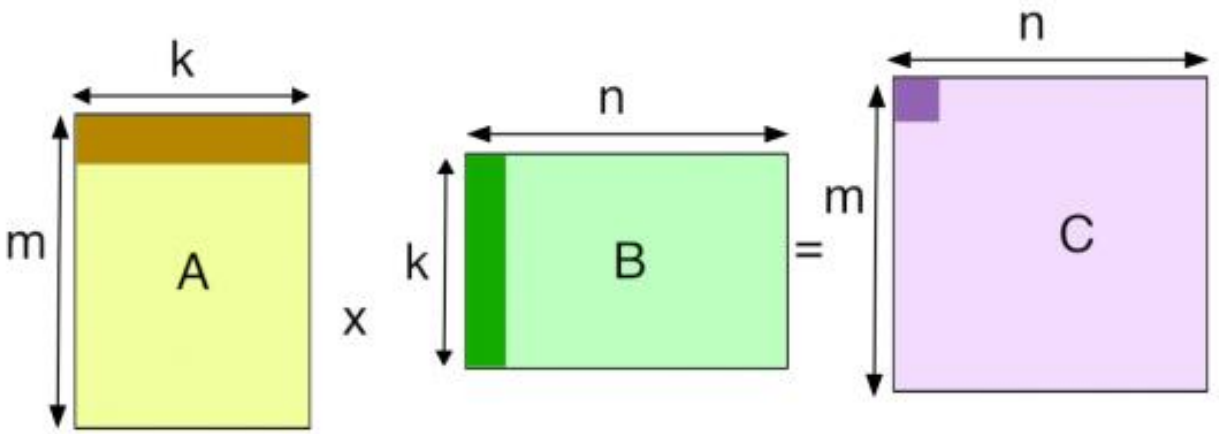
THE UNIVERSITY of EDINBURGH
informatics

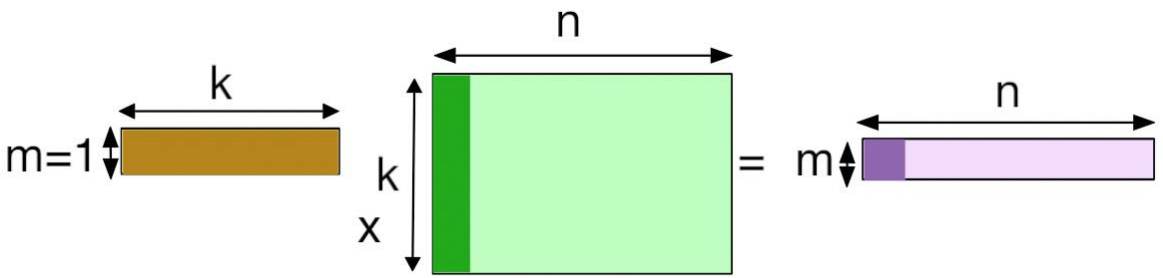
GPU Architecture and Programming

Speaker: Yeqi Huang, Congjie He, Luo Mai

GEMM & GEMV



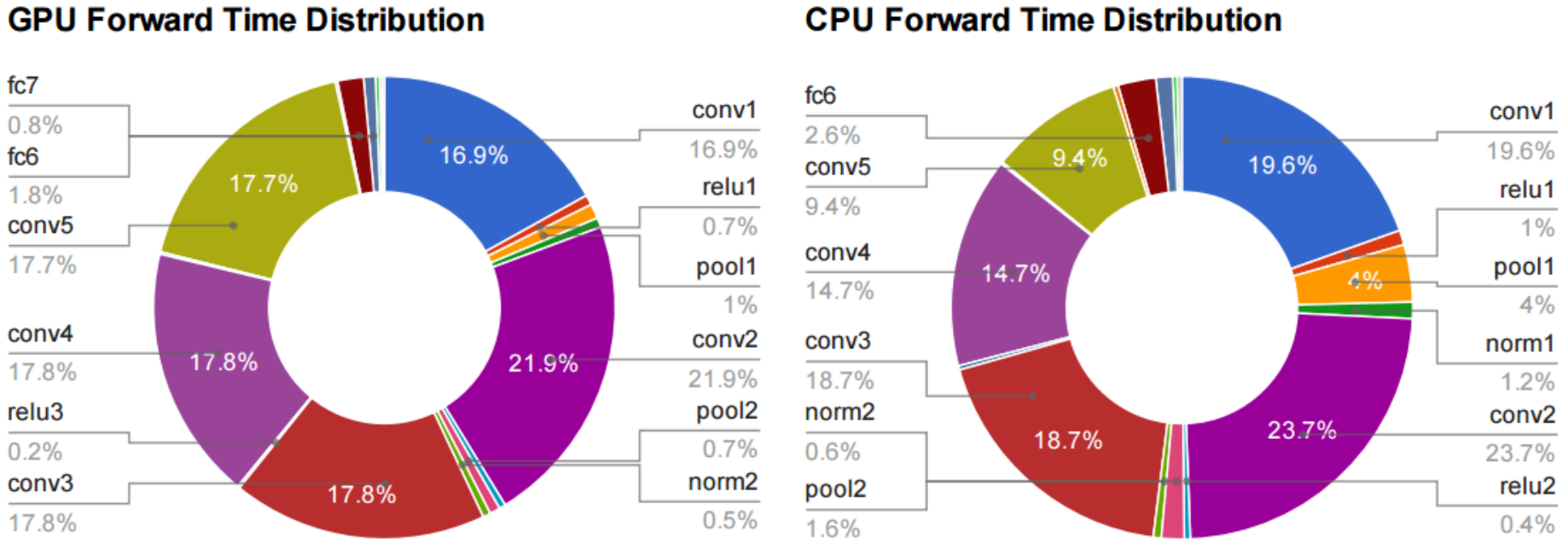
GEMM



GEMV

- GEMM (General Matrix Multiplication):
  - C = A * B, where A, B, and C are matrices.
- GEMV (General Matrix-Vector multiplication):
  - y = A * x, where A is a matrix and x, y are vectors.

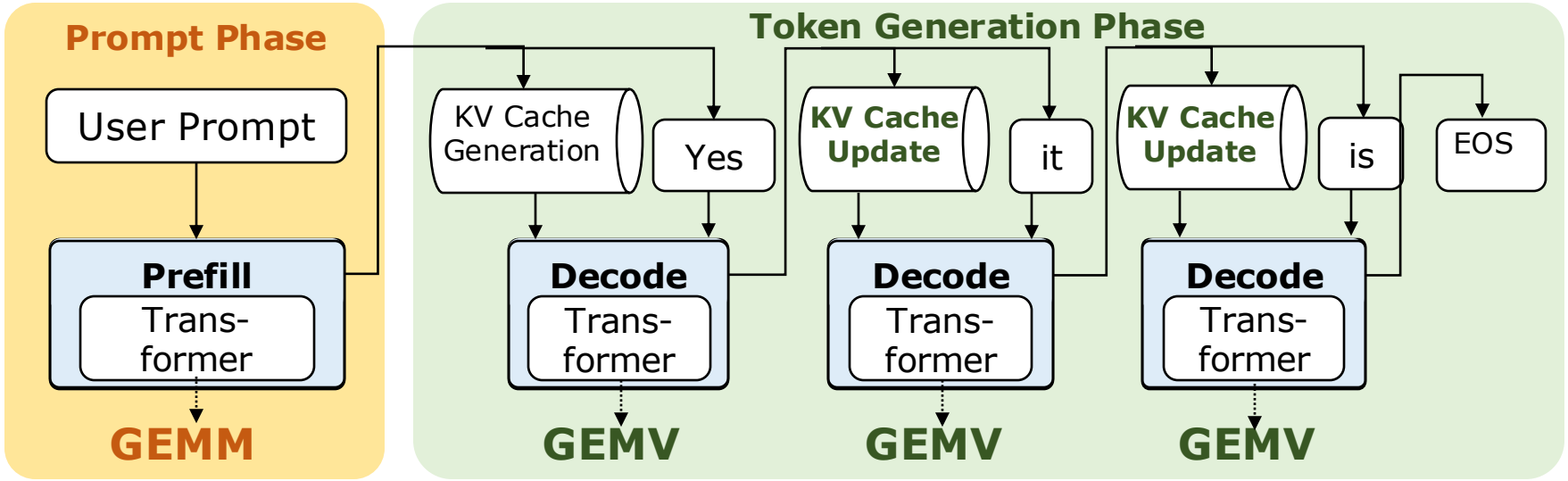
Why GEMM is at the heart of deep learning?



- Deep learning relies heavily on matrix multiplication
- Two key operations: convolutions and fully connected layers
- Matrix operations dominate training and inference
- Convolutions consume majority of AlexNet runtime
- Performance bottleneck is matrix computations

https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf

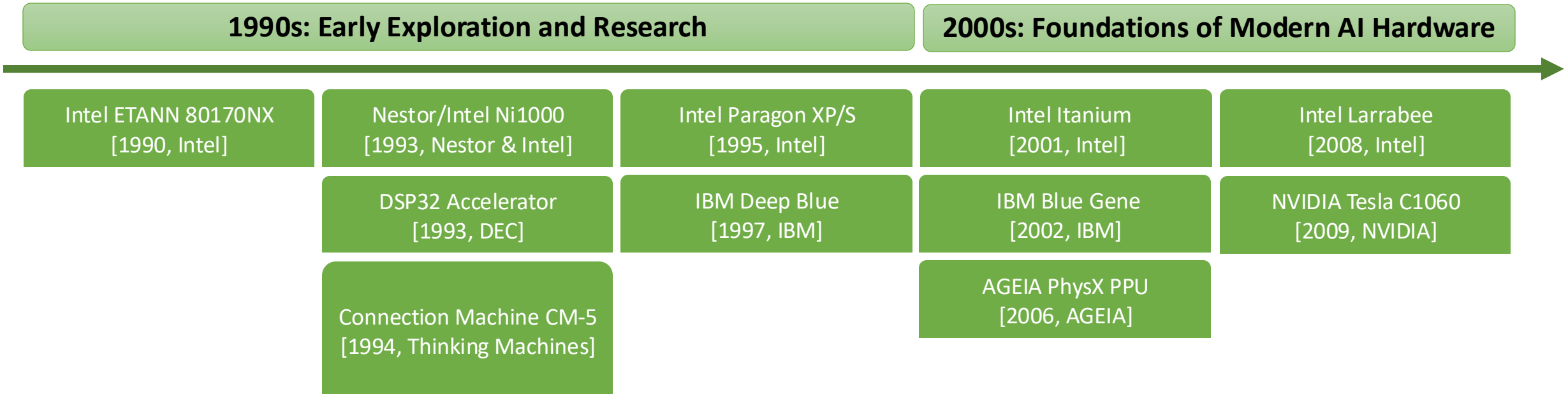
GEMM and GEMV in modern AI applications



Recent research about LLM inference/training, researchers are optimizing the core part: GEMM and GEMV to gain perfromacne, such as:

- [FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics](https://dai.sjtu.edu.cn/my_file/pdf/5da8e046-38d3-44d9-a50f-8ef7d5210bd4.pdf)
- [[2206.09557] LUT-GEMM: Quantized Matrix Multiplication based on LUTs for Efficient Inference in Large-Scale Generative Language Models](https://arxiv.org/abs/2206.09557)
- [[2407.00088] T-MAC: CPU Renaissance via Table Lookup for Low-Bit LLM Deployment on Edge](https://arxiv.org/abs/2407.00088)

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



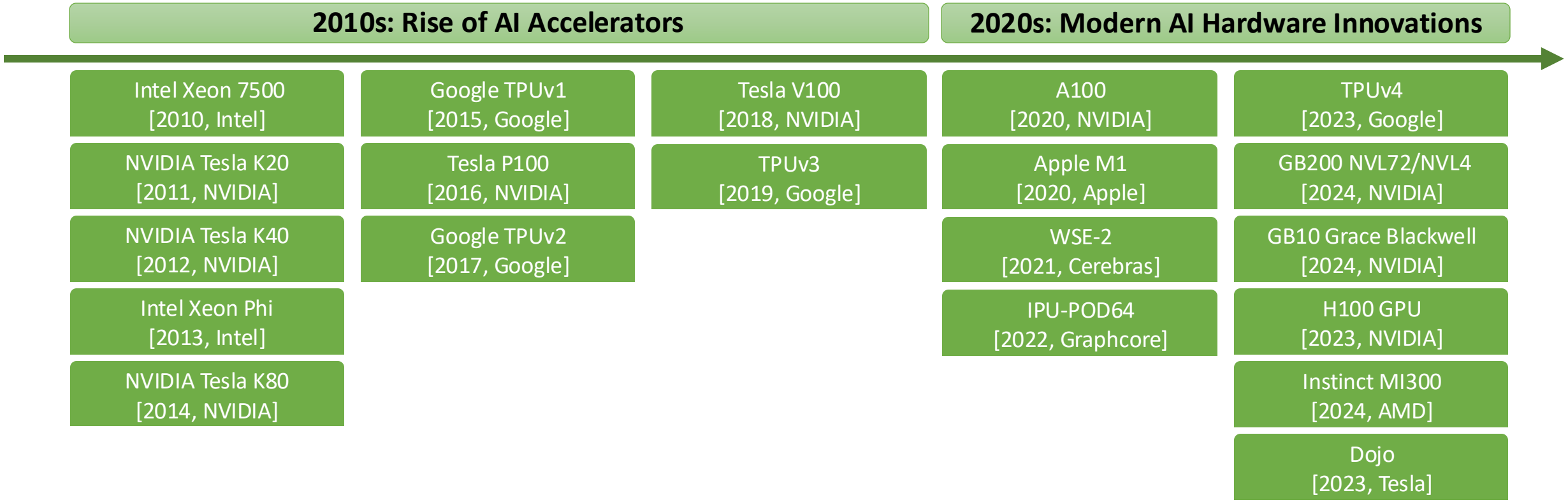
1990s: Early Exploration and Research

- **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 CPUs to specialized AI chips (ASICs, GPUs).
- 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.

In deep learning, this refers to GEMM/GEMV.

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/

Zooming into GPU step by step: Read **whitebook** together

- H100 GPU Whitebook: [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.

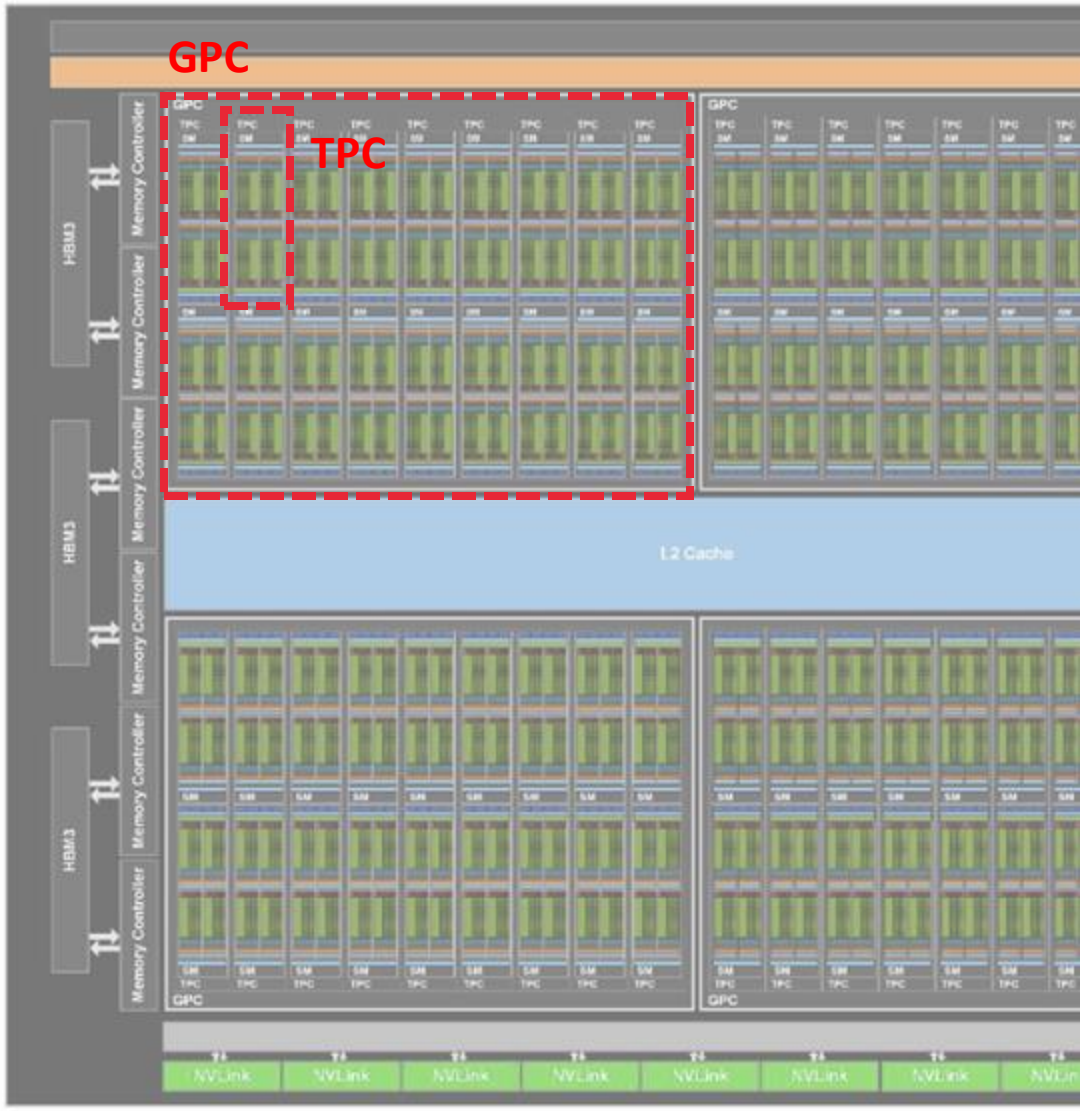
Zooming into GPU step by step: Read whitebook together

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 white book.

Zooming into GPU step by step: Read whitebook together



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.

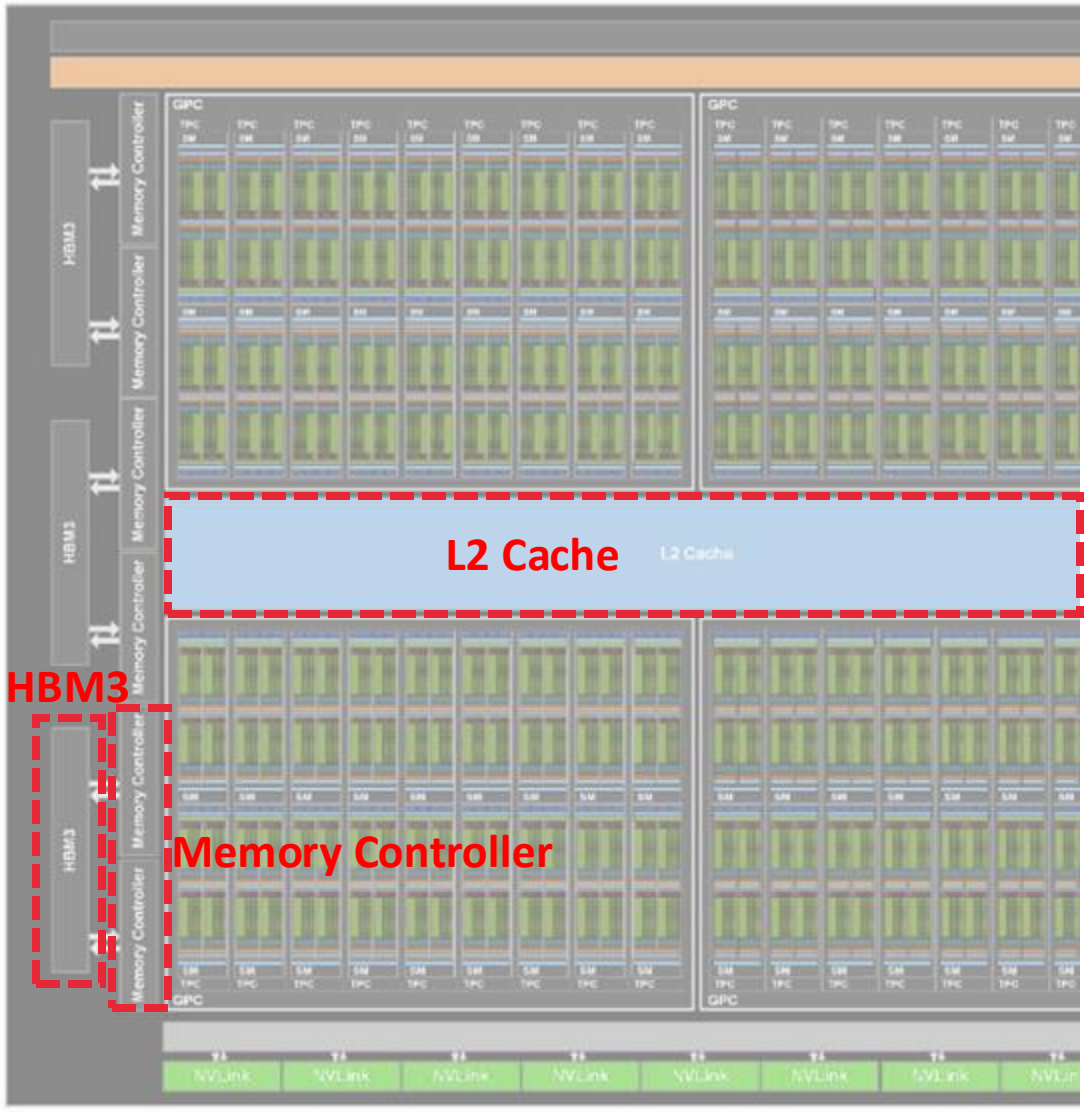
Modular Growth: By organizing the GPU into GPCs and TPCs, manufacturers can scale performance by adding more clusters. For instance, a GPU can have multiple GPCs, each containing several TPCs, allowing for increased parallelism without redesigning the entire architecture.

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

Zooming into GPU step by step: Read whitebook together



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.

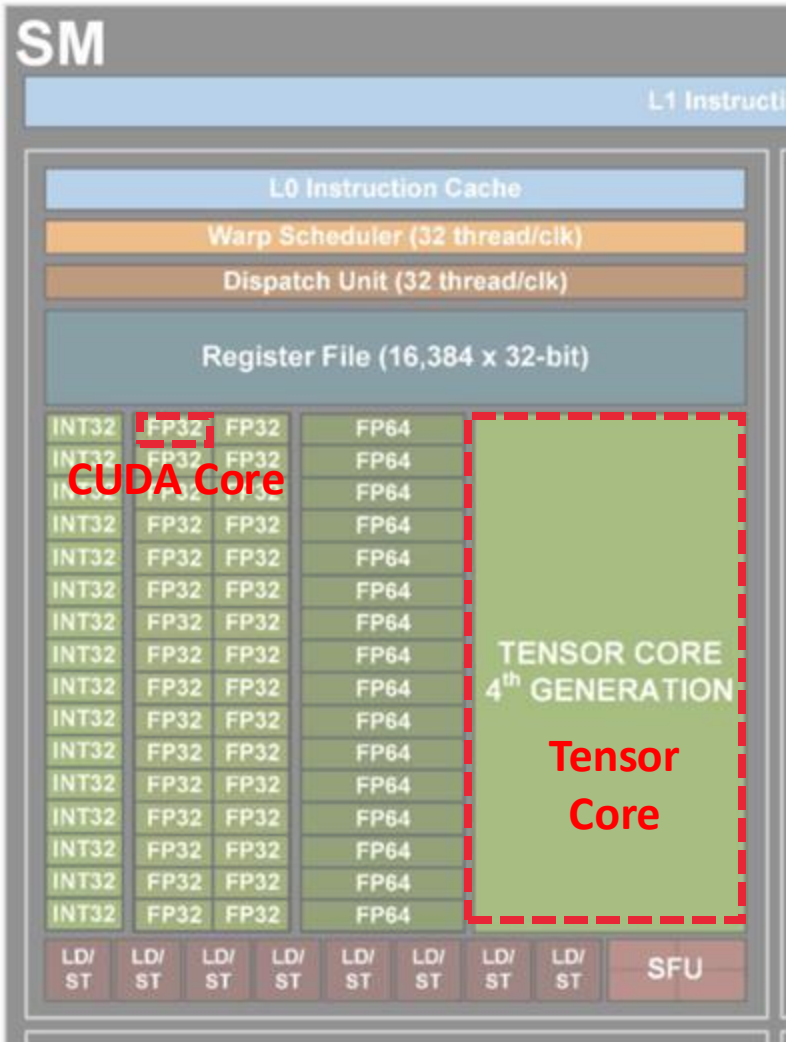
SM (Streaming Multiprocessor)

In every SM

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

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

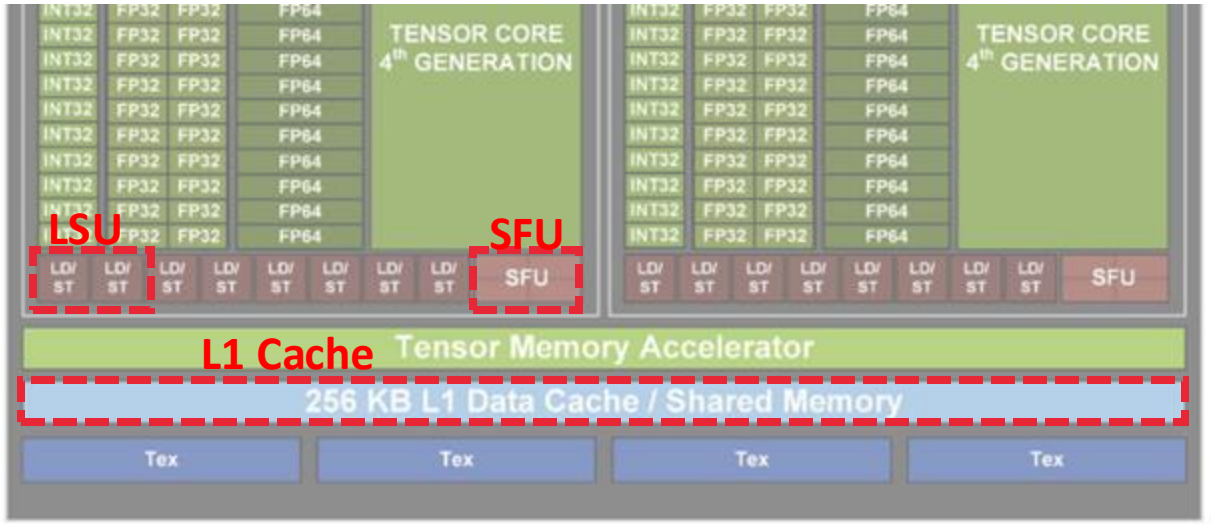
Tensor Cores are GPU cores that work on entire matrices with each instruction.



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.

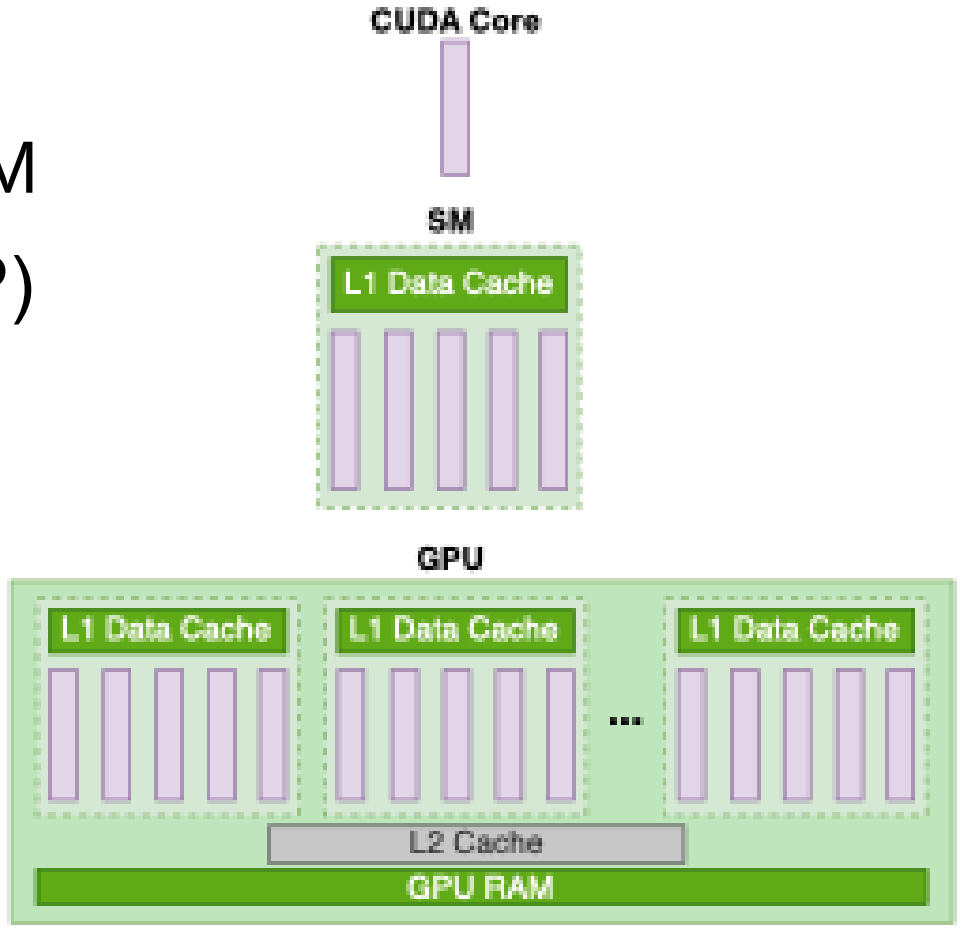


The L1 data cache is the private memory of the Streaming Multiprocessor (SM).

Let's focus on a simplified GPU hardware model

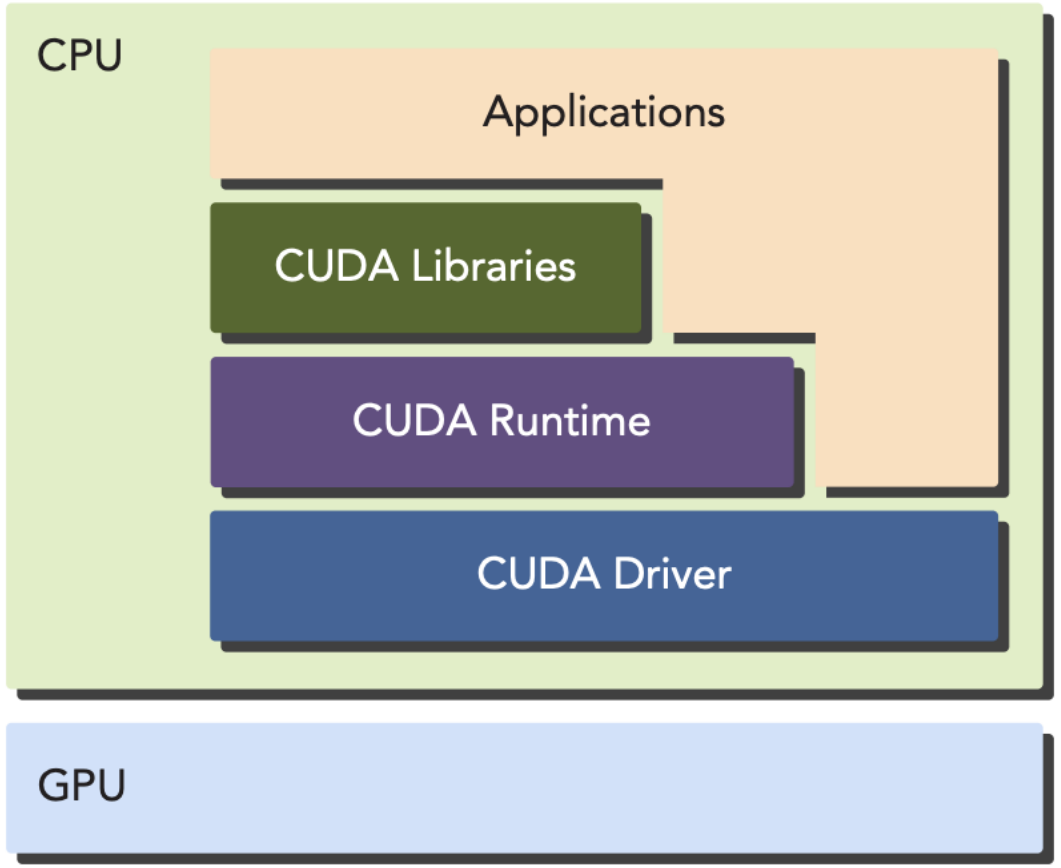
- GPU ≈ multiple Streaming Multiprocessors (SM) + L2 cache + RAM
- SM ≈ multiple Streaming Processor (SP)

  + L1 cache
- SP ≈ several cuda cores



Notice: in Fermi or some old architectures, SP = CUDA core;
in modern architectures like Kepler, Maxwell, or Pascal, SP = server CUDA cores + tensor cores.

What is 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

A blasting start to use CUDA.

```cpp
__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.

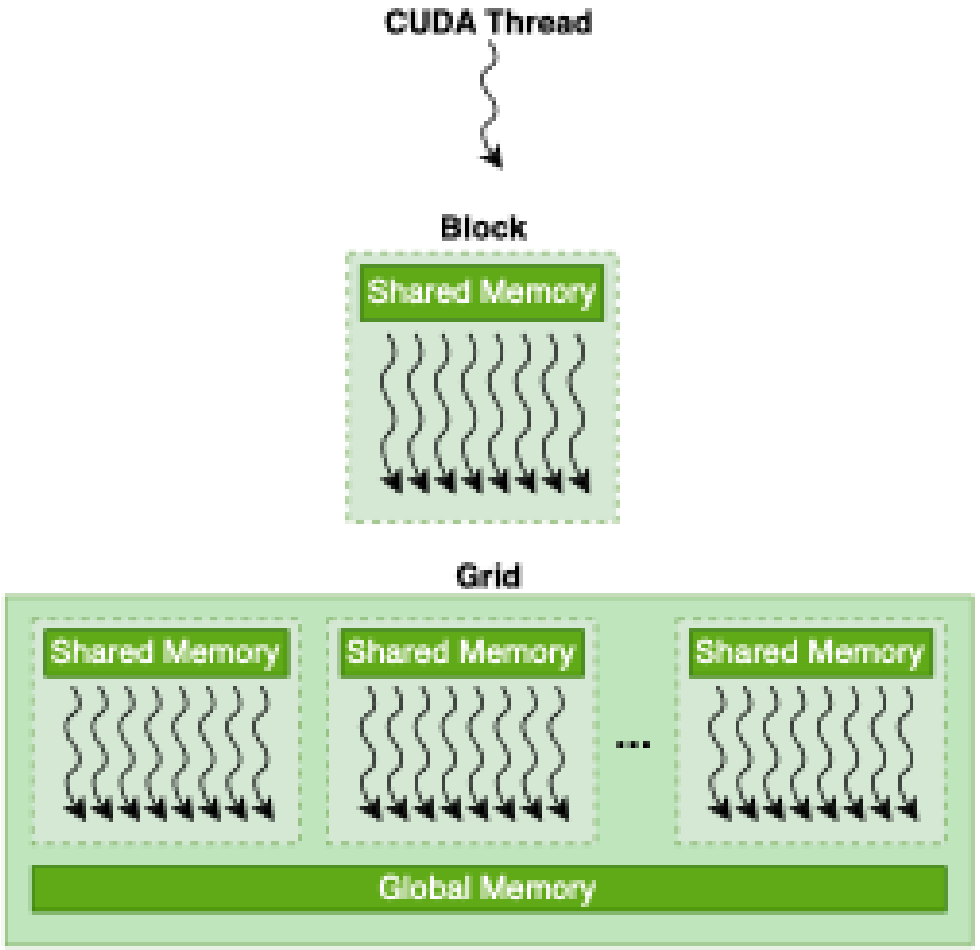
Kernel Code and Thread

CUDA Kernel Code:

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

CUDA Thread:

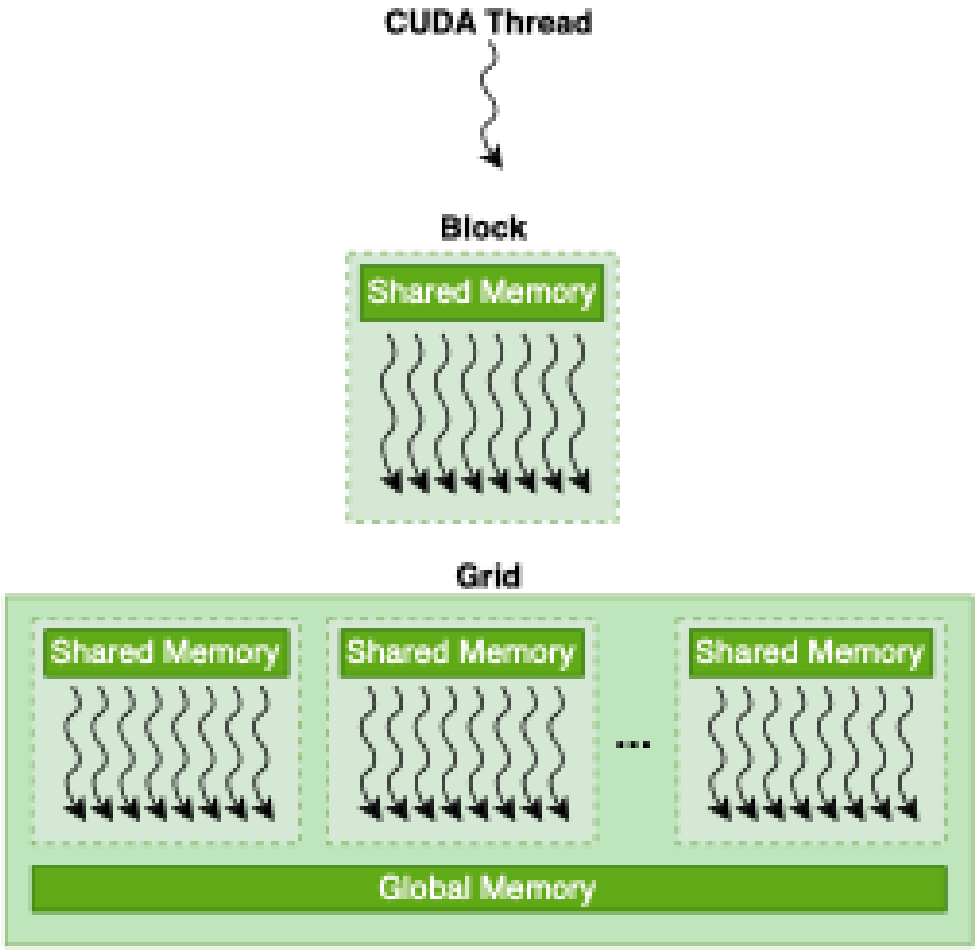
- A *thread of execution* (or "thread" for short) is the lowest unit of programming for GPUs.



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.

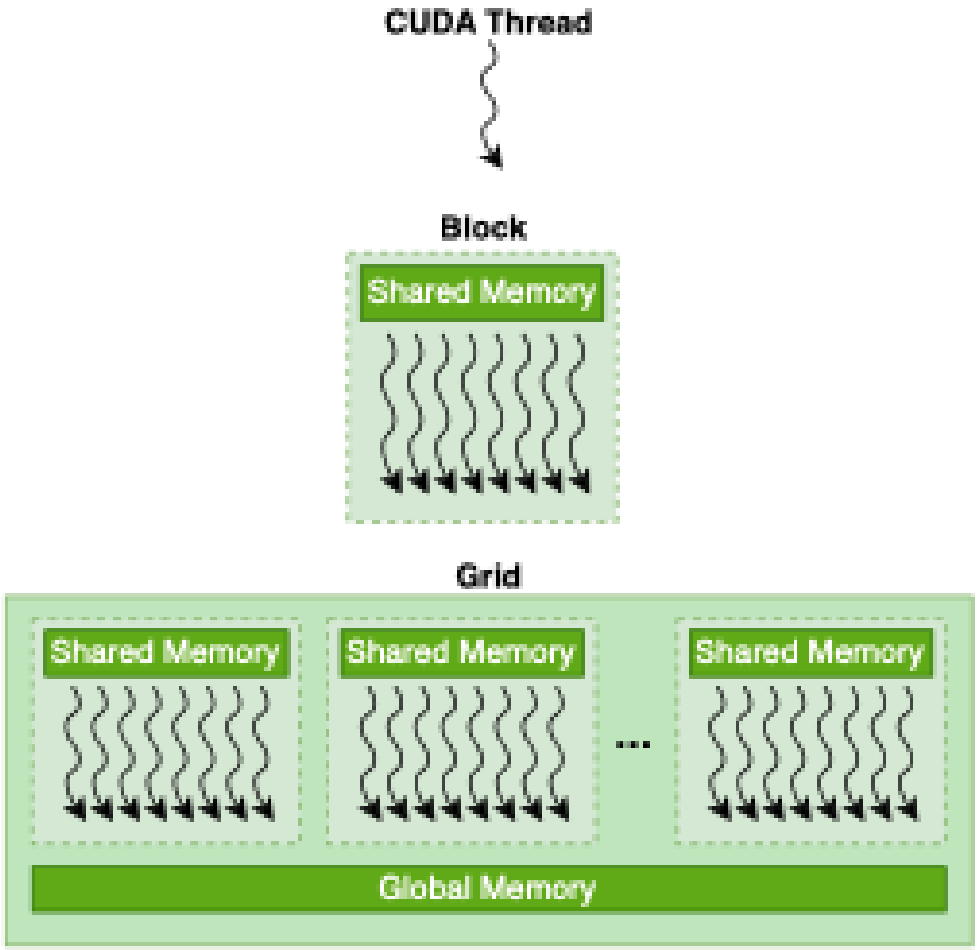
Blocks must execute independently, so that any execution order for blocks is valid, from fully serial in any order to all interleavings.

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.

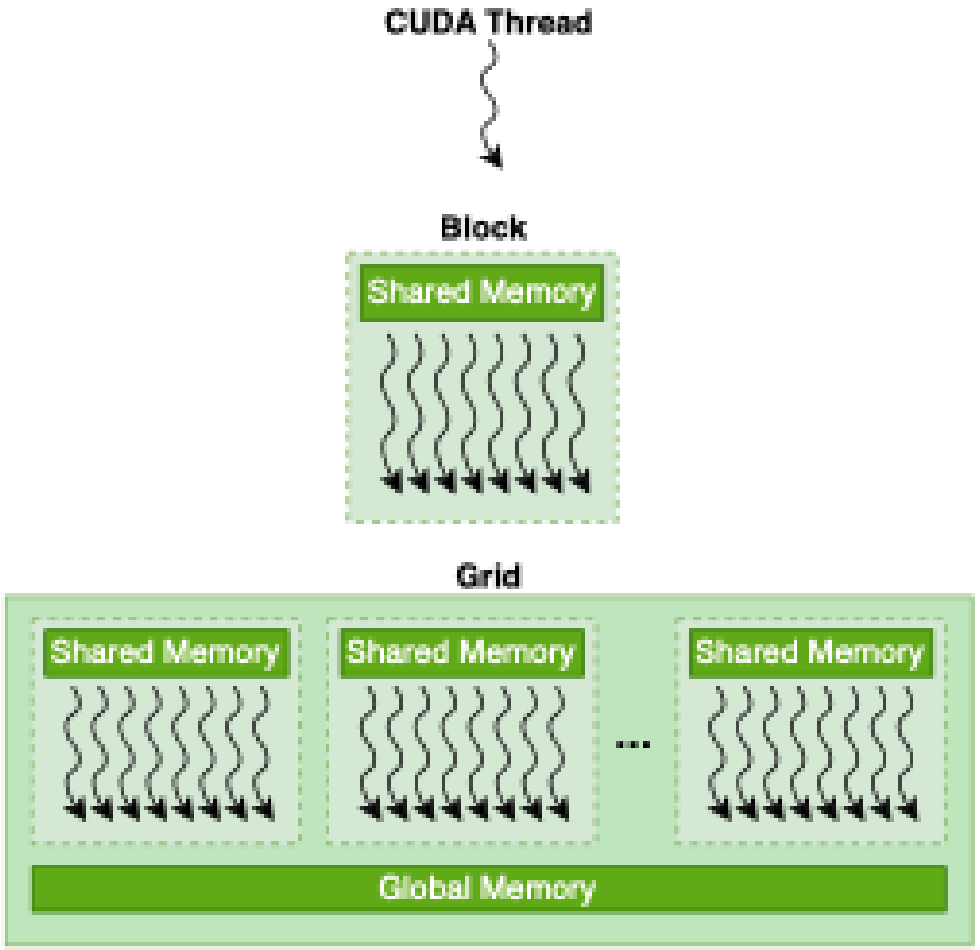
The matching level of the memory hierarchy is the global memory .

Block and Grid are both logical concept!

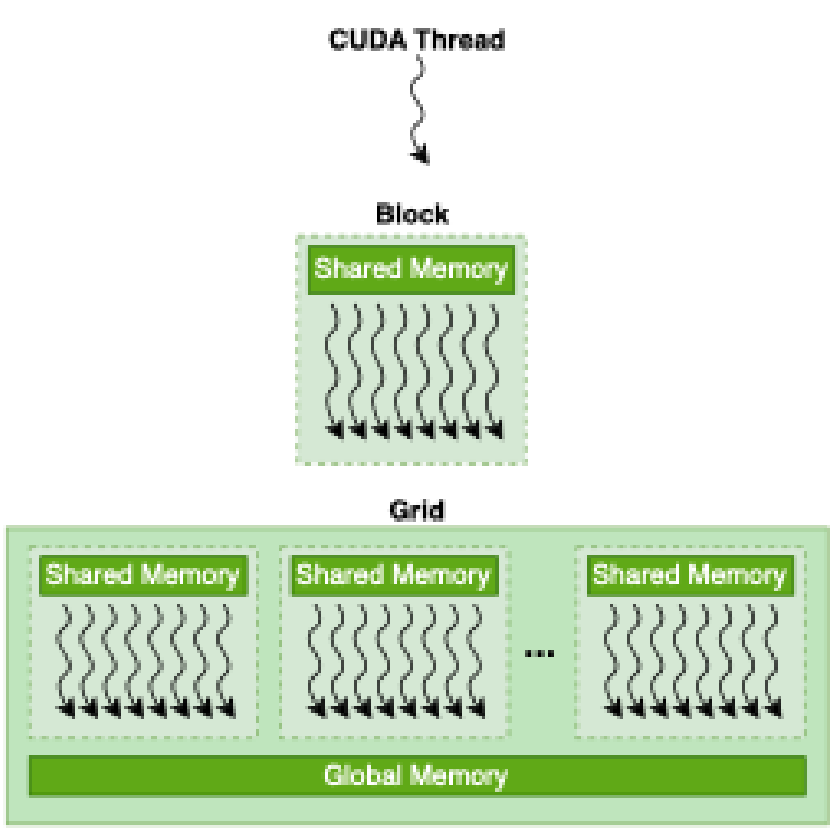


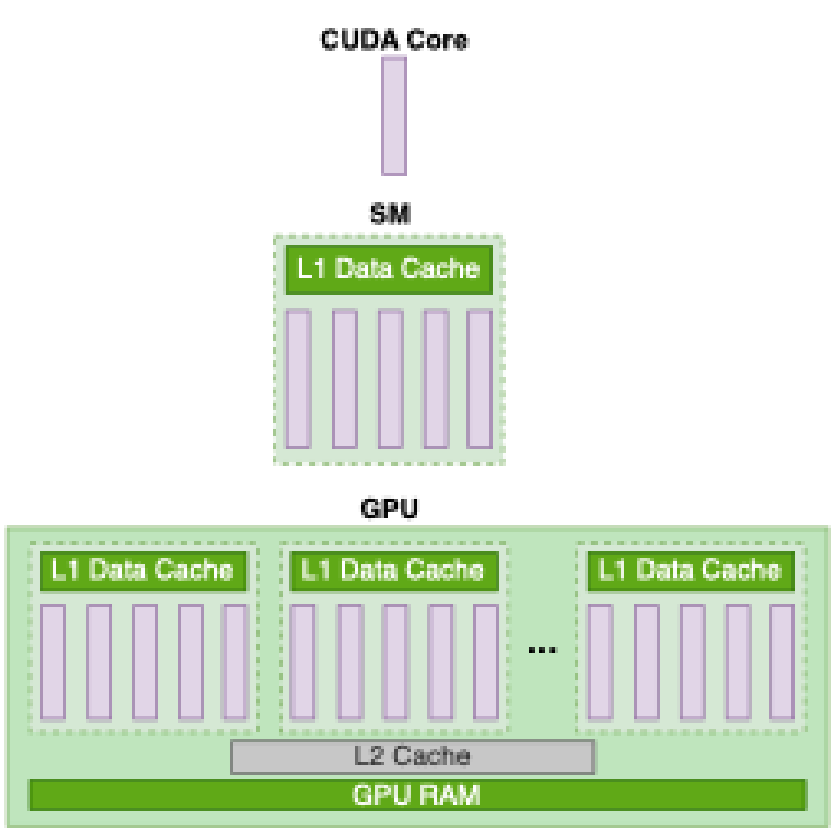
Logic Model on GPU = Block-Thread Parallel

- Block and Tread 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



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



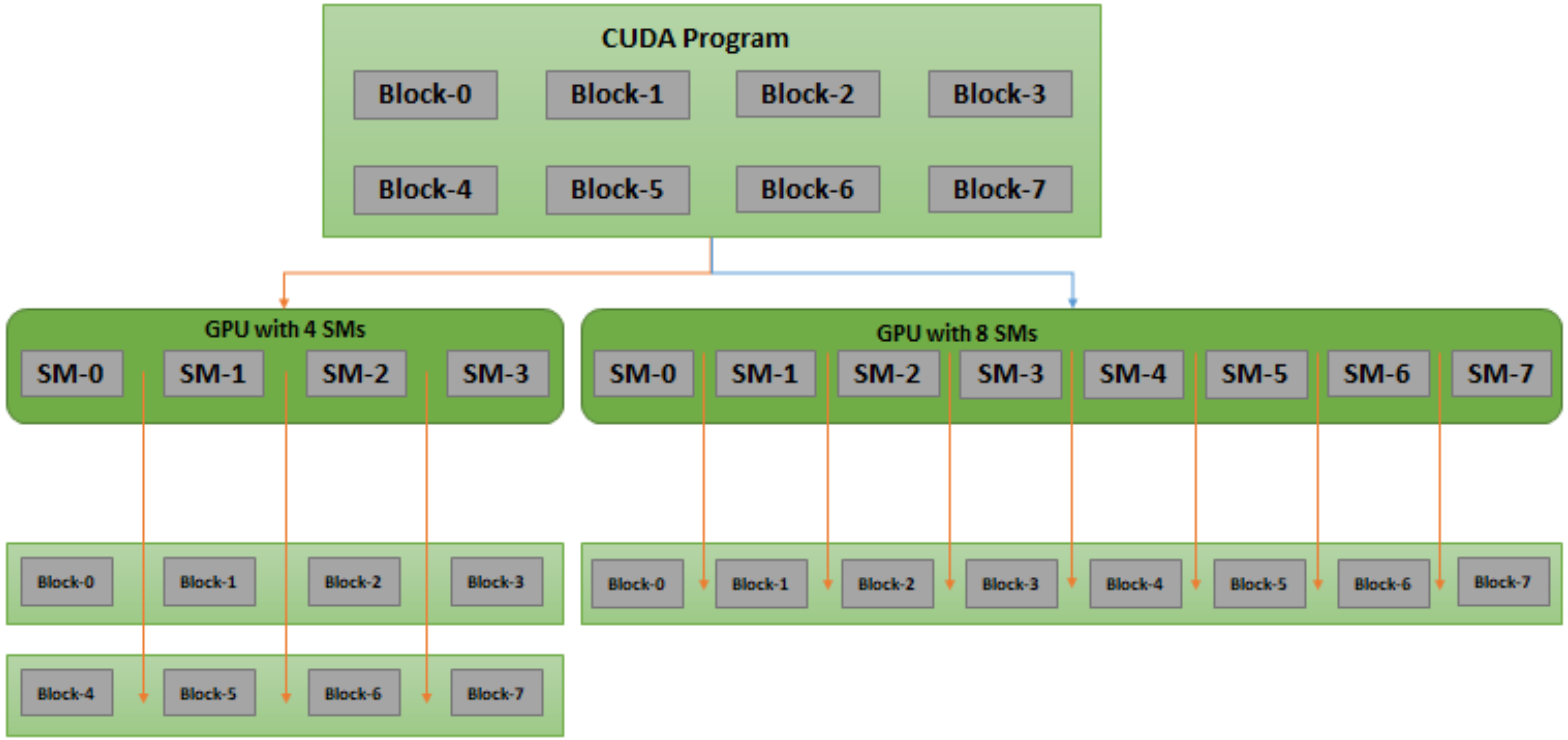


- SP(Cuda Cores) is used to handle threads.
- SM is used to handle blocks.

- 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.

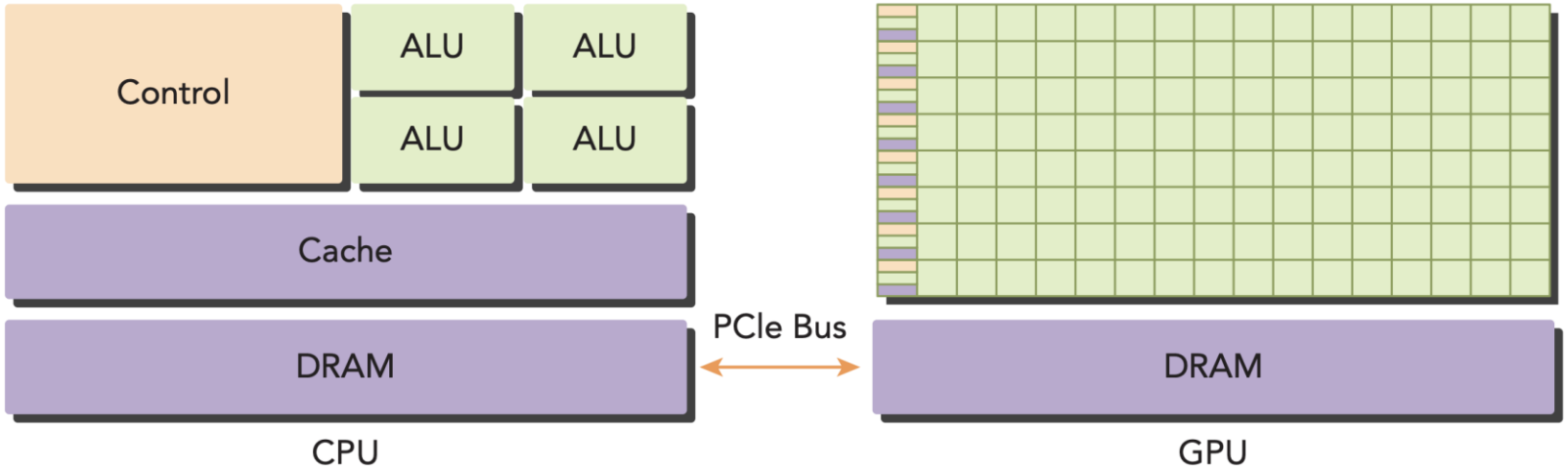
Block and Grid



- 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!)**

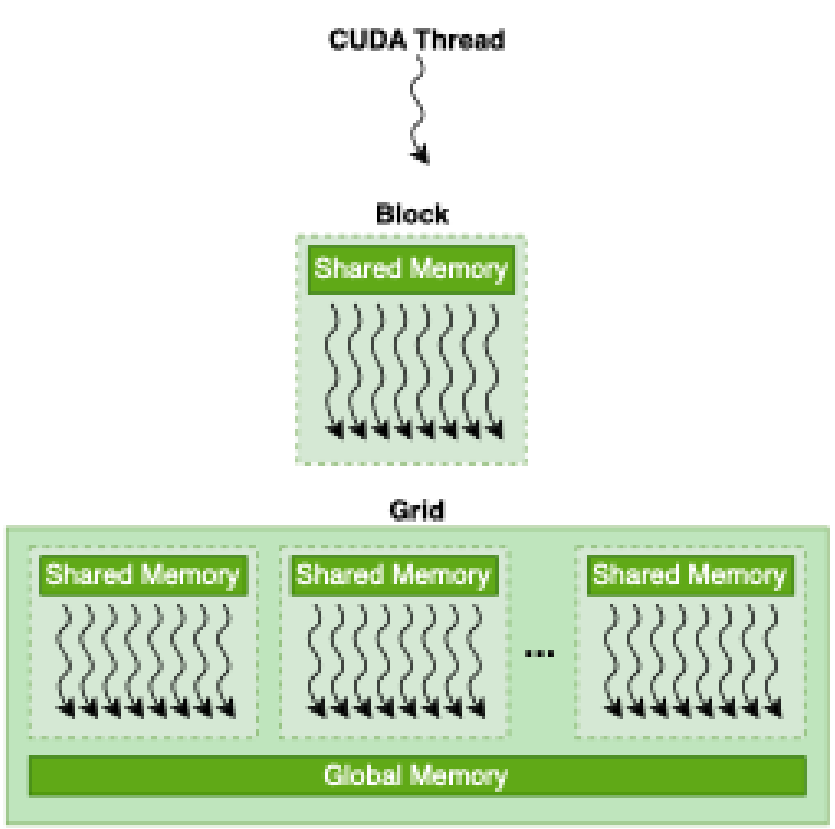
Why parallel on GPU not CPU?

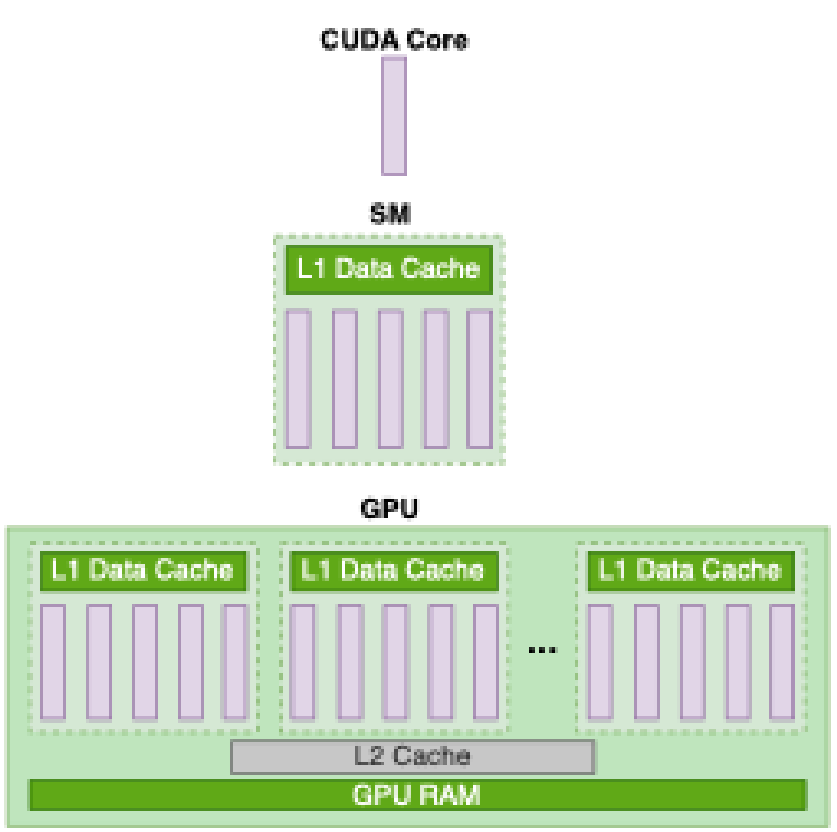


- GPU SMs can execute more threads in parallel.
- For comparison, an AMD EPYC 9965 CPU draws at most 500 W and has 192 cores, each of which can execute instructions for at most two threads at a time, for a total of **384** threads in parallel, running at about **1.25 W per thread**.
- An H100 SXM GPU 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 **5 cW** apiece. Note that this is truly parallel: each of the 16,000 threads can make progress with each clock cycle.

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

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.

A blasting start to use CUDA.

```cpp
__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.


THE UNIVERSITY of EDINBURGH
informatics
Learn GPU Programming by Examples

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

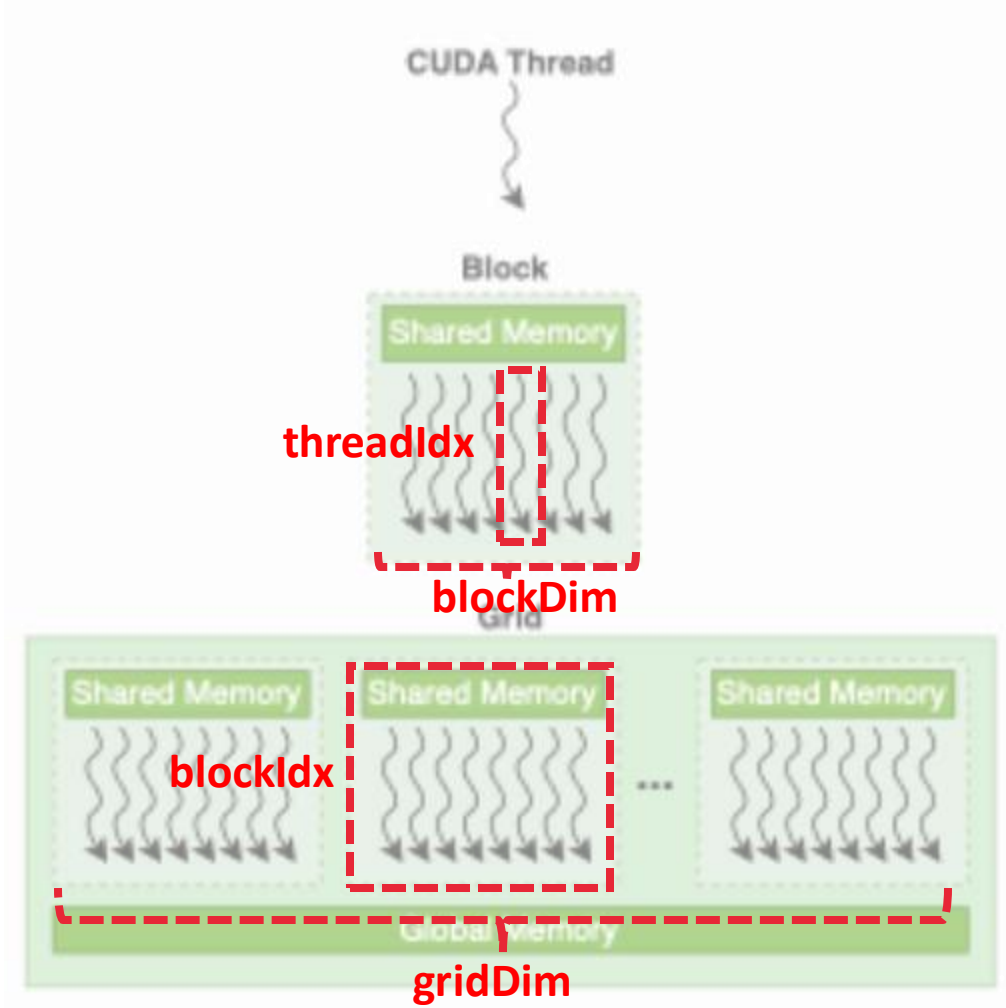
```
__global__ void vector_add(const float* a, const float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

// 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

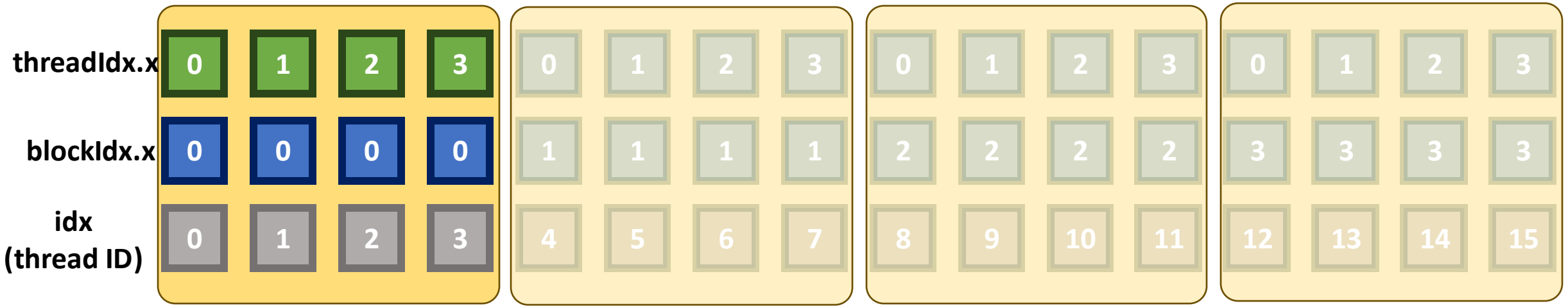


- Dim and Idx can be 1, 2, or 3 dimensions, but we typically use 1 dimension for 1D array and 2 dimensions for 2D-array(matrix).

In 1D array:

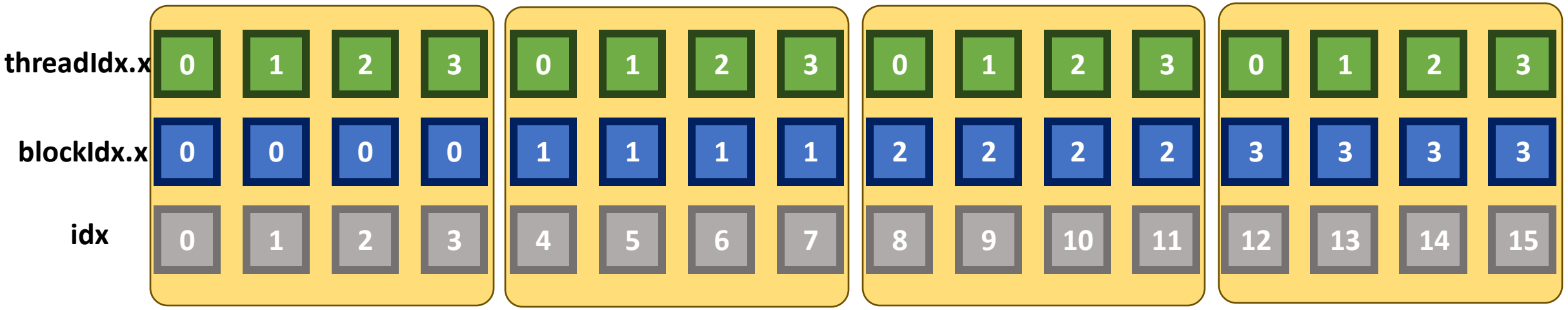
idx = blockId.x * blockDim.x + threadIdx.x

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



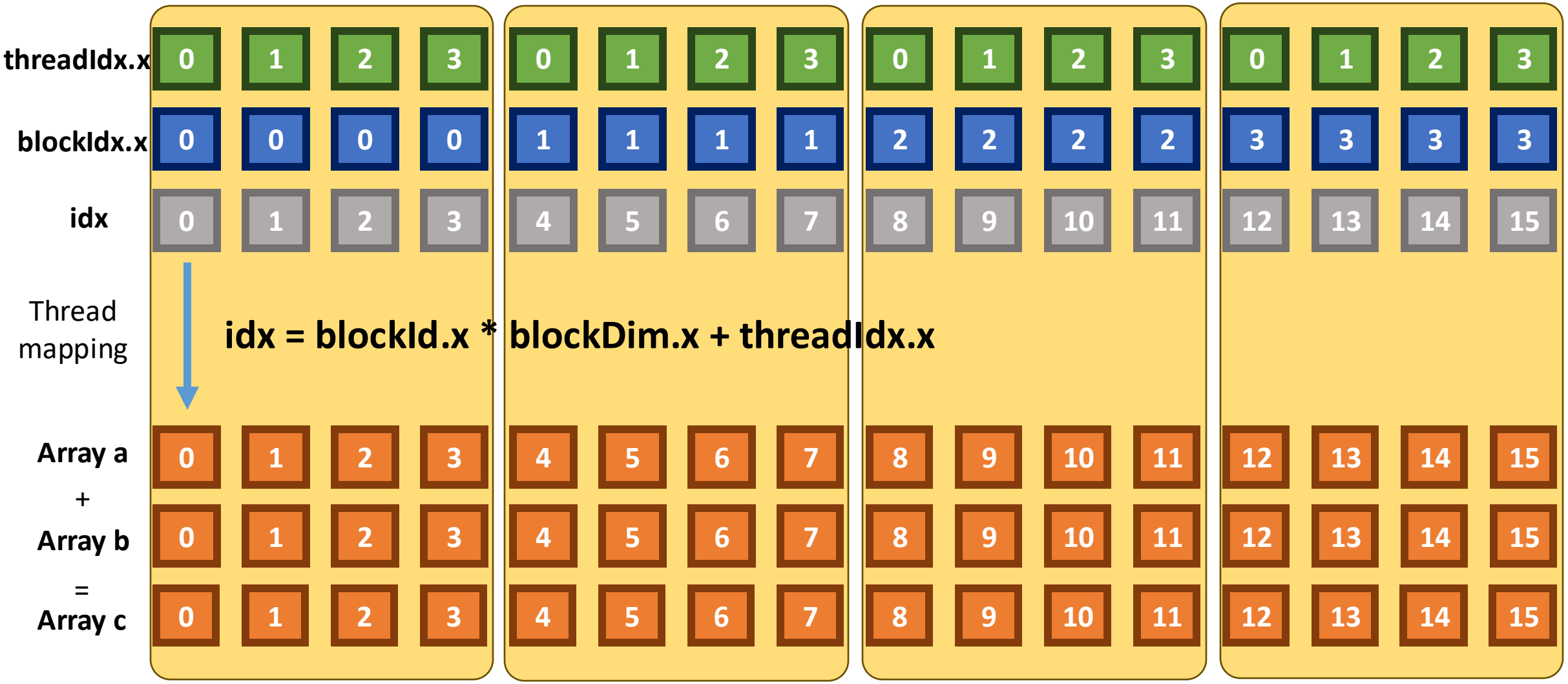
idx = blockId.x * blockDim.x + threadIdx.x

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 Wrap Architecture**
  - 256 divides evenly by 32 (#threads in a wrap), 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.

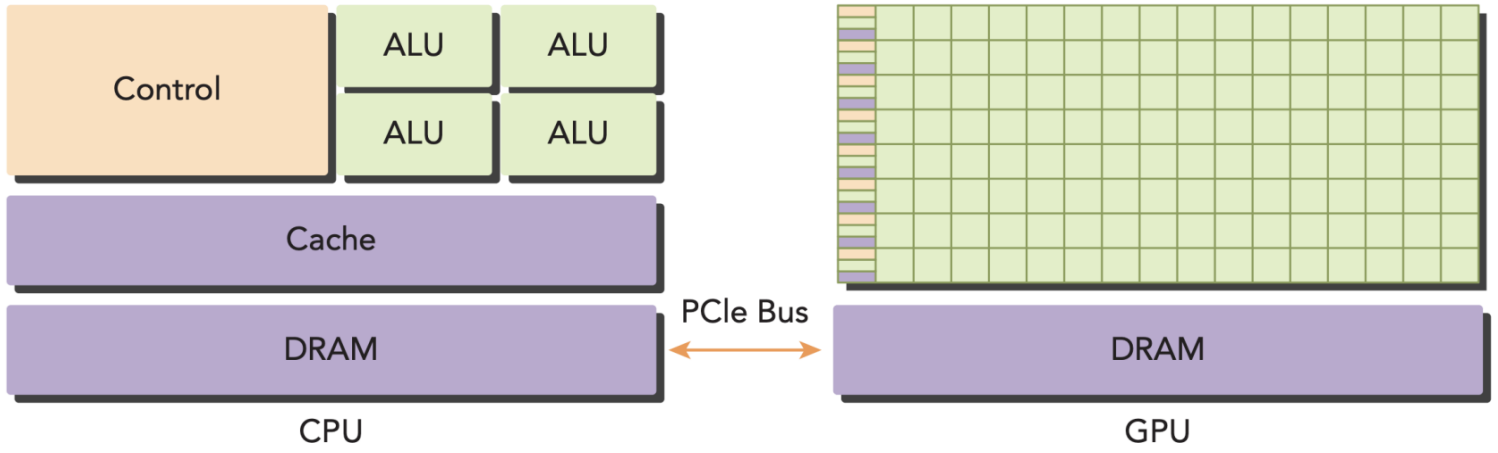
- Equivalent to $\lceil n/blockSize \rceil$

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

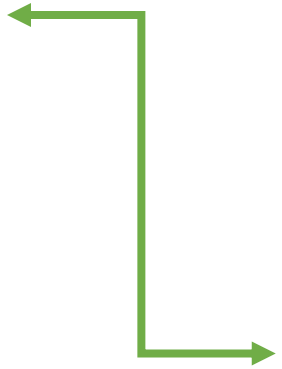
- Device can only visit device memory by address
- Host can only visit host memory by address

- 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

```cuda
__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

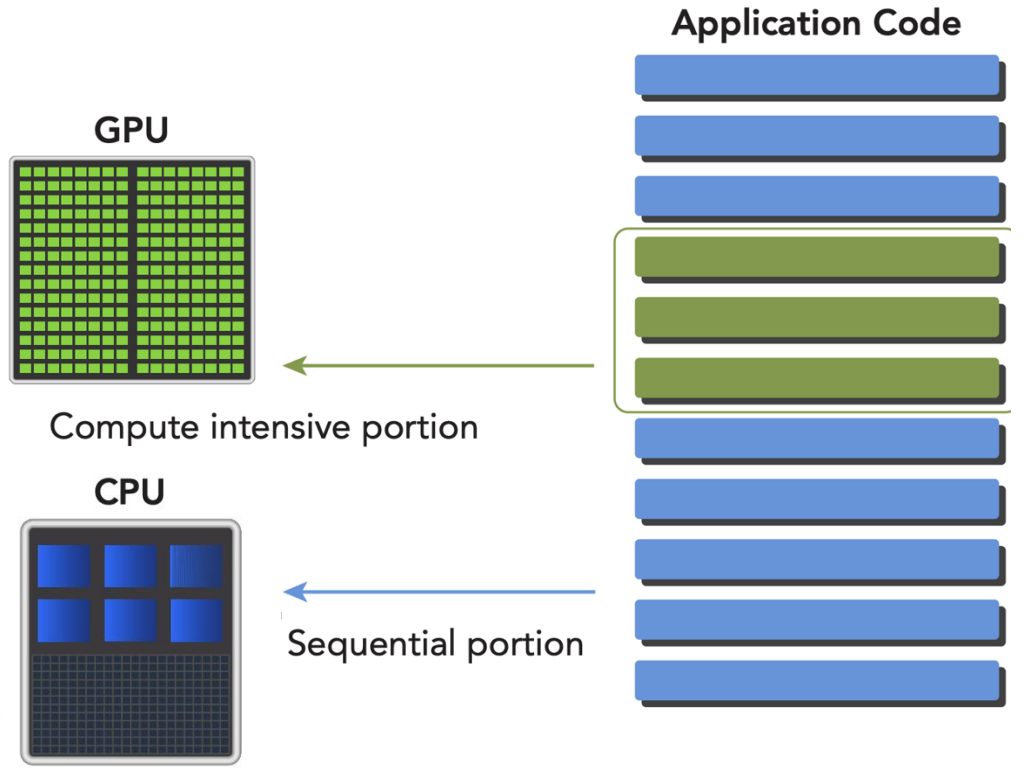
```cuda
// Host memory
float *h_a, *h_b, *h_c;
h_a = (float*)malloc(size);
h_b = (float*)malloc(size);
h_c = (float*)malloc(size);

// 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:

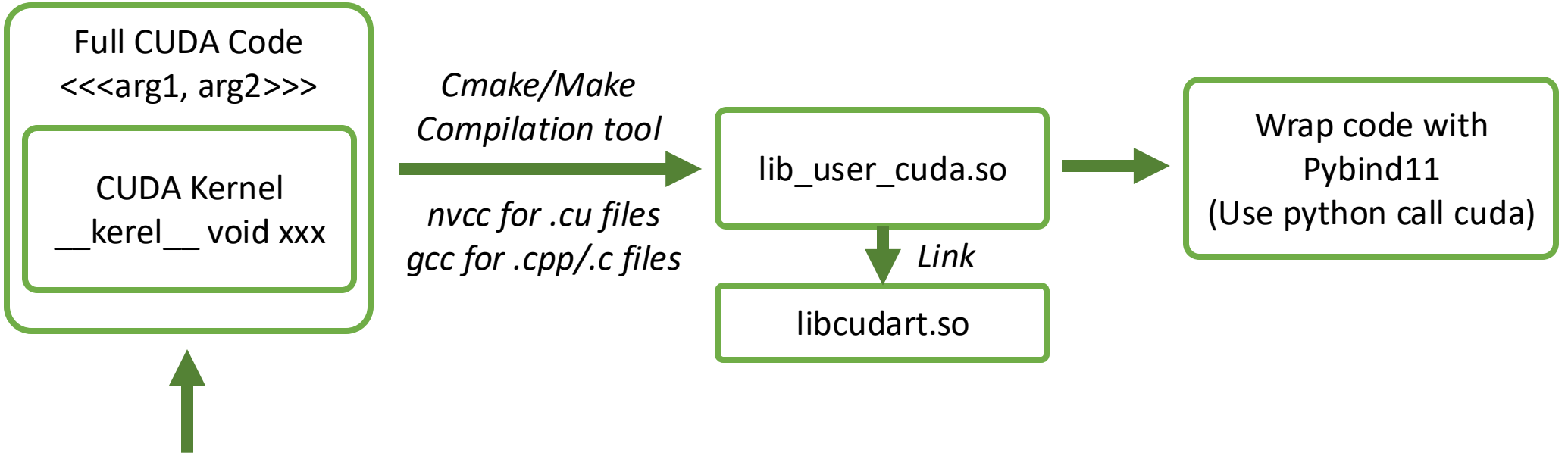
- Identify the sections that can be parallelized
- Copy the relevant data to the GPU
- Run the CUDA kernel
- Copy the results back to the CPU

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



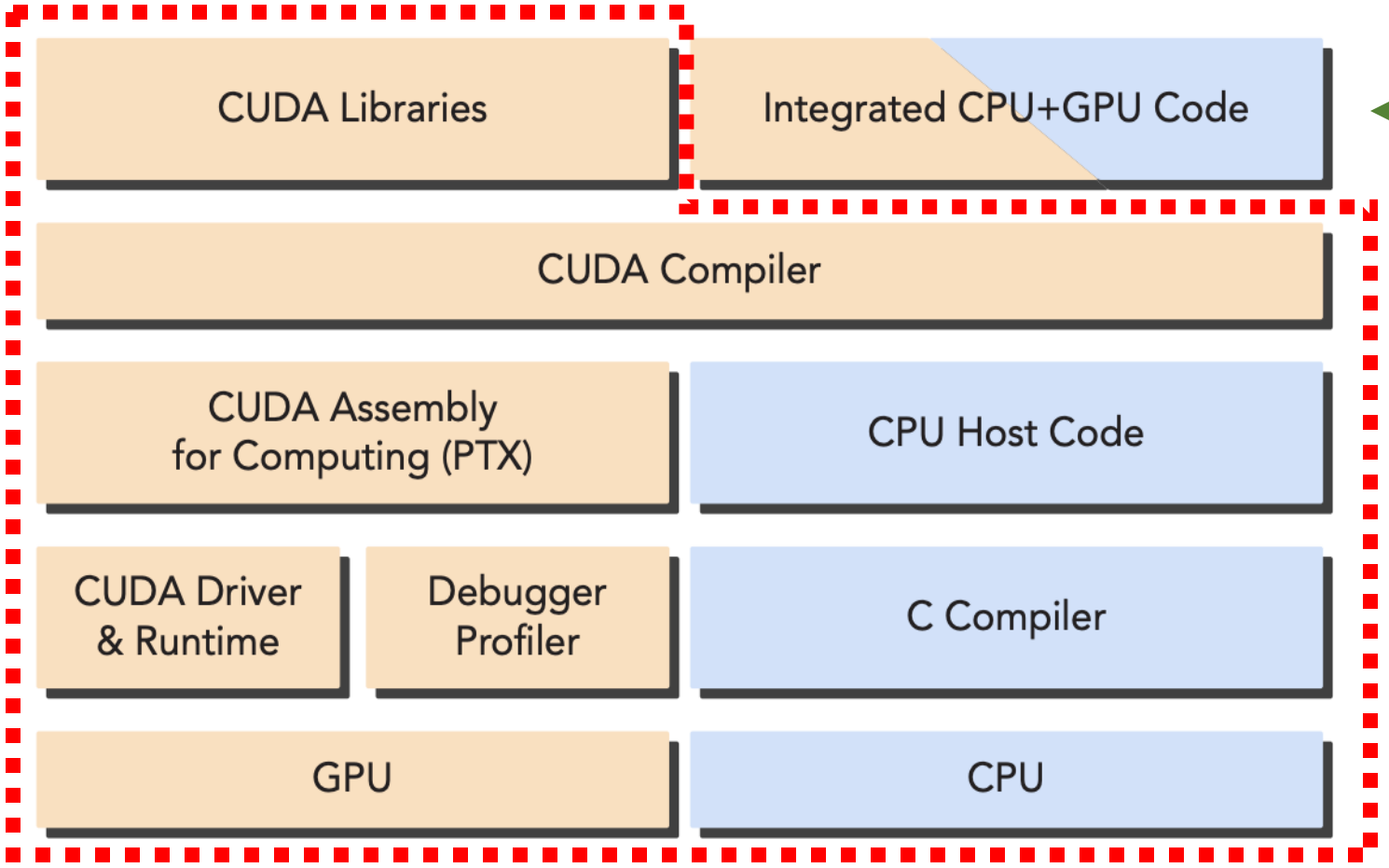
A simple workflow for compiling cuda and bind to python API



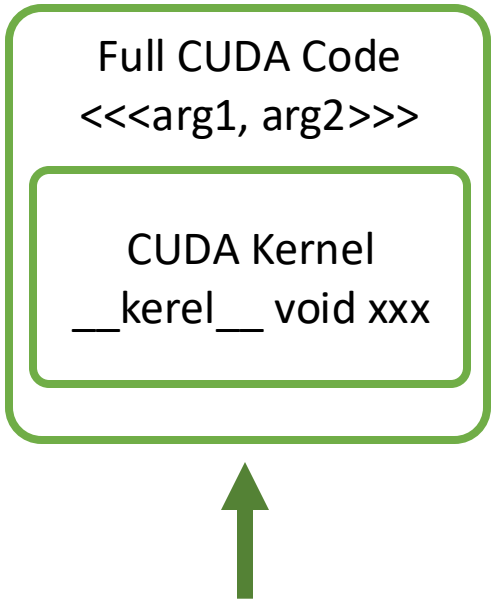
What we have learned just now.

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.



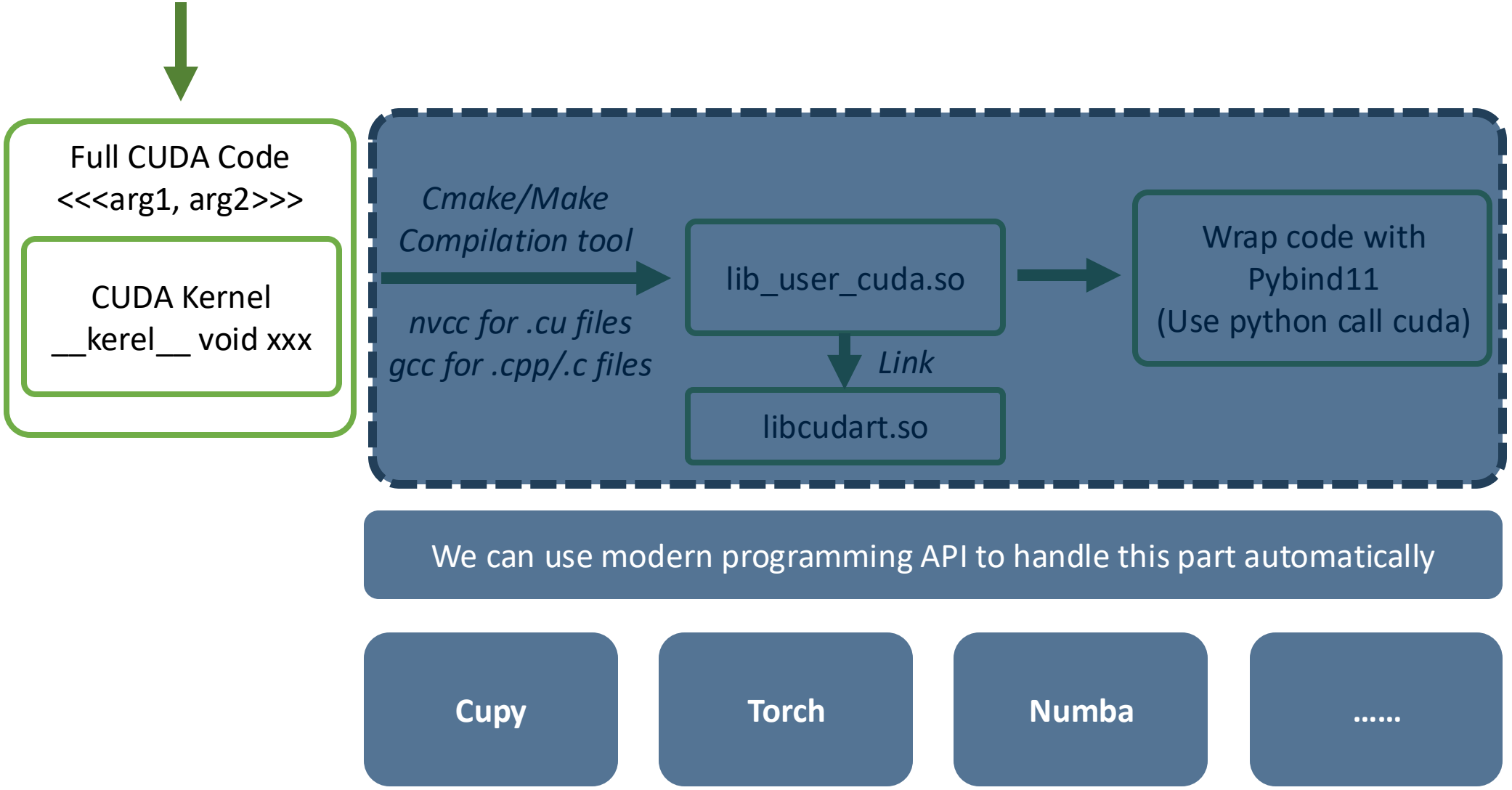
What we still don't know



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

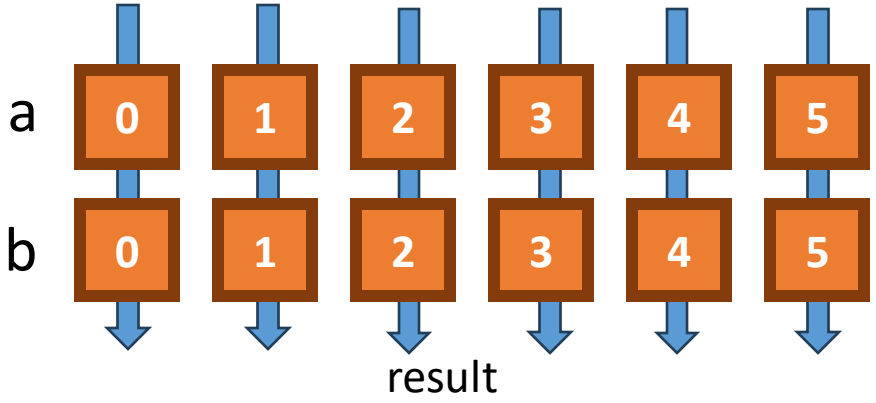
```python
import torch

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

```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()
```

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

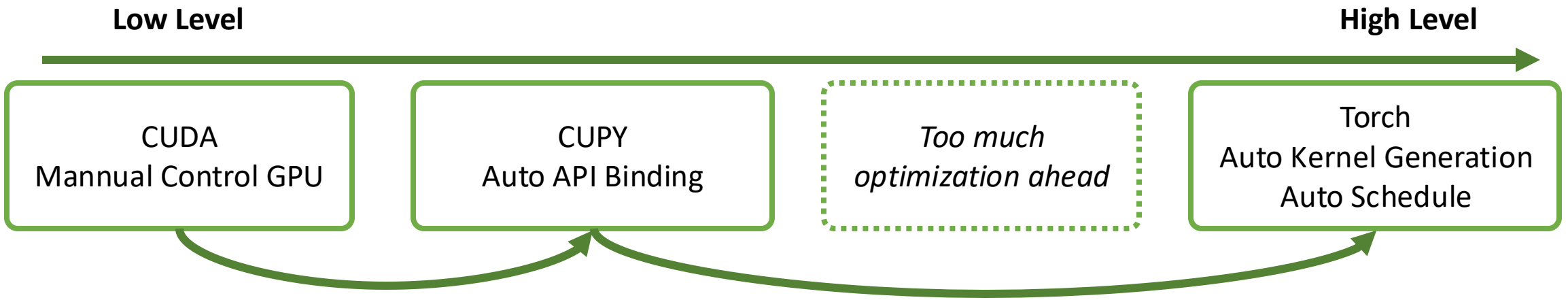
- When use other types of kernels, they still need to set proper grid size and block size

- 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 wrap

- 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

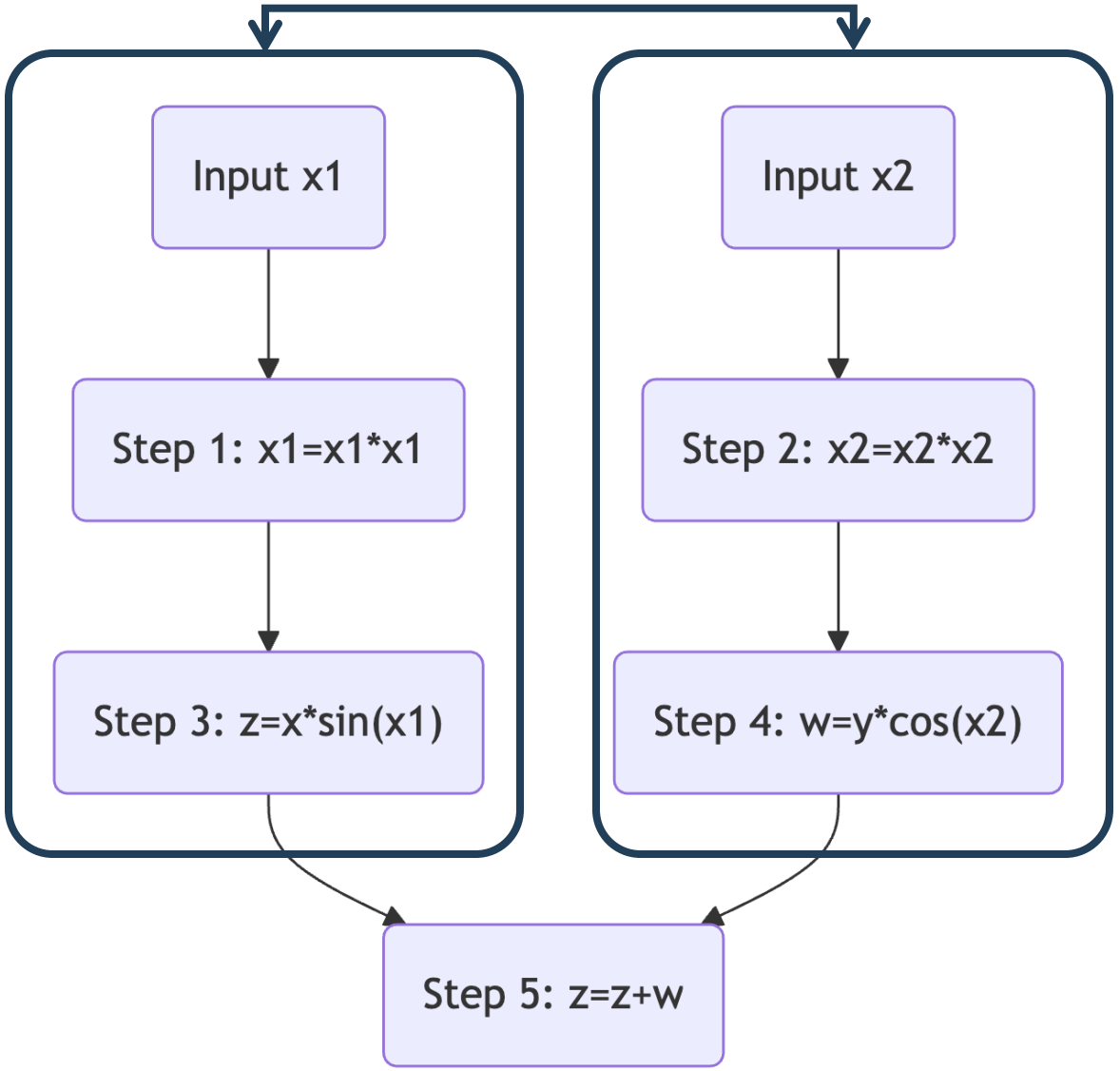
    (usually refer to overlap data transfer and computation)

[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

Create Streams

Stream 1

Stream 2

```python
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
```

Torch Version Stream

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

Torch Stream

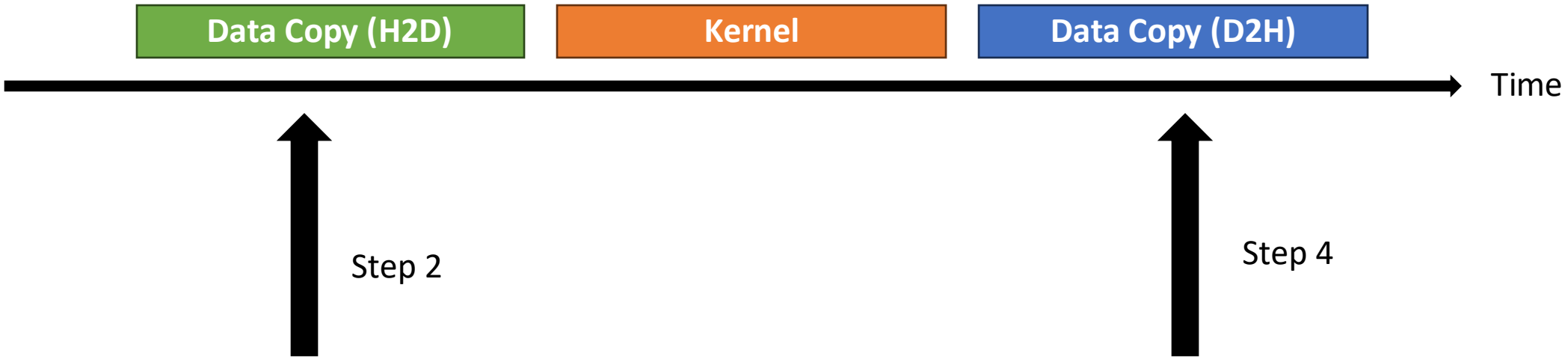
Create Streams

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 Senario when we need Cuda Stream

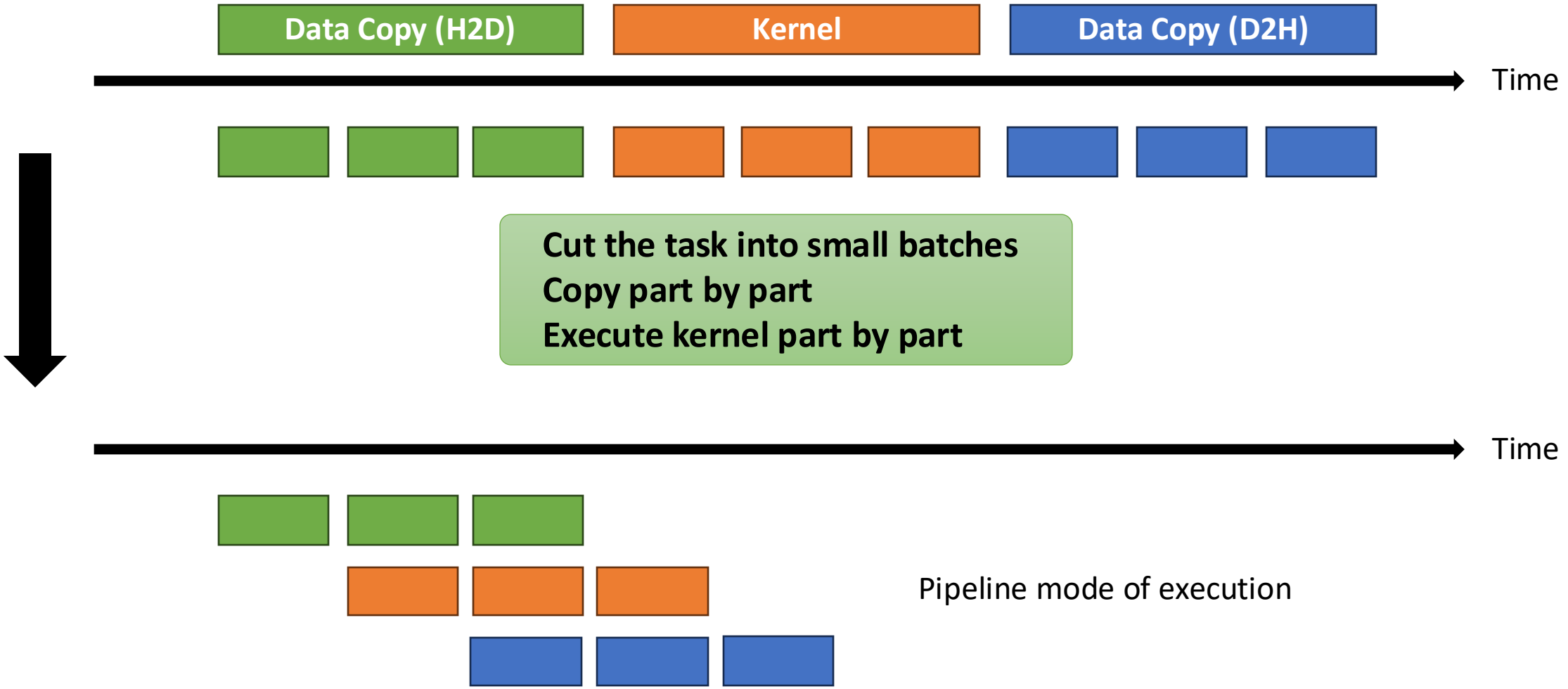


Programming pattern on CUDA:

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 Senario when we need Cuda Stream



Cupy Stream Example

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

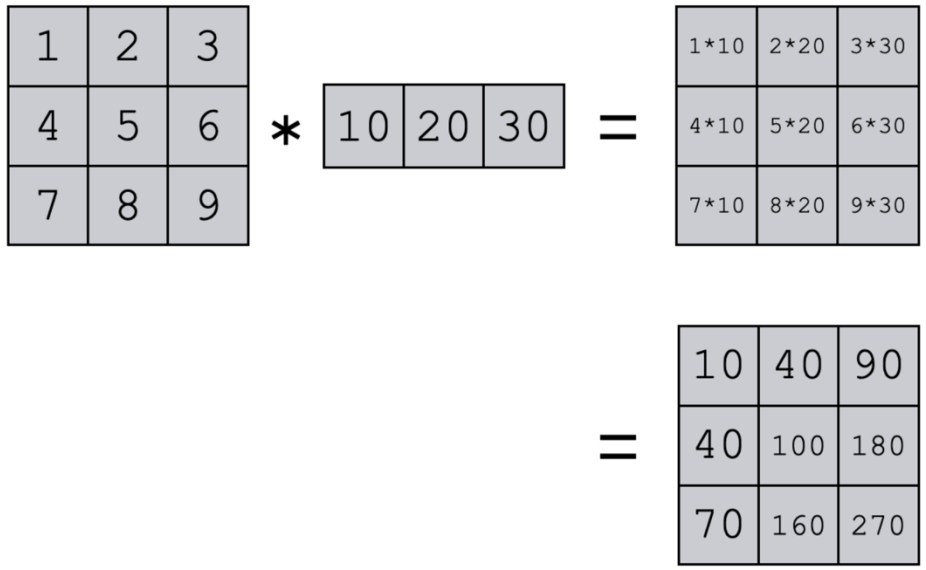
- Stream 1: copy x and y from host to device
- Stream 2: compute z
- Stream 3: copy z from device to host

- 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: Hirachy 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: Hirachy 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: Hirachy 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: Hirachy 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?

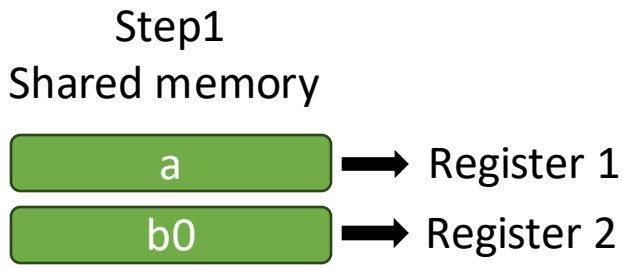
- Dummy strategy on Cupy

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]
```



Cost = 2 * load time

Why we need to optimize shared memory?

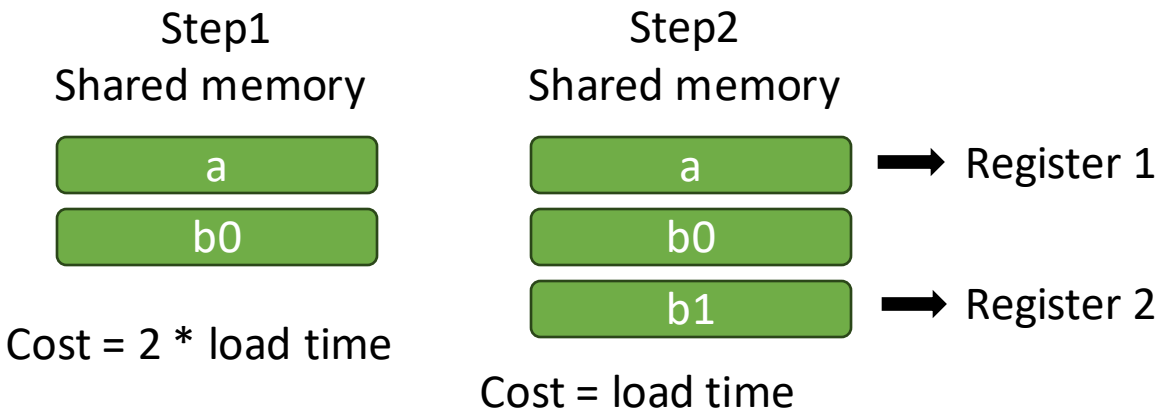
- Dummy strategy on Cupy

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?

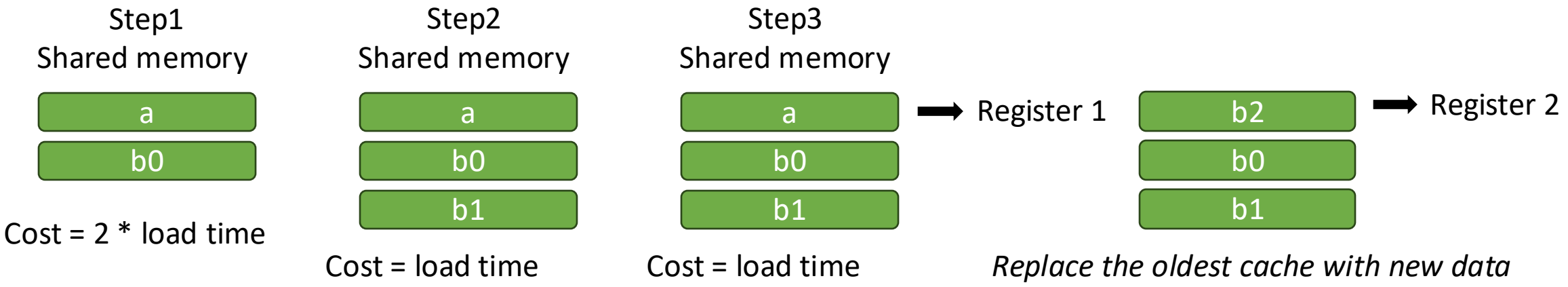
- Dummy strategy on Cupy

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?

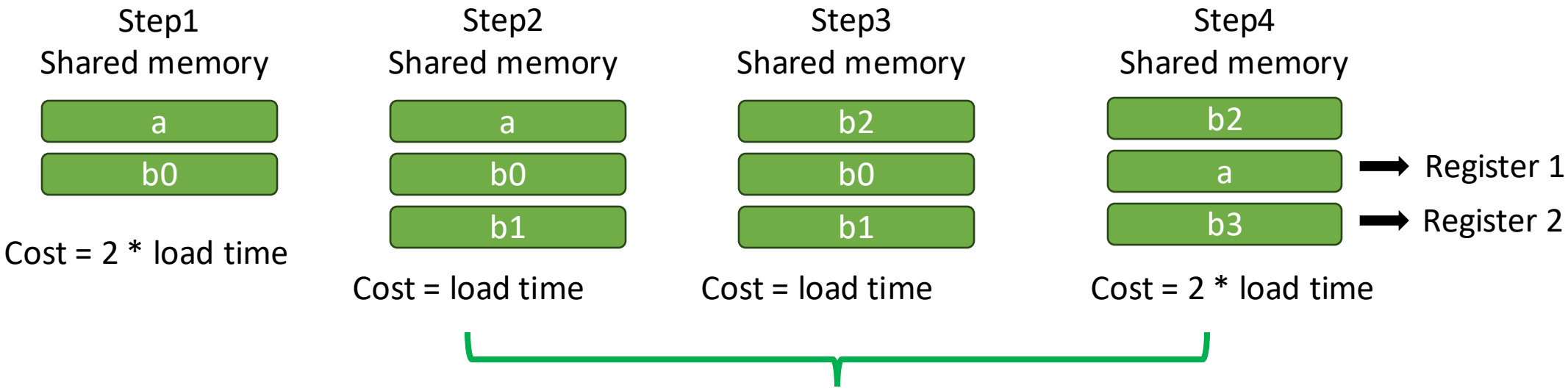
- Dummy strategy on Cupy

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?

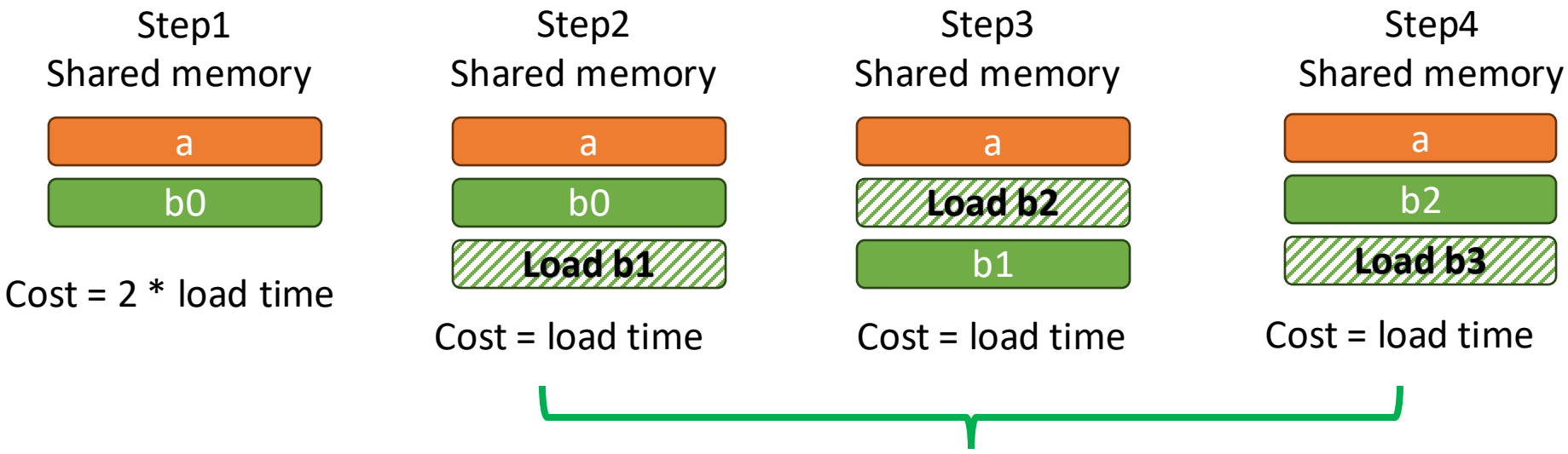
- Dummy strategy on Cupy

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.

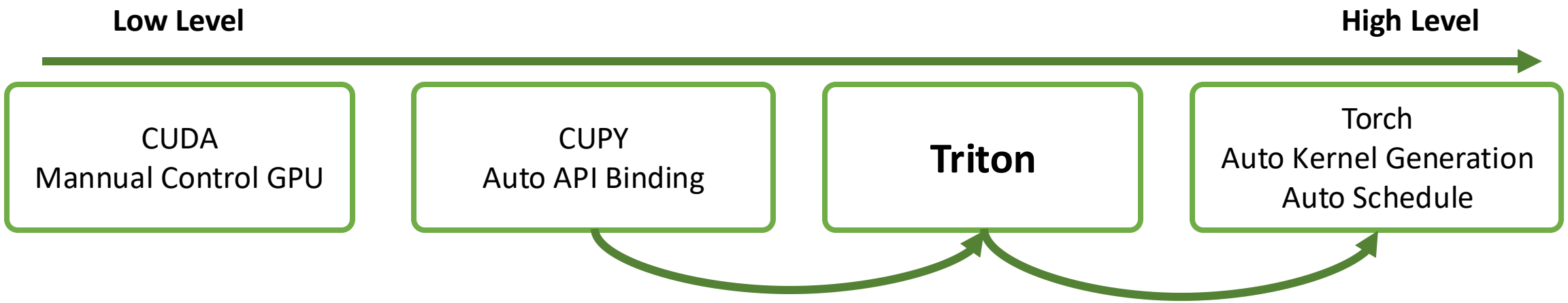


Cupy API for shared memory

- 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 wrap 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

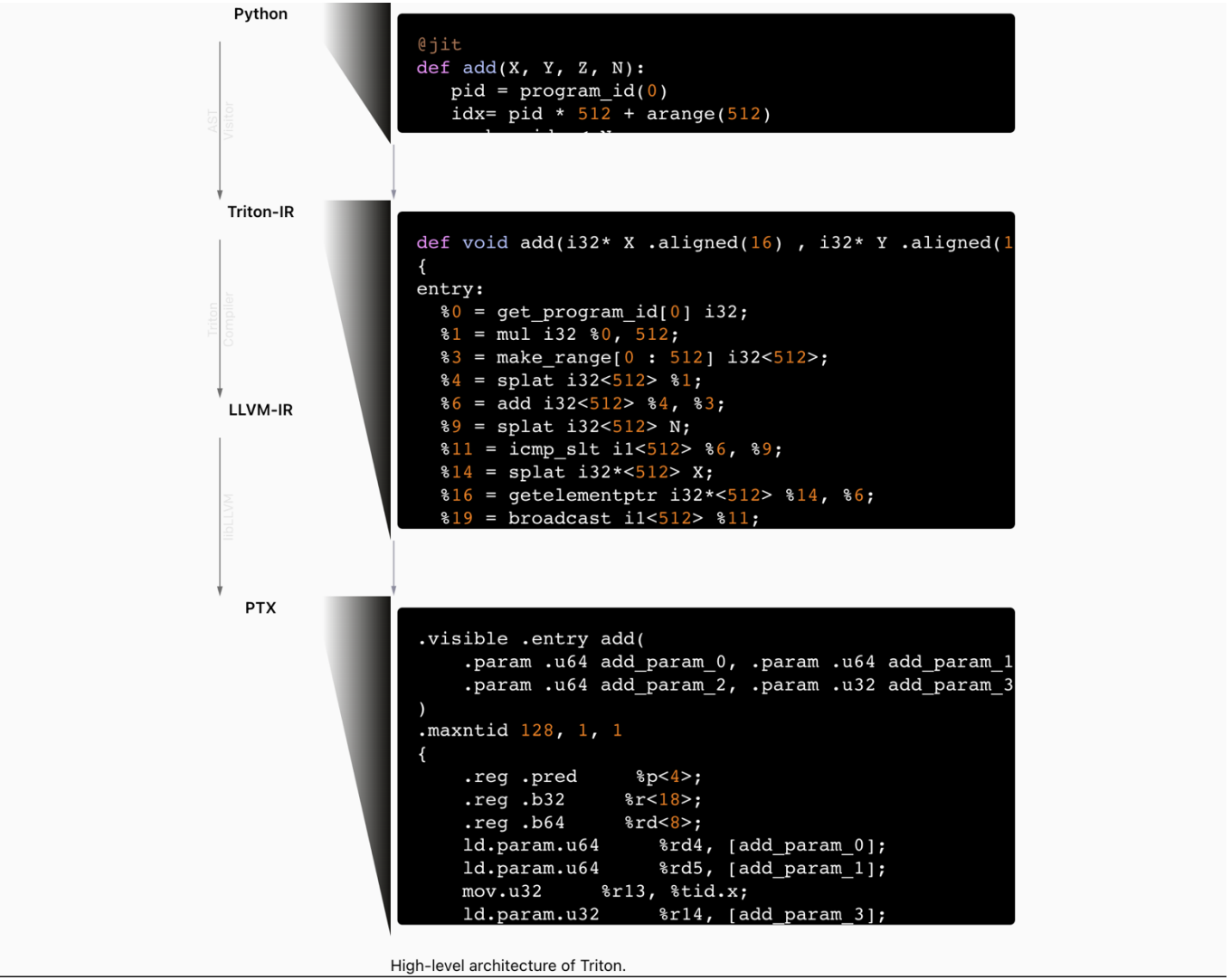
Triton Add

```
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)
```

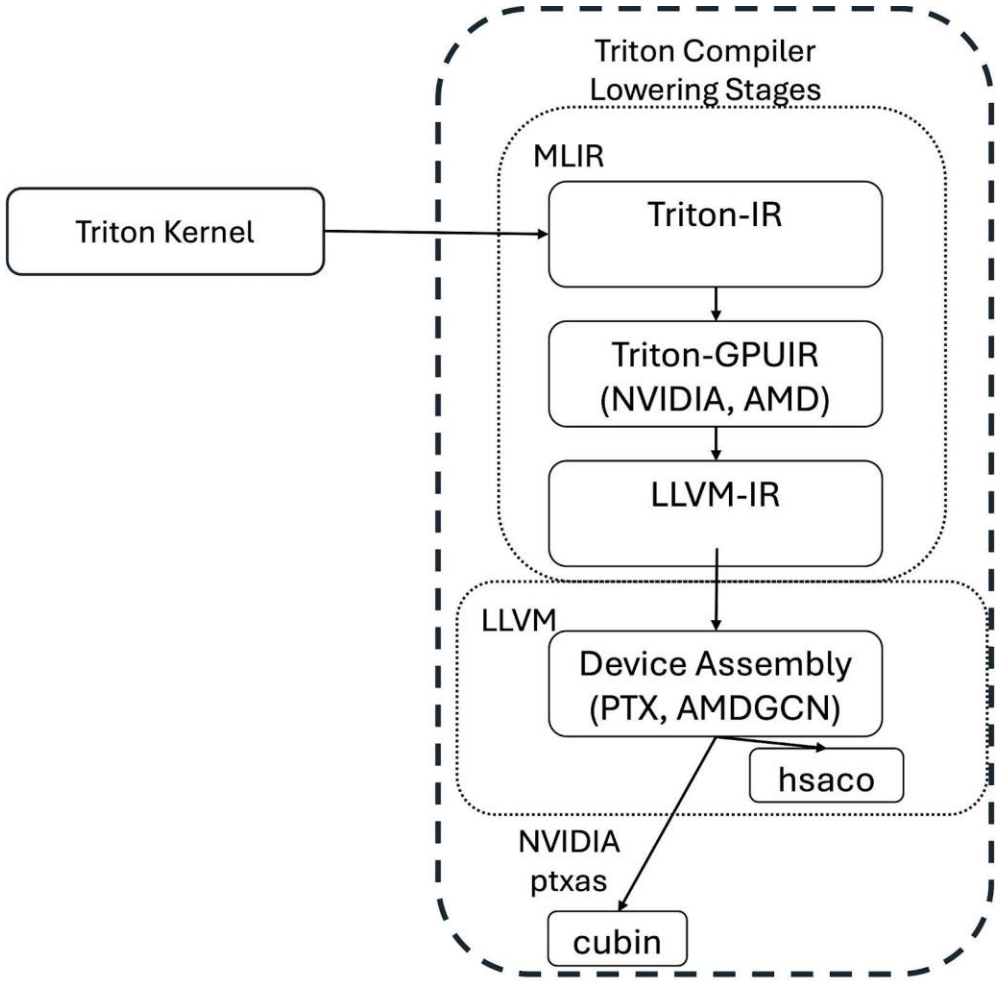
Triton Vector Add

```
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

```python
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

```python
# 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.

Tasks

- **Task 1: Precise KNN Top 10**
- **Task 2: KMeans and ANN**

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 Hobbyists Blog
  - https://codesphere.com/articles/5-best-free-cloud-gpu-providers-for-hobbyists
  - 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

GEMM Example: Optimization

- In most scenario, we can get super fast GEMM provided by cuBLAS, a linear algebra provided by NVIDIA
- Cupy/PyTorch can trigger matrix multiplication into cuBLAS GEMM automatically

- But we can do better with manually designed cuda kernel

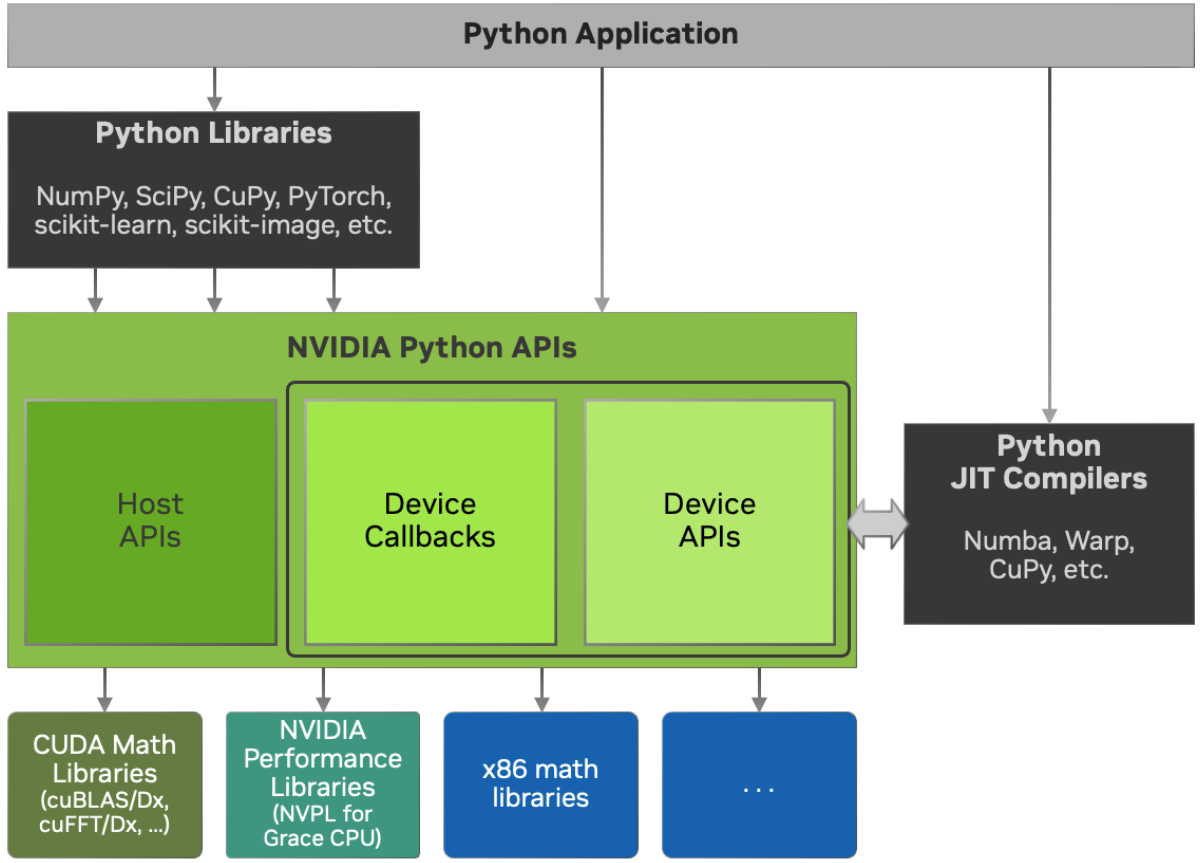
What Cupy do when it finds GEMM/GEMV

Minimal lead-time
to access new CUDA library features from Python

Inter-operability
with core Python numeric packages

Pythonic library interface
to CUDA accelerated libraries
Pythonic APIs are built on top of low-level 1:1 Python bindings to C APIs

Platform-agnostic
Runs on GPU, CPU (x86 & Arm) and Grace-Hopper

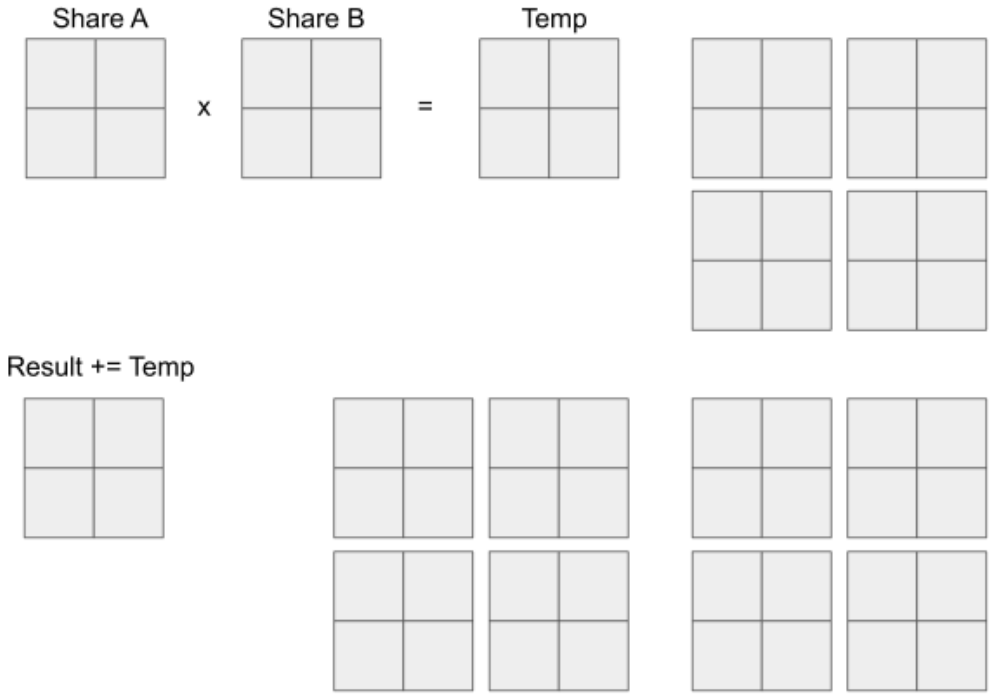


Access to all expanded and peak-performance features
Reduced & mixed precisions
Device APIs
Functional customizations (e.g. cuFFT inline callbacks)

Unified design for high-productivity & performance
End Users
Package Developers
Kernel Authors

GEMM Optimization: Tiling

- Before CUDA/GPU cuBLAS, we use OpenBLAS (open-source library), MKL(math kernel library by Intel) and many *feature optimized library.*
- They made optimization based on tensor/matrix feature, like shape, dimension, value range and so on.
- But cuBLAS use the most simple method in HPC, which is called tiling.



https://penny-xu.github.io/blog/tiled-matrix-multiplication

There are still a lot of dark knowledge 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

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/

If you want to learn more about Cuda……

- [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)