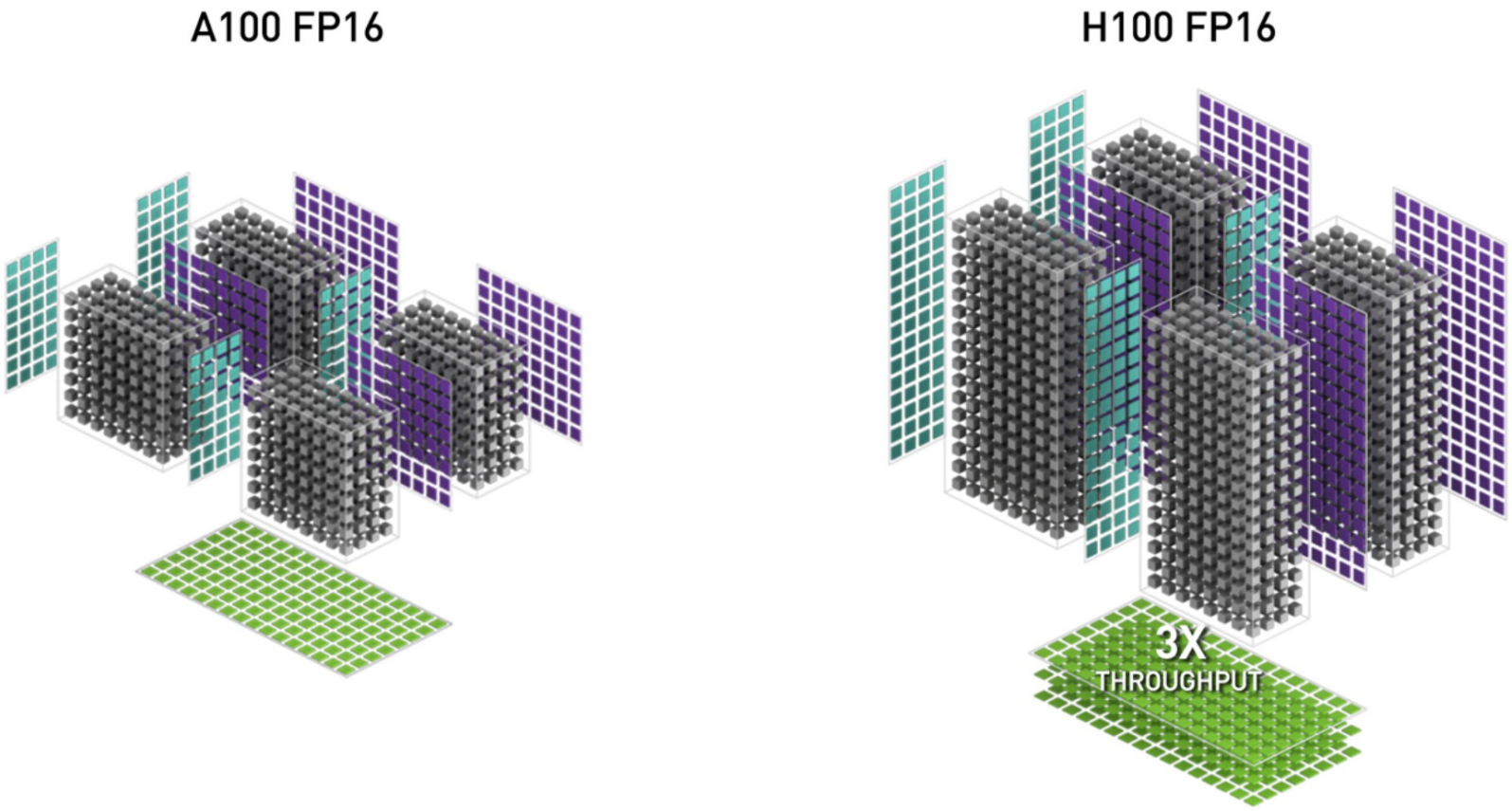
Tile programming with CUDA Tile IR allows developers to write more portable and hardware-agnostic code, simplifying compatibility across different generations of NVIDIA GPUs.

Most programmers will interact with CUDA Tile through tools like NVIDIA cuTile Python, while advanced users can leverage CUDA Tile IR to build custom compilers, frameworks, or libraries.



Focus on Your Algorithm—NVIDIA CUDA Tile Handles the Hardware | NVIDIA Technical Blog

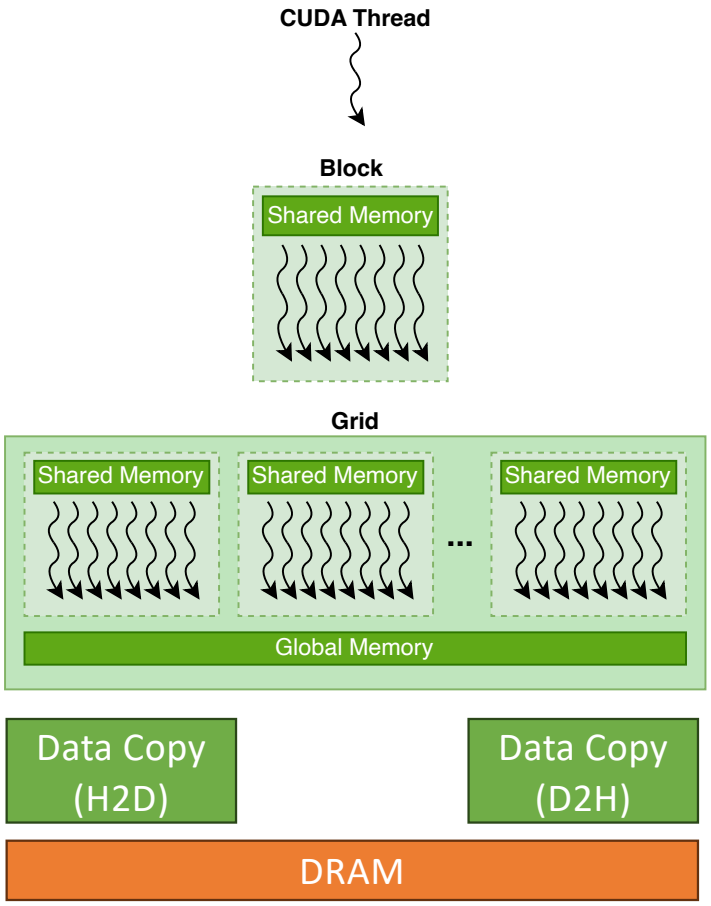
We rewrite codes for different architectures

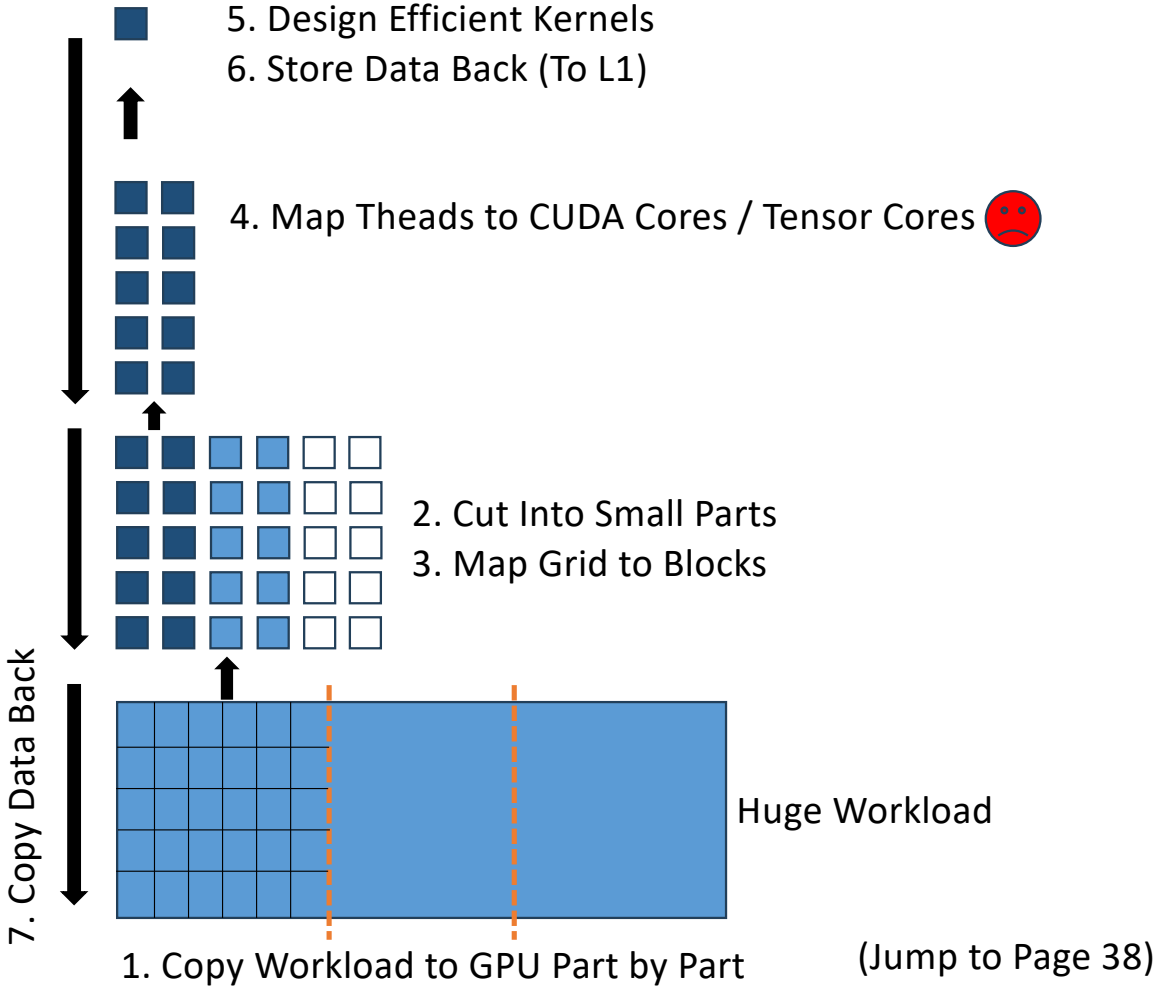


H100 FP16 Tensor Core has 3x throughput compared to A100 FP16 Tensor Core

NVIDIA Hopper Architecture In-Depth | NVIDIA Technical Blog

MAPPING IS DIFFICULT

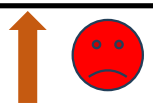




(Jump to Page 38)

We rewrite codes for different architectures

Architecture	Tensor Core FLOP/Cycle/SM	Max MMA Shape	Max FLOPs per PTX Instruction (2 * m * n * k)
Volta	F16: 1024	m8n8k4	**512**
Ampere	F16: 2048	m16n8k16	**4096**
Hopper	F16: 4096 F8: 8192	**Warpgroup Level** F16: m64n256k16 F8: m64n256k64	**Warpgroup Level** F16: 524,288 F8: 2,097,152
Blackwell	F16: 8192 F8: 16384 F4: 32768	**2 SM** F16: m256n256k16 F8: m256n256k32 F4: m256n256k96	**2 SM** F16: 2,097,152 F8: 4,191,304 F4: 12,582,912

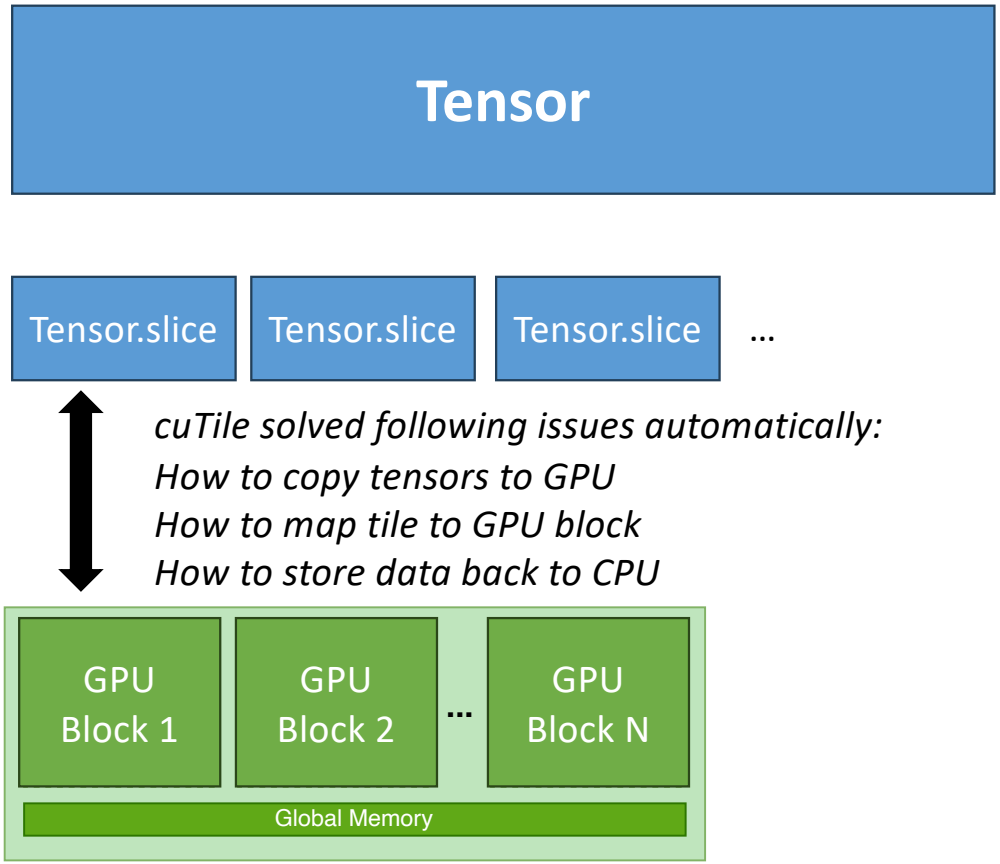


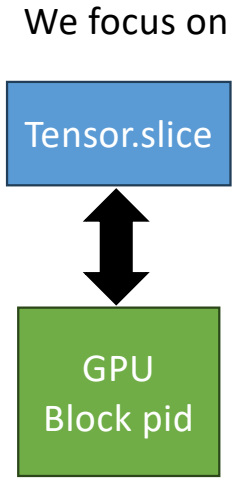
The mapping methods vary depending on the **MMA-supporting shape** and **the number of tensor cores**.

Annotations:

- FLOP/Cycle/SM: the number of floating-point operations a single Streaming Multiprocessor (SM) performs per clock cycle.
- Warpgroup: Groups 4 warps (128 threads) to execute larger matrix operations efficiently within Hopper architecture.
- 2 SM (Blackwell): Couples two physical cores to share resources and process massive matrices as a single unit.
- **MMA**: Matrix Multiply-Accumulate. A hardware instruction performing **D = A * B + C** on matrix blocks simultaneously.

In short, we write **Kernel** on **Tile**






```python
@ct.kernel
def kernel_code(input, output, size):
  pid = ct.bid(0)
  # load tile
  # compute
  # store tile
```

If you're familiar with numpy, you've learned 90% already!

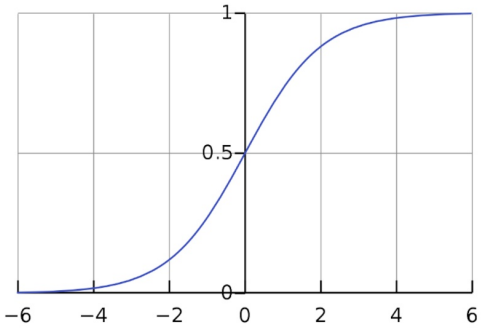
NumPy Version

```python
import cuda.tile as ct


def sigmoid(input, output):


    # Compute
    exp_neg_x = np.exp(-input)
    output = 1.0 / (1.0 + exp_neg_x)

```

cuTile Version

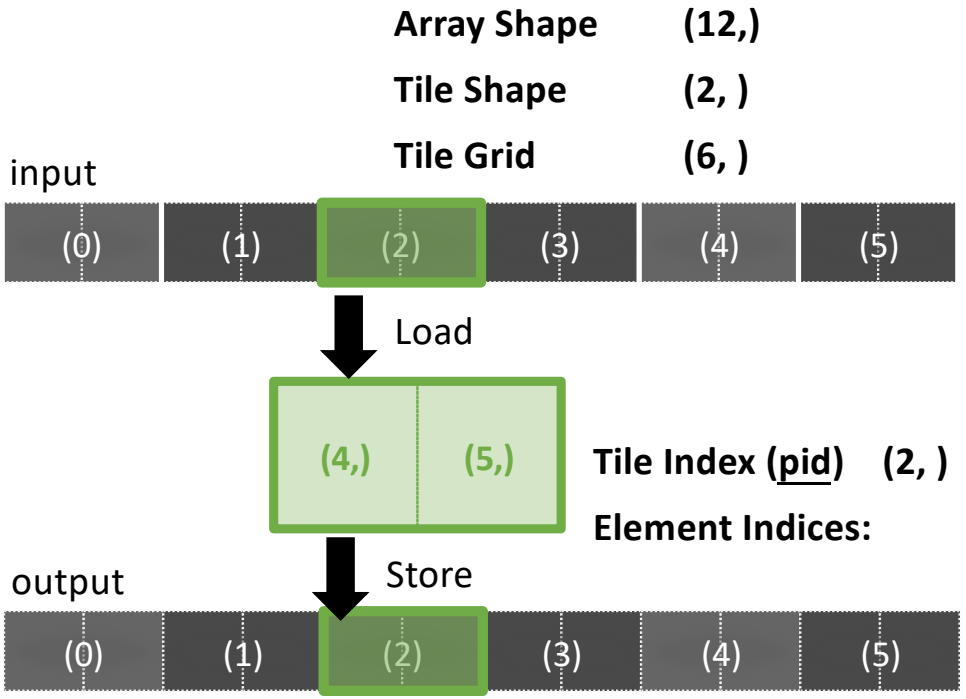
```python
import cuda.tile as ct

@ct.kernel
def sigmoid_kernel(input_ptr, output_ptr, tile_size: ct.Constant[int]):
    # Load
    pid = ct.bid(0)
    x_tile = ct.load(input_ptr, index=(pid,), shape=(tile_size,))
    # Compute
    neg_x = -x_tile
    exp_neg_x = ct.exp(neg_x)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)
    # Store
    ct.store(output_ptr, index=(pid,), tile=sigmoid_tile)
```

$$S(x) = \frac{1}{1 + e^{-x}}$$



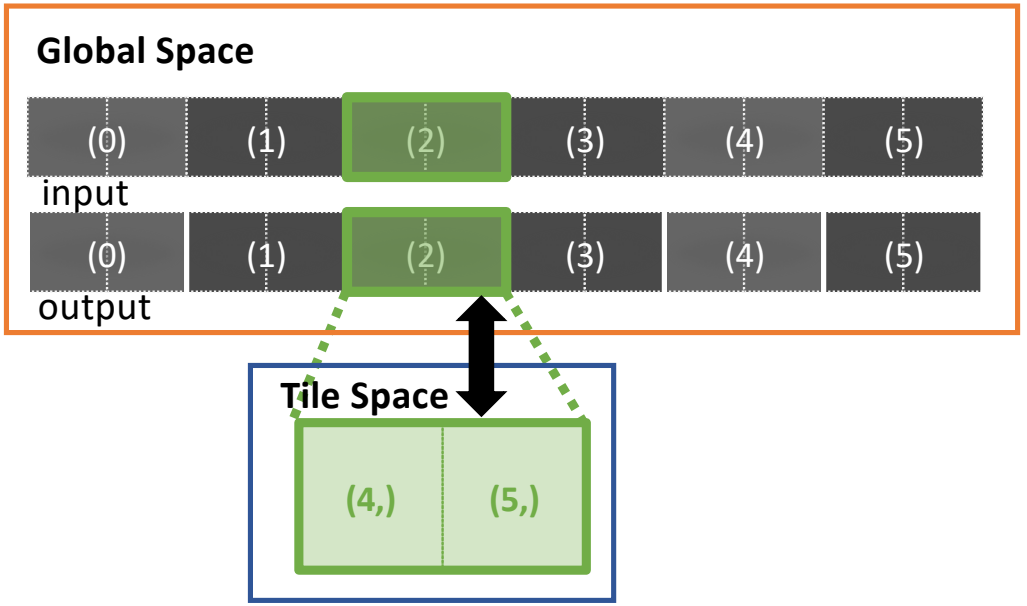
Understanding cuTile Part by Part - Index

```python
@ct.kernel
def sigmoid_kernel(input, output, tile_size: ct.Constant[int]):
    # Get the block ID (tile index in the 1D grid)
    # bid(0) returns the index of the current tile being processed
    pid = ct.bid(0)

    # Load a tile of data from global memory
    # index=(pid,) specifies which tile to load (tile-based indexing)
    # shape=(tile_size,) specifies the shape of the tile
    x_tile = ct.load(input, index=(pid,), shape=(tile_size,))

    # Compute sigmoid: σ(x) = 1 / (1 + exp(-x))
    # cuTile provides ct.exp() for element-wise exponential
    exp_neg_x = ct.exp(-x_tile)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)

    # Store the result back to global memory
    ct.store(output, index=(pid,), tile=sigmoid_tile)
```



Understanding cuTile Part by Part – Global & Tile Space

```python
@ct.kernel
def sigmoid_kernel(input, output, tile_size: ct.Constant[int]):
    # Get the block ID (tile index in the 1D grid)
    # bid(0) returns the index of the current tile being processed
    pid = ct.bid(0)

    # Load a tile of data from global memory
    # index=(pid,) specifies which tile to load (tile-based indexing)
    # shape=(tile_size,) specifies the shape of the tile
    x_tile = ct.load(input, index=(pid,), shape=(tile_size,))

    # Compute sigmoid: σ(x) = 1 / (1 + exp(-x))
    # cuTile provides ct.exp() for element-wise exponential
    exp_neg_x = ct.exp(-x_tile)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)

    # Store the result back to global memory
    ct.store(output, index=(pid,), tile=sigmoid_tile)
```



We can only operate on Tile Space data/arrays in ct.kernel code.

Execute cuTile Kernel

```python
@ct.kernel
def sigmoid_kernel(input, output, tile_size: ct.Constant[int]):
    pid = ct.bid(0)
    # Load a tile of data from global memory
    x_tile = ct.load(input, index=(pid,), shape=(tile_size,))
    # Compute sigmoid: σ(x) = 1 / (1 + exp(-x))
    exp_neg_x = ct.exp(-x_tile)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)
    # Store the result back to global memory
    ct.store(output, index=(pid,), tile=sigmoid_tile)

# Vector size and Tile size
N = 1024
TILE_SIZE = 32

# Calculate grid dimension (number of tiles needed)
num_tiles = ct.cdiv(N, TILE_SIZE)
grid = (num_tiles, 1, 1)

data_in = cp.linspace(-5, 5, N, dtype=cp.float32)
data_out = cp.zeros_like(data_in)

# Launch kernel
ct.launch(cp.cuda.get_current_stream(),
          grid, sigmoid_kernel,
          (data_in, data_out, TILE_SIZE))
```

1. `ct.load(array, index=(pid,), shape=(tile_size,))`

Description: Load data from GPU memory into a Tile.

Parameters:

- **array:** Source array.
- **index:** Tile index (indicates which data block).
- **shape:** Shape of the Tile (size of the data block).

2. `ct.store(array, index=(pid,), tile=result)`

Description: Store Tile data back into GPU memory.

Parameters:

- **array:** Target array (destination).
- **index:** Tile index (indicates the position to store data).
- **tile:** The Tile data to be stored.

3. `pid = ct.bid(dim)`

Description: Get the index of the current processor (Block ID).

Parameters & Details:

- **dim:** Dimension parameter (0=x, 1=y, 2=z).

Return Value: Index number of the current block.

4. `ct.launch(stream, grid, kernel, args)`

Description: Launch the GPU kernel function.

Parameters:

- **stream:** CUDA stream object. (cp.cuda.get_current_stream() is okay)
  - If you want to learn more about streaming control, go page 64
- **grid:** Grid size (number of processors).
- **kernel:** The kernel function.
- **args:** Tuple of arguments.

Understanding cuTile Part by Part - Tile

```python
@ct.kernel
def sigmoid_kernel(input, output, tile_size: ct.Constant[int]):
    # Get the block ID (tile index in the 1D grid)
    # bid(0) returns the index of the current tile being processed
    pid = ct.bid(0)

    # Load a tile of data from global memory
    # index=(pid,) specifies which tile to load (tile-based indexing)
    # shape=(tile_size,) specifies the shape of the tile
    x_tile = ct.load(input, index=(pid,), shape=(tile_size,))

    # Compute sigmoid: σ(x) = 1 / (1 + exp(-x))
    # cuTile provides ct.exp() for element-wise exponential
    exp_neg_x = ct.exp(-x_tile)
    sigmoid_tile = 1.0 / (1.0 + exp_neg_x)

    # Store the result back to global memory
    ct.store(output, index=(pid,), tile=sigmoid_tile)
```

Array Shape	**(12,)**
Tile Shape	**(2,)**
Tile Grid	**(6,)**

You can launch significantly more Blocks at the software level than the physical hardware can handle simultaneously.

Logical Grid: This is the grid parameter set in your code via cuda.tile.launch(stream, grid, ...). For example, you can set the Grid to (2048,). This essentially tells the GPU: "I have 2,048 independent tasks to perform."
Physical Execution: Although the number of Blocks that can run simultaneously on the GPU is limited (constrained by the number of SMs and available resources), the GPU hardware scheduler automatically handles this. It schedules these 2,048 Blocks to execute on the hardware in batches (often called "Waves"). As soon as one batch finishes, the hardware automatically loads the next one, continuing until all 2,048 Blocks are completed.
Conclusion: Even if the GPU can physically run only **80 Blocks** at a time, you can still launch a Grid containing 2,048 Blocks. In terms of code logic, each Block processes its corresponding Tile (i.e., the ct.bid(0)-th Tile).
However, if you want to chase the speed of light, you should choose grid carefully. Make sure `grid_size % GPU_block_num == 0`.

2D Tile Example

```python
import cupy as cp
import numpy as np
import cuda.tile as ct


@ct.kernel
def grid_map_2d(output, tile_size_x: ct.Constant[int], tile_size_y: ct.Constant[int]):
    # Get the 2D block IDs (tile coordinates)
    # bid(0) corresponds to the x-dimension of the grid
    # bid(1) corresponds to the y-dimension of the grid
    pid_x = ct.bid(0)
    pid_y = ct.bid(1)

    # We want to fill the output tile with a value that represents its coordinate.
    # For visualization, let's store: pid_x * 1000 + pid_y
    # This makes it easy to read: 2003 means x=2, y=3
    val = pid_x * 1000 + pid_y

    # Create a tile filled with 'val' using ct.full and store it.
    # ct.full creates a tile of the specified shape filled with the given value.
    # Note: numpy/cupy use (row, col) which is (y, x)
    val_tile = ct.full((tile_size_y, tile_size_x), val, ct.int32)

    # Store to global memory
    # index=(pid_y, pid_x) matches the (row, col) layout
    ct.store(output, index=(pid_y, pid_x), tile=val_tile)
```

```python
height = 128
width = 128

# Size of each tile
tile_h = 16
tile_w = 16

# Calculate grid dimensions (number of tiles in each direction)
grid_x = ct.cdiv(width, tile_w)
grid_y = ct.cdiv(height, tile_h)

grid = (grid_x, grid_y, 1)

# Allocate output memory on device
# Note: shape is (height, width) i.e., (y, x)
output = cp.zeros((height, width), dtype=cp.int32)

# Launch kernel
ct.launch(cp.cuda.get_current_stream(),
          grid,
          grid_map_2d,
          (output, tile_w, tile_h))
#...
```

2D Tile Example

```python
import cupy as cp
import numpy as np
import cuda.tile as ct


@ct.kernel
def grid_map_2d(output, tile_size_x: ct.Constant[int], tile_size_y: ct.Constant[int]):
    # Get the 2D block IDs (tile coordinates)
    # bid(0) corresponds to the x-dimension of the grid
    # bid(1) corresponds to the y-dimension of the grid
    pid_x = ct.bid(0)
    pid_y = ct.bid(1)

    # We want to fill the output tile with a value that represents its coordinate.
    # For visualization, let's store: pid_x * 1000 + pid_y
    # This makes it easy to read: 2003 means x=2, y=3
    val = pid_x * 1000 + pid_y

    # Create a tile filled with 'val' using ct.full and store it.
    # ct.full creates a tile of the specified shape filled with the given value.
    # Note: numpy/cupy use (row, col) which is (y, x)
    val_tile = ct.full((tile_size_y, tile_size_x), val, ct.int32)

    # Store to global memory
    # index=(pid_y, pid_x) matches the (row, col) layout
    ct.store(output, index=(pid_y, pid_x), tile=val_tile)
```

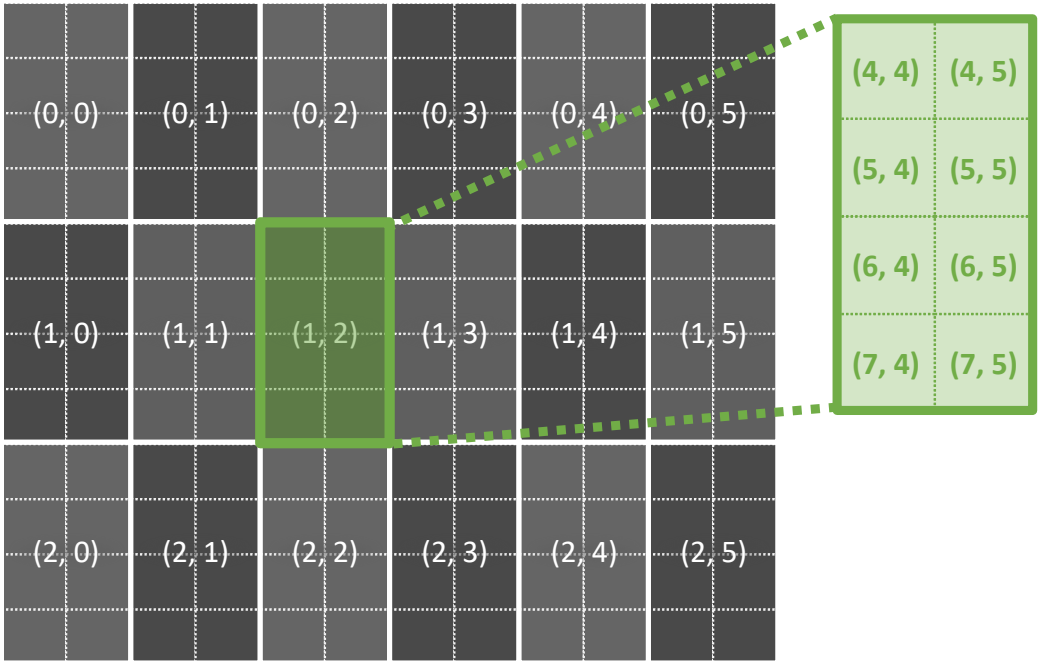
Array Shape (12, 12)

Tile Shape (4, 2)

Tile Grid (3, 6)

Tile Index (1, 2)

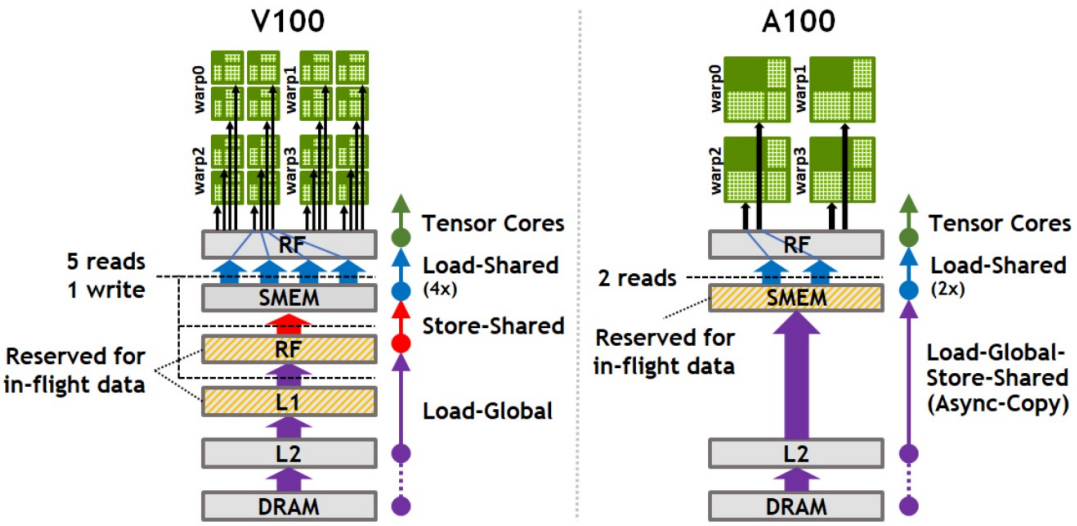
Element Indices:



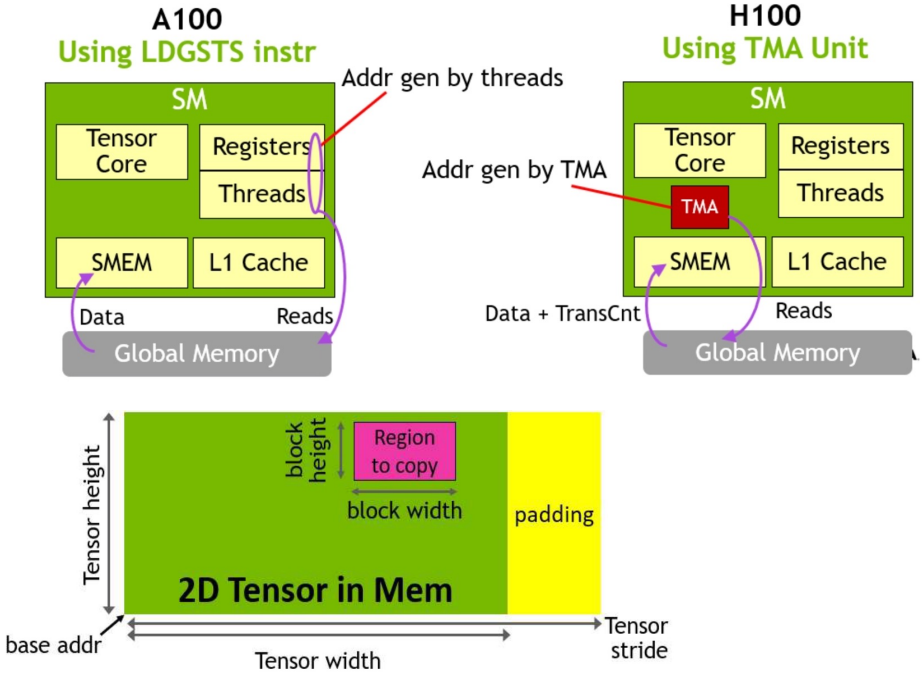
Why CuTile is simple: Let's see what is behind `load` and `store`

```
x_tile = ct.load(input, index=(pid,), shape=(tile_size,))
```

Different GPUs have totally different load and store method/instructions.
V100: copy + sync; A100: async copy + SMEM management instruction; H100: TMA(Tensor Memory Accelerator)



A100 improves SM bandwidth efficiency with a new **load-global-store-shared asynchronous copy** instruction that bypasses L1 cache and register file (RF). Additionally, A100's more efficient Tensor Cores reduce shared memory (SMEM) loads



The TMA operation is asynchronous and leverages the shared memory-based asynchronous barriers introduced in A100.

So, if you're still using CUDA on latest hardware…

```cpp

// -------------------- FP32: align-aware float4 + ILP unroll --------------------
template<int ILP>
__global__ void sigmoid_f32_opt(const float* __restrict__ x,
                                float* __restrict__ y,
                                int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    // Try to use float4 path only if both pointers are 16B-aligned
    uintptr_t xa = reinterpret_cast<uintptr_t>(x);
    uintptr_t ya = reinterpret_cast<uintptr_t>(y);
    bool aligned16 = ((xa | ya) & 0xF) == 0;

    if (aligned16) {
        int n4 = n >> 2;
        const float4* __restrict__ x4 = reinterpret_cast<const float4*>(x);
        float4* __restrict__ y4 = reinterpret_cast<float4*>(y);

        // Each loop handles ILP float4's per thread to increase ILP
        for (int i = tid * ILP; i < n4; i += stride * ILP) {
#pragma unroll
            for (int k = 0; k < ILP; ++k) {
                int idx = i + k * stride;
                if (idx < n4) {
                    float4 v = x4[idx];
                    v.x = sigmoid_fast(v.x);
                    v.y = sigmoid_fast(v.y);
                    v.z = sigmoid_fast(v.z);
                    v.w = sigmoid_fast(v.w);
                    y4[idx] = v;
                }
            }
        }

        // Tail (scalar)
        int base = n4 << 2;
        for (int i = base + tid; i < n; i += stride) {
            y[i] = sigmoid_fast(x[i]);
        }
    } else {
        // Fallback scalar (still grid-stride)
        for (int i = tid; i < n; i += stride) {
            y[i] = sigmoid_fast(x[i]);
        }
    }
}
```

```cpp

// ---------------- FP16: align-aware half2; optional wider 16B load ----------------
__device__ __forceinline__ __half2 sigmoid_fast_h2(__half2 hx) {
#if __CUDA_ARCH__ >= 530
    float2 fx = __half22float2(hx);
    fx.x = __fdividef(1.0f, 1.0f + exp_neg_fast(fx.x));
    fx.y = __fdividef(1.0f, 1.0f + exp_neg_fast(fx.y));
    return __floats2half2_rn(fx.x, fx.y);
#else
    float2 fx = __half22float2(hx);
    fx.x = 1.0f / (1.0f + expf(-fx.x));
    fx.y = 1.0f / (1.0f + expf(-fx.y));
    return __floats2half2_rn(fx.x, fx.y);
#endif
}

template<int ILP>
__global__ void sigmoid_f16_opt(const __half* __restrict__ x,
                                __half* __restrict__ y,
                                int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    // half2 requires 4B alignment ideally; but cudaMalloc is aligned.
    uintptr_t xa = reinterpret_cast<uintptr_t>(x);
    uintptr_t ya = reinterpret_cast<uintptr_t>(y);
    bool aligned4 = ((xa | ya) & 0x3) == 0;

    if (aligned4) {
        int n2 = n >> 1;
        const __half2* __restrict__ x2 = reinterpret_cast<const __half2*>(x);
        __half2* __restrict__ y2 = reinterpret_cast<__half2*>(y);

        for (int i = tid * ILP; i < n2; i += stride * ILP) {
#pragma unroll
            for (int k = 0; k < ILP; ++k) {
                int idx = i + k * stride;
                if (idx < n2) {
                    __half2 v = x2[idx];
                    y2[idx] = sigmoid_fast_h2(v);
                }
            }
        }

        // Odd tail
        if ((n & 1) && tid == 0) {
            float fx = __half2float(x[n - 1]);
            float fy = __fdividef(1.0f, 1.0f + exp_neg_fast(fx));
            y[n - 1] = __float2half_rn(fy);
        }
    } else {
        // Fallback scalar half
        for (int i = tid; i < n; i += stride) {
            float fx = __half2float(x[i]);
            float fy = __fdividef(1.0f, 1.0f + exp_neg_fast(fx));
            y[i] = __float2half_rn(fy);
        }
    }
}
```

Secret Notes on cuTile

Tile Size must be power of 2.

```python
vector_size = 2**12
tile_size = 31
grid = (ct.cdiv(vector_size, tile_size), 1, 1)
```

```
  File "/home/chivier/Projects/cutile-tutorial/.venv/lib/python3.11/site-packages/cuda/tile
/_ir/op_impl.py", line 198, in require_constant_shape
    raise _make_type_error(f"Dimension #{i} of shape {tuple(shape)} is not a power of two",
cuda.tile._exception.TileTypeError: Invalid argument "shape" of load(): Dimension #0 of sha
pe (31,) is not a power of two
  In file "/home/chivier/Projects/cutile-tutorial/1-vectoradd/vectoradd.py", line 21, col 1
4--57:
        a_tile = ct.load(a, index=(pid,), shape=(tile_size,))
```

Ct.load and Store only accept Constant Shape.

```python
@ct.kernel
def vector_add(a, b, c, tile_size: ct.Constant[int]):
    # Get the 1D pid
    pid = ct.bid(0)
```

```
    raise _make_type_error("Expected a constant integer tuple,"
cuda.tile._exception.TileTypeError: Invalid argument "shape" of load(): Expected a constant
 integer tuple, but given value is not constant
  In file "/home/chivier/Projects/cutile-tutorial/1-vectoradd/vectoradd.py", line 21, col 1
4--57:
        a_tile = ct.load(a, index=(pid,), shape=(tile_size,))
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Read Operations — cuTile Python, you can find powerful weapons like: ct.mma, ct.matmul
Those operators are well optimized by Nvidia engineers.

We will upload everything to:
https://github.com/ed-aisys/edin-mls-26-spring

Discuss with AI:

https://notebooklm.google.com/notebook/5012cffe-8cb1-40a5-8e10-d55f681bc225

ASR Inference

- zai-org/GLM-ASR: GLM-ASR-Nano: A robust, open-source speech recognition model with 1.5B parameters

- Model Download:
- zai-org/GLM-ASR-Nano-2512 · Hugging Face

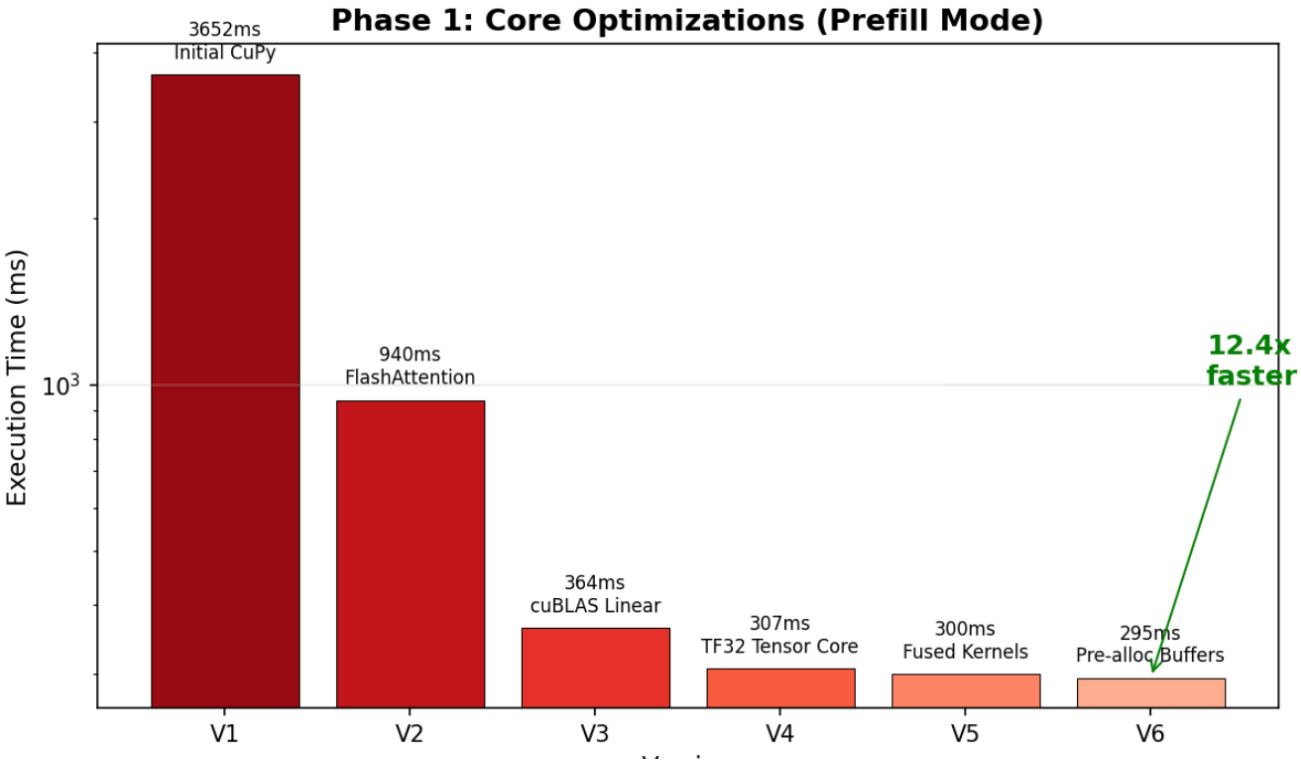
Our Template for you

Files Structure

File	Status	Description
`attention.py`	**TODO**	Attention kernels
`layers.py`	**TODO**	Layer kernels (RMSNorm, GELU, Linear)
`rope.py`	**TODO**	RoPE kernels
`flash_attention.py`	Complete	FlashAttention reference
`conv.py`	Complete	Convolution layers
`model.py`	Complete	Full model using your kernels
`weight_loader.py`	Complete	Weight loading utilities

```python
@ct.kernel
def gelu_kernel(x, output, tile_size: ct.Constant[int]):
    """
    GELU using tanh approximation.
    GELU(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))

    *** TODO: Implement this kernel ***

    Grid: (num_tiles,)
    """
    pid = ct.bid(0)

    # ============================================================================
    # TODO: Implement GELU kernel
    # ============================================================================
    #
    # x_tile = ct.load(x, index=(pid,), shape=(tile_size,))
    #
    # sqrt_2_over_pi = 0.7978845608028654
    # x3 = x_tile * x_tile * x_tile
    # inner = sqrt_2_over_pi * (x_tile + 0.044715 * x3)
    # result = x_tile * 0.5 * (1.0 + ct.tanh(inner))
    #
    # ct.store(output, index=(pid,), tile=result)

    # YOUR CODE HERE (approximately 6 lines)
    pass  # Remove this and implement
```

Optimiza Prefill



V1 - Initial Implementation
Pure CuPy with O(n^2) attention matrix materialization
16×16 tiles, no Tensor Core utilization

V2 - *FlashAttention* (3.9x↑)
Online softmax: O(1) memory complexity
exp2 optimization, 64×64 tiles, ct.mma acceleration

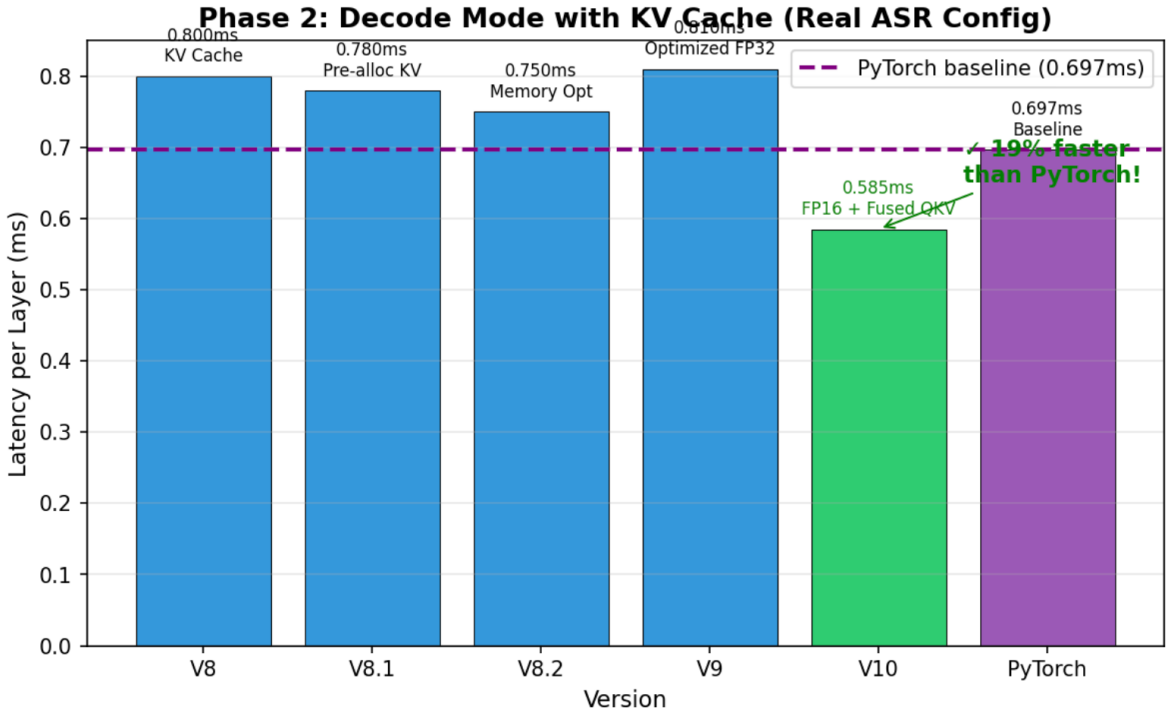
V3 - cuBLAS Linear (2.6x↑)
Replace slow CuTile linear with cuBLAS GEMM
Profiling revealed linear_kernel took 87.7% time

V4 - TF32 Tensor Core (1.2x↑)
FP32 MMA doesn't use Tensor Cores
TF32 enables Tensor Core path with acceptable precision

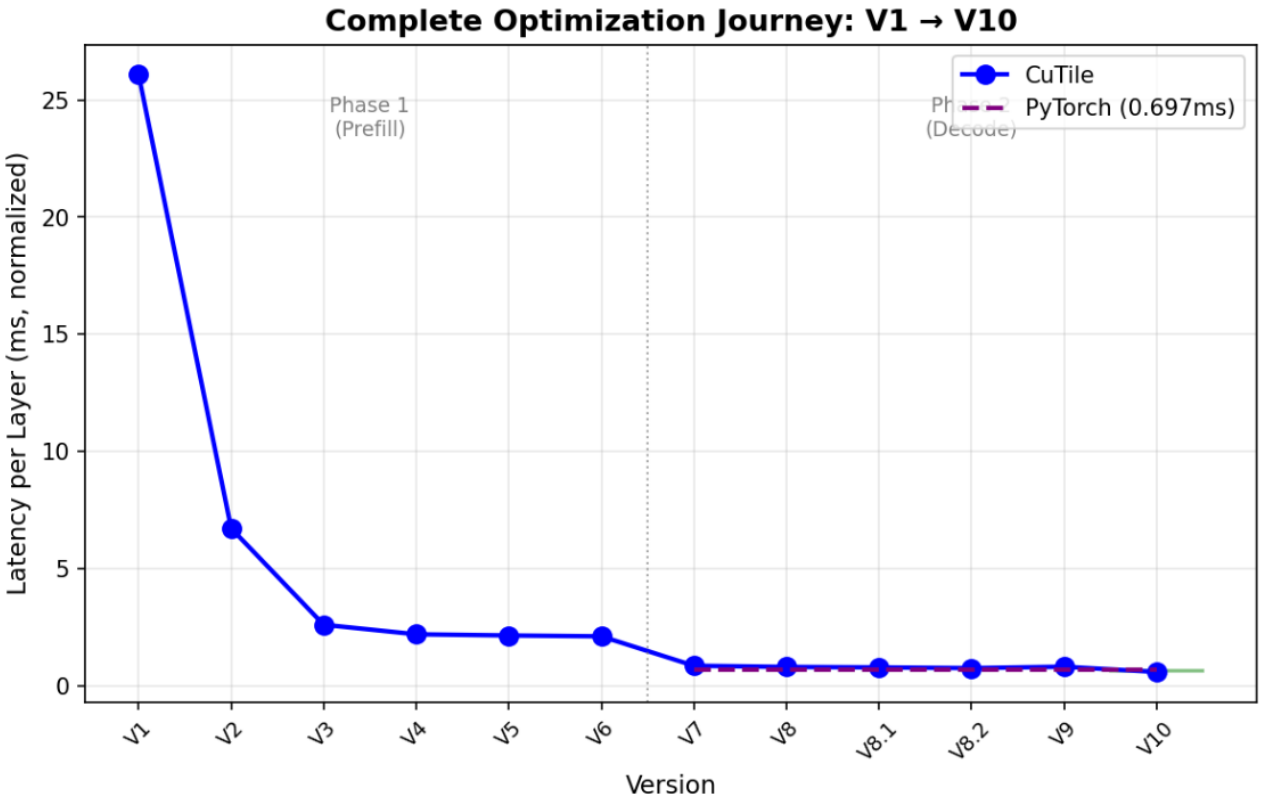
V5 - Fused Kernels (~1.02x↑)
Fused SwiGLU: gate + up + SiLU in one kernel
Reduced kernel launches and memory allocation

V6 - Pre-allocated Buffers (~1.02x↑)
Pre-allocate generation buffers to avoid dynamic allocation
Limitation: Still no KV cache, O(n^2) complexity

Optimiza end to end



V7 - Adaptive Backend
CuTile TF32 for prefill (M≥64), cuBLAS for decode
New test config with more tokens generated
V8 - KV Cache (1.25x↑)
Cache K/V per layer, only compute new token
Complexity: $O(n^2) \rightarrow O(n)$ per step
V8.1 - Pre-allocated KV (1.02x↑)
Slice assignment instead of concatenation
O(1) write per token vs O(n) memory copy
V8.2 - Memory Optimization (1.04x↑)
FlashAttention handles GQA natively, remove expand_kv
Optimized contiguity checks
V10 - Comprehensive Optimization (1.38x↑ vs V9)
CuTile RMSNorm: Support arbitrary hidden size (3584)
CuTile RoPE decode kernel: 1.9x faster
Fused QKV projection: 3 kernels → 1 kernel
Fused gate+up MLP projection
cuBLAS FP16 with FP32 accumulator
Dedicated decode attention kernel (faster than FlashAttention)



Note: Hardware is NOT H200. Absolute times are reference only. Focus on speedup ratios.

Final Result

V1 → V10 Total Speedup: ~8.6x

CuTile V10 vs PyTorch: 1.19x faster

Per-layer: 0.585ms (CuTile) vs 0.697ms (PyTorch)

Dark Knowledge About Free GPU

- Lightning - The AI Development Platform (https://lightning.ai/)
  - 15 hours free A100 GPU!
- Pricing for GPU Instances, Storage, Serverless (https://www.runpod.io/pricing)
  - 2-3 hours free A100/H100 GPU!
- Welcome To Colab - Colab(https://colab.research.google.com/)
  - 12 hours free GPU
- 5 Best Free Cloud GPU Providers For Students
  - 5 Best Free Cloud GPUs for Students in 2026: (100% Free Now)
  - Some other free resources

- (Remember, you have a TEAM)
  - If you have 3 members in your group, if each of you have your school email + outlook email + gmail, you will have 9 account on each platform

Advanced Topics in CUDA.

- Bank conflict
  - Multithreading - What is a bank conflict - Stack Overflow: https://stackoverflow.com/questions/3841877/what-is-a-bank-conflict-doing-cuda-opencl-programming)
- Register spill
  - Local Memory and Register Spilling: https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf
- Memory Coalescing
  - In CUDA, what is memory coalescing, and how is it achieved? - Stack Overflow: https://stackoverflow.com/questions/5041328/in-cuda-what-is-memory-coalescing-and-how-is-it-achieved

Advanced Topics in cuTile

- Performance Tuning — cuTile Python
- Tile IR — Tile IR
- NVIDIA/TileGym: Helpful kernel tutorials and examples for tile-based GPU programming

Some notes about environment configuration

- Miniconda and virtual environment
  - https://medium.com/@aminasaeed223/a-comprehensive-tutorial-on-miniconda-creating-virtual-environments-and-setting-up-with-vs-code-f98d22fac8e2
- How to install CuPy
  - https://docs.CuPy.dev/en/stable/install.html
- How to install triton
  - https://triton-lang.org/main/getting-started/installation.html
- How to install Pytorch
  - https://pytorch.org/get-started/locally/

- BTW, GPT is always a good teacher.

I Still Suggest You Learn a little CUDA Programming. It can help you write better cuTile Kernels

- [NVIDIA/cuda-samples](https://github.com/NVIDIA/cuda-samples)
- [An Even Easier Introduction to CUDA | NVIDIA Technical Blog](https://developer.nvidia.com/blog/even-easier-introduction-cuda/)
- [HMUNACHI/cuda-repo: From zero to hero CUDA for accelerating maths and machine learning on GPU.](https://github.com/HMUNACHI/cuda-repo)