
CSIT6000V

Advanced AI Computing and Systems

· 2026 FALL ·

Tutorial 3 : CUDA Programming I

TA: Zhuo CHEN

zchenkv@connect.ust.hk



DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

- What is **CUDA (Compute Unified Device Architecture)** ?
 - CUDA® is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

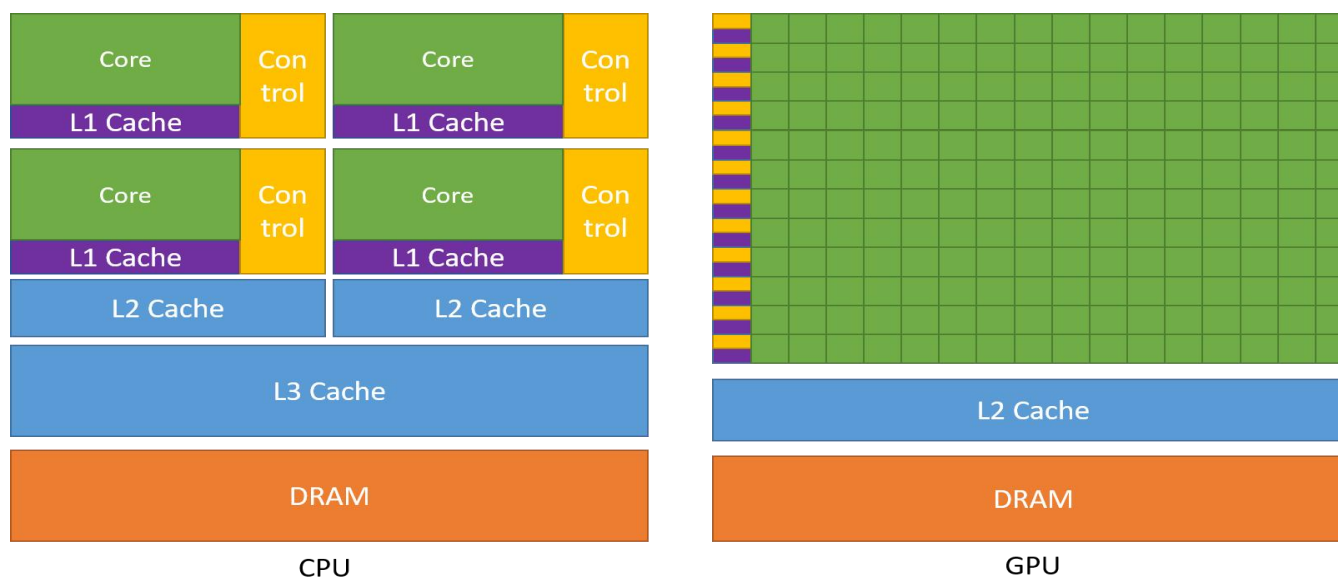


Figure 1 The GPU Devotes More Transistors to Data Processing

From One GPU to GPU Systems

Vendor	Hardware	Software Stack	
 NVIDIA	GPU	CUDA	NCCL
 AMD	GPU	ROCm / HIP	RCCL
 HUAWEI	Ascend NPU	CANN / Ascend C	HCCL
 Cambricon	MLU	Neuware / BANG C++	CNCL
 MOORE THREADS	GPU	MUSA	MCCL

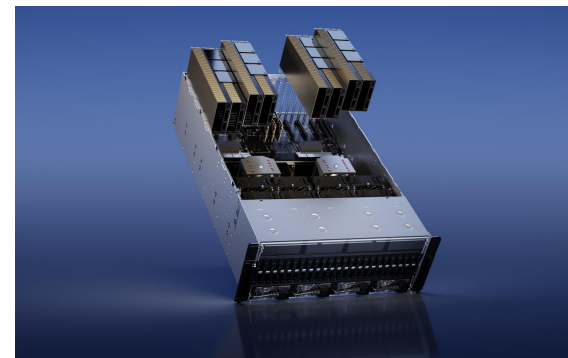


Figure 2 h100



Figure 3 NVL72

Figure 2 Different Vendor ' s Software Stack

GPU programming is no longer only about one GPU.

From One GPU to GPU Systems

KIMI K3: OPEN FRONTIER INTELLIGENCE

TECHNICAL REPORT OF KIMI K3

Kimi Team

ABSTRACT

We introduce Kimi K3, a 2.8T parameter Mixture-of-Experts model with 104 billion activated parameters, native vision capabilities, and a 1-million-token context window. Kimi K3 is built on Kimi Delta Attention [64] and Attention Residuals [58], which improve information flow across sequence length and model depth. Together with Stable LatentMoE, which effectively activates 16 of 896 routed experts per token, and refined training and data recipes, these advances yield an approximately $2.5\times$ improvement in overall scaling efficiency over Kimi K2 [59]. Post-training highlights reinforcement learning across general, agentic, and coding domains and multiple reasoning-effort levels, enabling compositional generalization and robust long horizon execution. At 2.8T scale, Kimi K3 is supported by infrastructure advances in multiple areas: algorithm-system co-design for KDA, perfectly balanced expert-parallel training with efficient memory management, million-token agentic RL with persistent rollout and sandbox states, and deployment innovations.

$$\frac{2.8T \times 16 \text{ bit}}{80 \text{ GB GPU Memory}} = 70 \text{ GPUs}$$

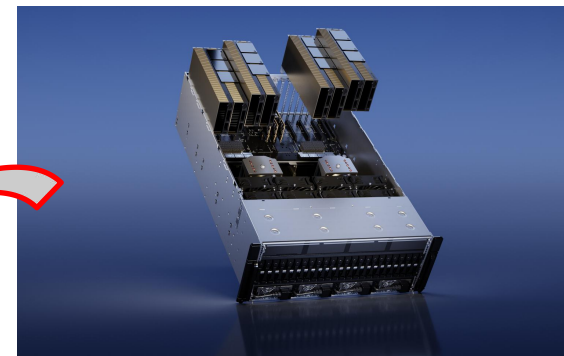


Figure 2 h100



Figure 3 NVL72

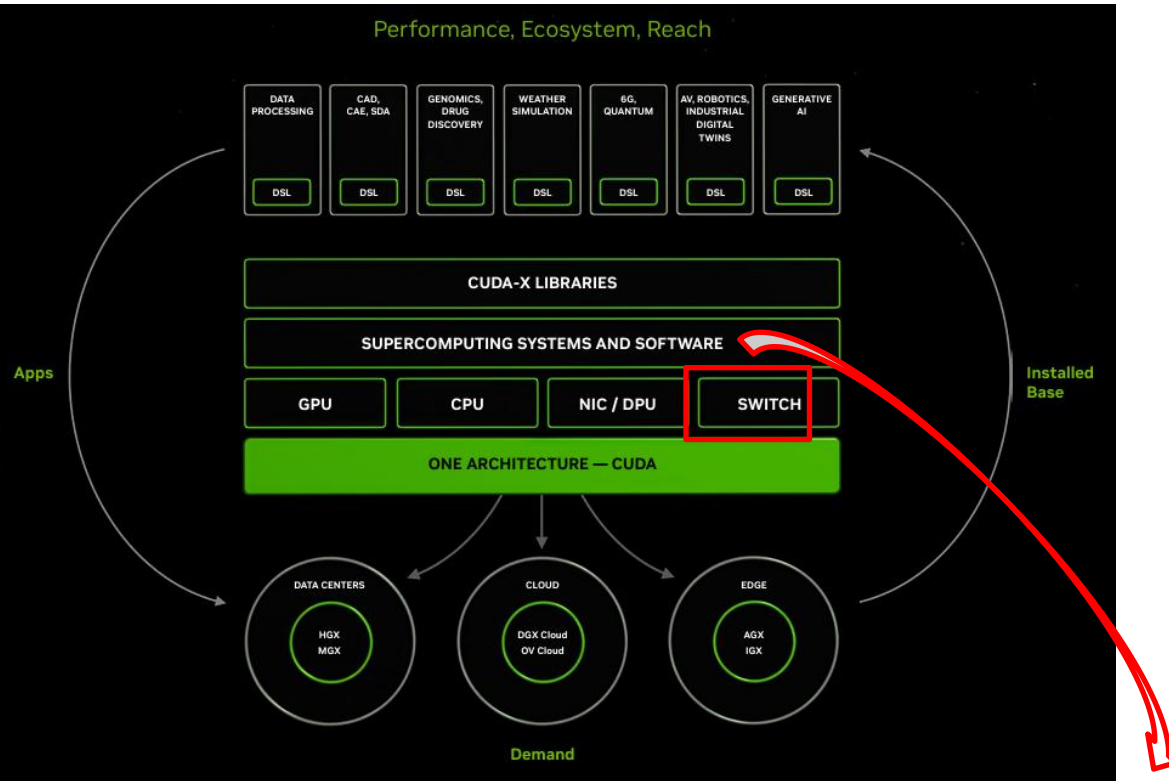
GPU programming is no longer only about one GPU.

Why CUDA?

“NVIDIA’s moat is its software and ecosystem.”

The Edge Singapore

The global NVIDIA ecosystem now approaches 5 million developers—capping a record-setting year for new developers. Forty thousand companies have worked with NVIDIA. There are now more than 3,300 GPU-accelerated applications. More than 1,600 generative AI companies are building on NVIDIA.



GPU Series

RTX 5090 (2025)
Ampere A100 (2020)
Hopper H100 (2022)
Blackwell B200(2024)

CPU Series

Grace CPU
(2021)
Vera CPU (2026)

Network Interface Cards

ConnectX CX8 (2024) CX9 (2025)
Data Processing Unit
Blue Field BF3(2021) BF4(2025)

IB Switch

Quantum
Ethernet Switch
Spectrum X

Where Does CUDA Fit?

NCCL DeepLearning stack

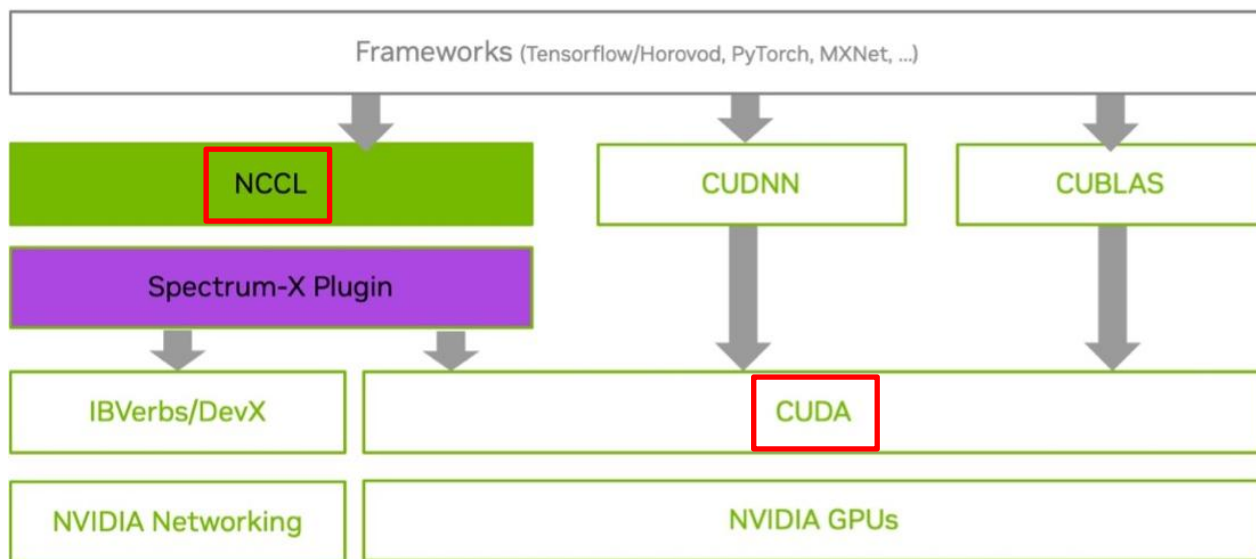


Figure 3 NCCL DeepLearning stack

- **CUDA GPU Computing**
 - Kernels
 - memory
 - GPU threads
- **NCCL GPU Communication**
 - AllReduce
 - AllGather
- **MPI Process Communication**

MPI → Process Communication / Coordination

NCCL → GPU Communication

CUDA → GPU Computing

Processes to CUDA Threads

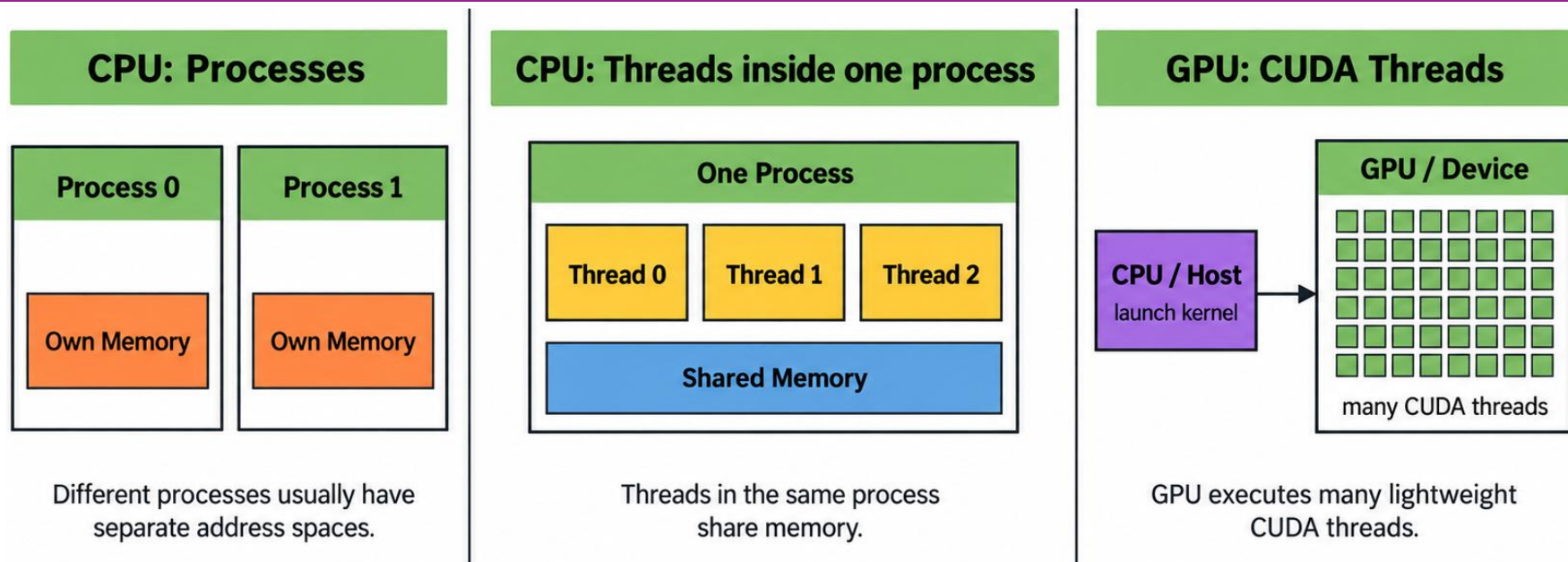


Figure 3 Processes, CPU Threads, and CUDA Threads

- **Process**
 - A running instance of a program, usually with its own address space.
- **CPU Thread**
 - An execution unit inside a process. Threads in the same process share memory.
- **CUDA Thread**
 - A lightweight execution unit running on the GPU.

How Are CUDA Threads Organized?

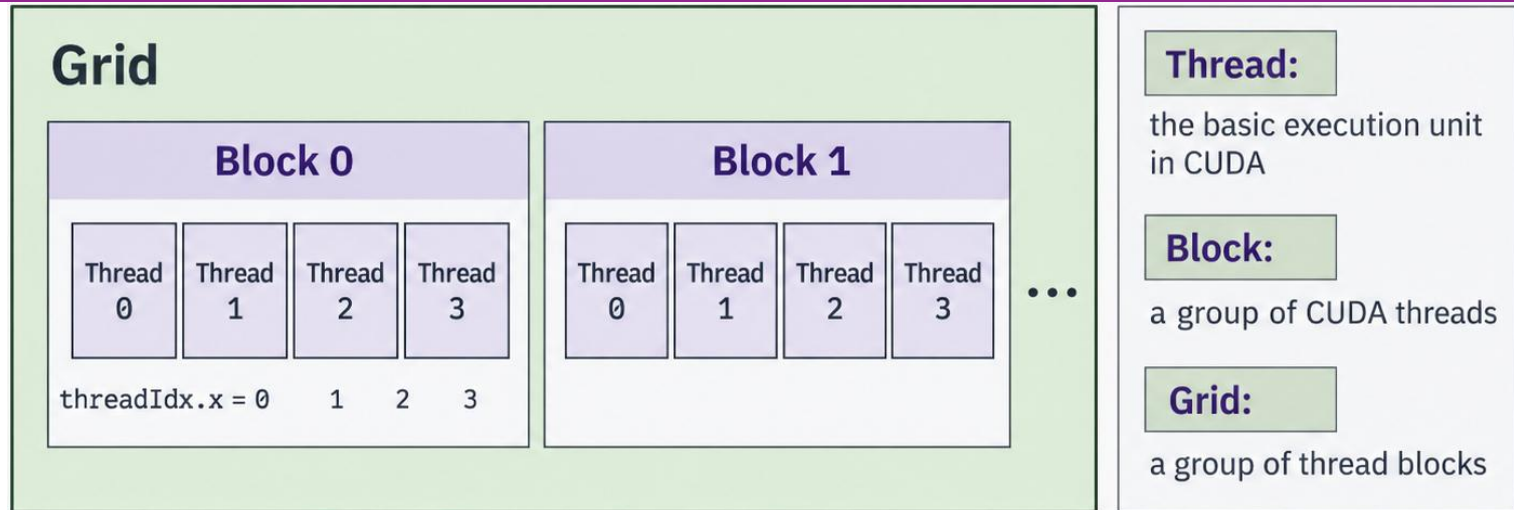


Figure 4 GPU Grid block and Threads

- Key ideas
 - A thread is the basic execution unit of a CUDA kernel.
 - A block is a group of CUDA threads.
 - A grid is a group of thread blocks
 - **threadIdx.x** is the thread's x-coordinate within its block.
- Example

```
hello<<<1, 4>>>();
```

1 block x 4 threads/block = 4 CUDA threads

MPI Processes to CUDA Threads

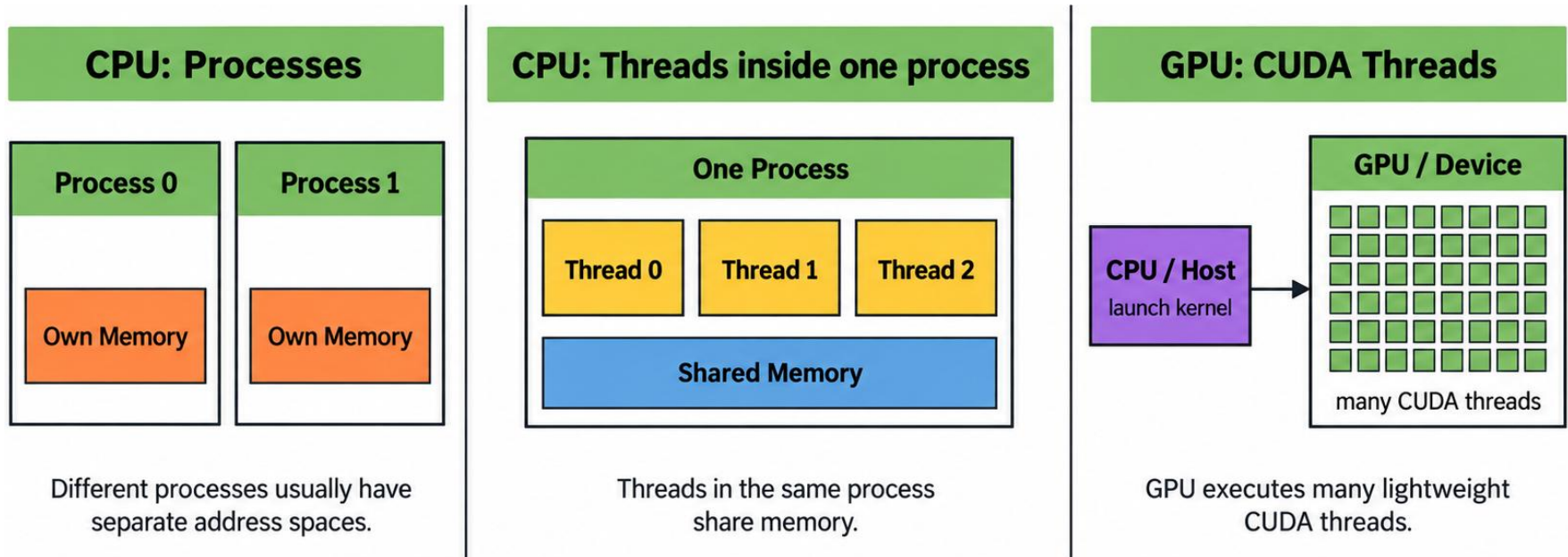


Figure 3 Processes, CPU Threads, and CUDA Threads

- Process
 - An running instance of a program, usually with its own address space.
 - CPU Thread
 - An execution unit inside a process. Threads in the same process share memory.
 - CUDA Thread
 - A lightweight execution unit running on the GPU.
- MPI → Communication between **processes**
- CUDA → Parallel computation with **GPU threads**
- NCCL → Collective communication between **GPUs**

How does CUDA work

- Host
 - The CPU side of a CUDA program
 - Prepares data and controls execution.
- Device
 - The GPU side of a CUDA program
 - Performs parallel computation
- Kernel
 - A function executed on the GPU
 - Run by many CUDA threads in parallel

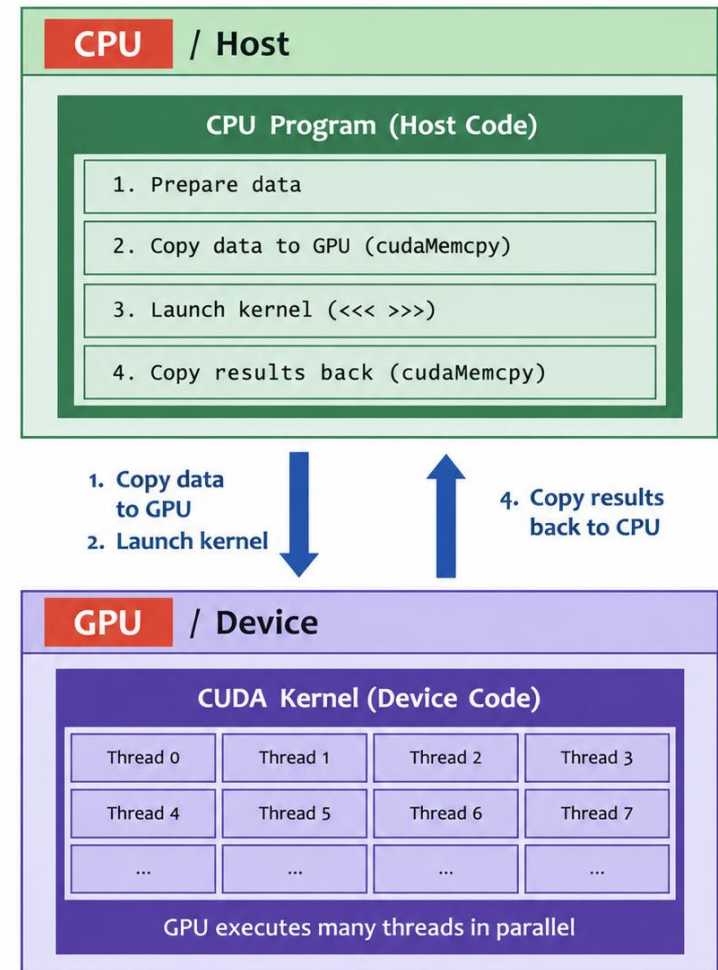


Figure 5 Processes, CUDA Host-Device Execution Model

How does CUDA work

- Host
 - The CPU side of a CUDA program
 - Prepares data and controls execution.
- Device
 - The GPU side of a CUDA program
 - Performs parallel computation
- Kernel
 - A function executed on the GPU
 - Run by many CUDA threads in parallel

Host controls the program;
Device performs the parallel work.

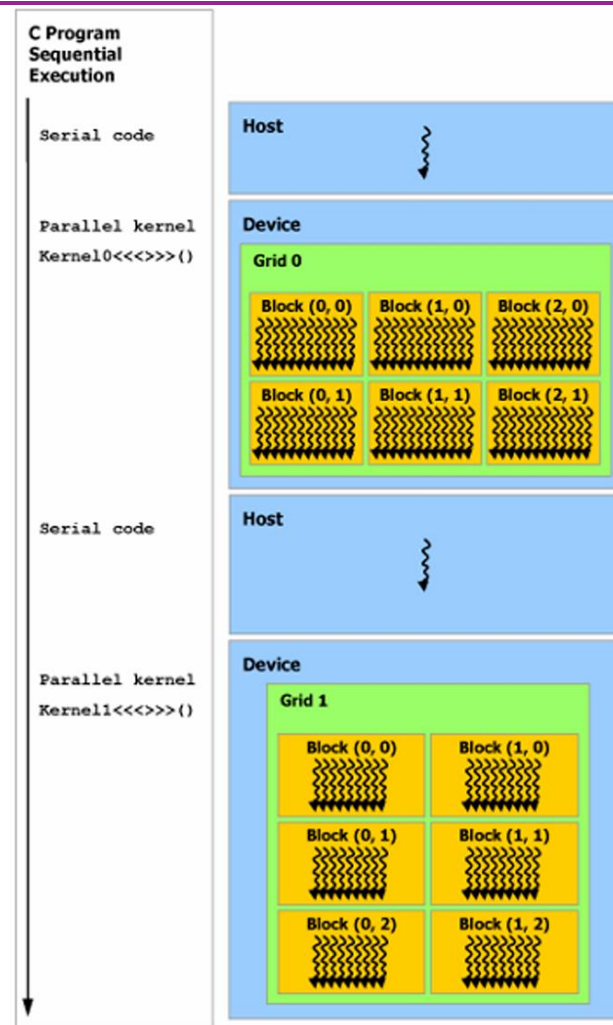


Figure 6: Heterogeneous Programming

How Do We Program CUDA

■ CUDA C ++ Used in this tutorial

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void hello()
{
    printf("Hello from GPU! Thread %d\n", threadIdx.x);
}

int main()
{
    printf("Hello from CPU!\n");

    hello<<<1, 4>>>();

    cudaDeviceSynchronize();

    return 0;
}
```

■ Other ways to use NVIDIA GPUs

□ Python

- CUDA Python
- PyTorch
- CuPy
- Numba

■ Others

- CUDA Fortran
- Java bindings



We use CUDA C++ for it exposes the programming model directly.

Where Do We Program CUDA

You can **write** CUDA code almost anywhere, but it must eventually **run** on a machine with a CUDA-capable NVIDIA GPU.

1 Local GPU

Your own computer with NVIDIA GPU



e.g., laptop with RTX 4060

Write code → Compile & run locally

- ✓ Directly on your own machine
- ✓ Fast and convenient for learning
- ✓ Good for small experiments and debugging

2 Remote GPU Server

SSH to a server with GPU

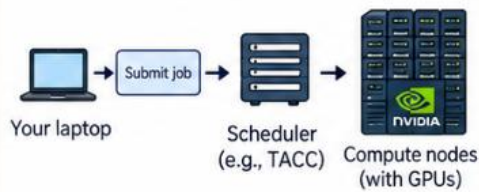


Write code locally → SSH to server and run

- ✓ Very common in research
- ✓ Your laptop does not need a GPU
- ✓ Compile and run on the remote server

3 GPU Cluster (e.g., TACC)

Submit a job and use allocated GPUs



Prepare code and job script → Submit job (e.g., sbatch) → Run on allocated GPUs

- ✓ For larger experiments
- ✓ Shared resources (multi-user)
- ✓ Need to use the cluster's job scheduler
- ✓ Common in research and production

4 Web / Cloud

Use online GPU environments



Write code in browser → Run on cloud GPU

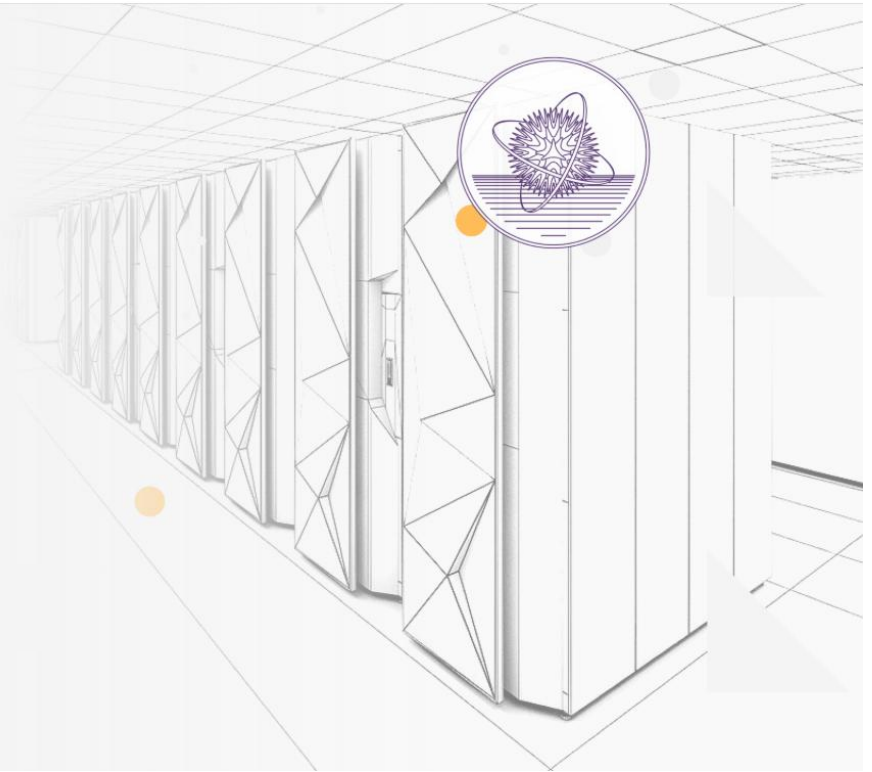
- ✓ No local installation needed
- ✓ Good for quick start
- ✓ Convenient for beginners
- ✓ Still runs on a remote NVIDIA GPU

Classroom Demonstration: by WEB <https://cuda.godbolt.org>

Other Choices: LeetGPU Playground / cuda.live Playground

Scalable AI Infrastructure Designed for Evolving Machine Learning Research

Tap into the Power of Turing AI Computing Cloud (TACC)



TACC is an efficient AI computing infrastructure optimized for machine learning applications in large-scale GPU clusters

Homework subbmission (by submit a Job): <https://tacc.ust.hk/>

Anatomy of a CUDA C++ Program

```
#include <stdio.h>
#include <cuda_runtime.h>

// GPU code
__global__ void kernel(...)
{
    // GPU computation
}

// CPU code
int main()
{
    // 1. Prepare data on the CPU

    // 2. Allocate memory on the GPU
    cudaMalloc(...);

    // 3. Copy input data from CPU to GPU
    cudaMemcpy(...);

    // 4. Launch the CUDA kernel
    kernel<<...>>>(...);

    // 5. Wait for the GPU to finish
    cudaDeviceSynchronize();

    // 6. Copy results from GPU to CPU
    cudaMemcpy(...);

    // 7. Free GPU memory
    cudaFree(...);

    return 0;
}
```

Two Parts in One .cu File

- GPU / Device Code
 - Defines the CUDA kernel
 - Executed on the GPU
 - Run by many CUDA threads
- CPU / Host Code
 - Prepares input data
 - Manages GPU memory
 - Transfers data between CPU and GPU
 - Launches the CUDA kernel
 - Collects the results

A CUDA C++ program contains both CPU code and GPU code.

Kernel Launch and Device printf

```

#include <stdio.h>
#include <cuda_runtime.h>

// GPU code CUDA kernel executed on the GPU
__global__ void hello()
{
    // Each CUDA thread executes this once
    printf("Hello from GPU! Thread %d\n", threadIdx.x);
}

// CPU code
int main()
{
    // 1. Prepare data on the CPU No input data is needed in this simple example
    printf("Hello from CPU!\n");

    // 2. Allocate memory on the GPU Not needed because there is no in/output array

    // 3. Copy input data from CPU to GPU Not needed because there is no input data

    // 4. Launch the CUDA kernel 1 block, 4 CUDA threads
    hello<<<1, 4>>>();

    // 5. Wait for the GPU to finish
    cudaDeviceSynchronize();

    // 6. Copy results from GPU to CPU Not needed because there is no output data

    // 7. Free GPU memory Not needed because no GPU memory was allocated.

    return 0;
}
```

Expected Output

```

Hello from CPU!
Hello from GPU! Thread 0
Hello from GPU! Thread 1
Hello from GPU! Thread 2
Hello from GPU! Thread 3
```

Why four GPU messages?

$$\begin{aligned} & \mathbf{1} \text{ block} \times \mathbf{4} \text{ threads} \\ & = \\ & \mathbf{4} \text{ CUDA threads} \end{aligned}$$

Each CUDA thread executes hello() once.

NVCC Compiles a CUDA Program

The screenshot displays the Visual Studio Code interface with the NVCC 12.5.1 compiler. The interface is divided into four main panes:

- Source Code (CUDA C++):** Shows the CUDA C++ source code. It includes a GPU kernel executed on the GPU and CPU code. The GPU kernel is a simple "hello" function that prints "Hello from GPU! Thread %d\n", where %d is the thread index. The CPU code calls the GPU kernel and prints "Hello from CPU!\n".
- Host Code (CPU Assembly):** Shows the CPU assembly code generated by the compiler. It includes instructions for loading the GPU kernel, launching the GPU, and copying data between the CPU and GPU.
- Device Code (PTX):** Shows the PTX code generated for the GPU. It includes instructions for loading the GPU kernel, launching the GPU, and copying data between the CPU and GPU.
- Output:** Shows the compilation output. It includes the return codes for the ASM and build compilers, and the execution output of the program. The output shows "Hello from CPU!" and "Hello from GPU! Thread 0" through "Hello from GPU! Thread 3".

CUDA C++
Source Code

Host Code
(CPU Assembly)

Device Code
(PTX)

Output

1 D Vector Addition

Vector A

index	0	1	2	3	4	5	6	7
A	1	2	3	4	5	6	7	8
	+	+	+	+	+	+	+	+
B	10	20	30	40	50	60	70	80
	↓	↓	↓	↓	↓	↓	↓	↓
C	11	22	33	44	55	66	77	88



```
for (int i = 0; i < N; i++)  
{  
    C[i] = A[i] + B[i];  
}
```

$$\forall i \in \{0, 1, \dots, N-1\}, C = A + B = \begin{bmatrix} A_0 + B_0 \\ \vdots \\ A_{N-1} + B_{N-1} \end{bmatrix}$$

$$A, B, C \in \mathbb{R}^N$$

$$\forall i \neq j, C_i \text{ is independent of } (A_j, B_j)$$

One output element $C_i \longleftrightarrow$ One CUDA thread

Mapping CUDA Threads to 1D Data

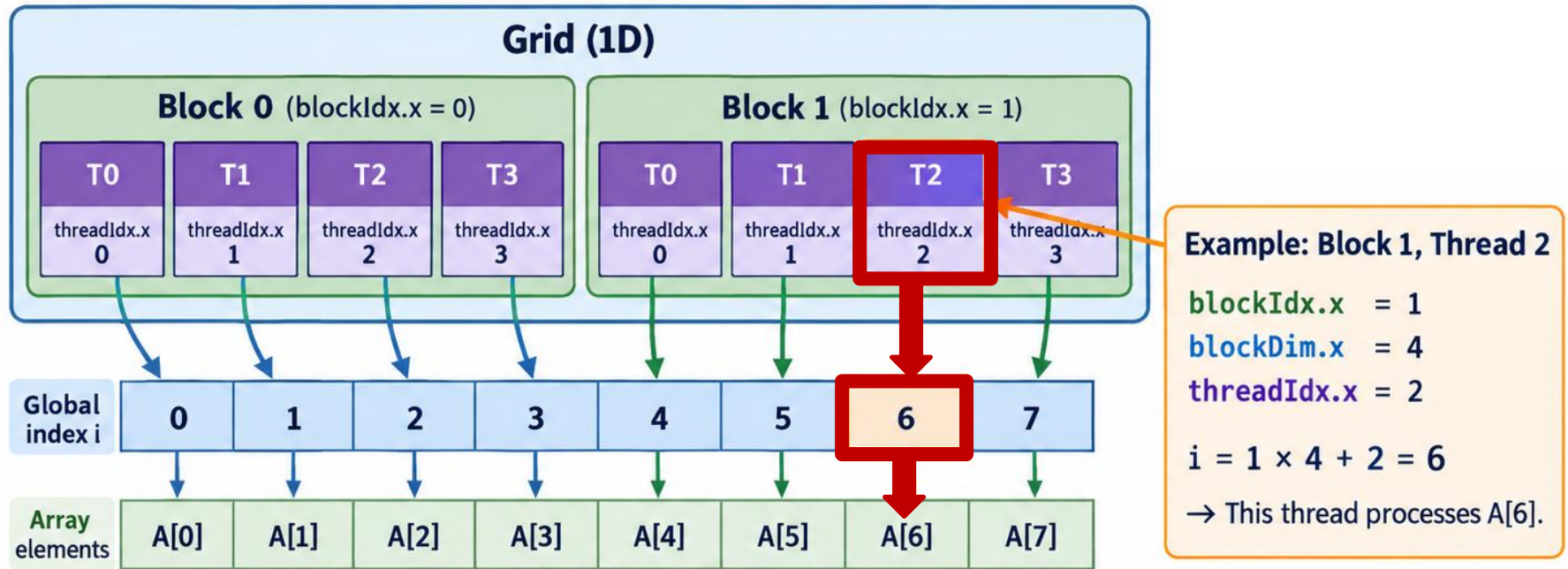


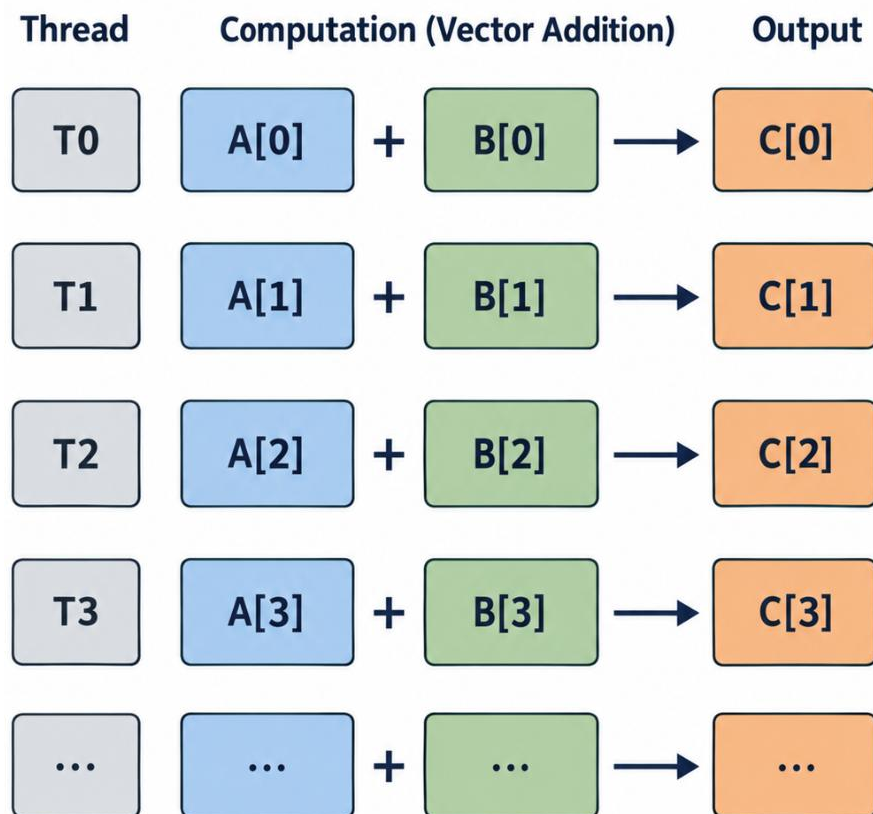
Figure 7 Mapping CUDA Threads to 1D Data

$$i = blockIdx.x \times blockDim.x + threadIdx.x$$

- $blockIdx.x$ Which block am I in?
- $blockDim.x$ How many threads are in one block?
- $threadIdx.x$ Which thread am I inside this block?

1D CUDA Kernel for Addition

One Thread, One Output Element



```
__global__ void vectorAdd(int *A, int *B, int *C, int N)
{
    int i = blockIdx.x * blockDim.x
           + threadIdx.x;

    if (i < N)
    {
        C[i] = A[i] + B[i];
    }
}
```

- `int i = ...` Which element do I process?
- `if (i < N)` Is the index valid?
- `C[i] = A[i] + B[i]` Do my computation

Figure 8 CUDA Kernel for Vector Addition

Each CUDA thread computes one output element

Data Movement and Execution Flow

GPU code (Device code)

```
__global__ void vectorAdd(int *A, int *B, int *C, int N)
{
    // 1. Each CUDA thread computes one output element
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. Check whether the index is valid
    if (i < N)
    {
        C[i] = A[i] + B[i];
    }
}
```

Each thread gets
its own index i

Read $A[i]$

Read $B[i]$

Write $C[i]$

GPU device memory (on GPU)

d_A
(int *)

d_B
(int *)

d_C
(int *)

Launch kernel & pass arguments

$d_A \rightarrow A$
 $d_B \rightarrow B$
 $d_C \rightarrow C$
 $N \rightarrow N$

CPU code (Host code)

```
int main()
{
    const int N = 8;

    // 1. Prepare data on the CPU
    int A[N] = {1, 2, 3, 4, 5, 6, 7, 8};
    int B[N] = {10, 20, 30, 40, 50, 60, 70, 80};
    int C[N] = {0};

    int *d_A;
    int *d_B;
    int *d_C;
    int bytes = N * sizeof(int);

    // 2. Allocate memory on the GPU
    cudaMalloc((void**)&d_A, bytes);
    cudaMalloc((void**)&d_B, bytes);
    cudaMalloc((void**)&d_C, bytes);

    // 3. Copy input data from CPU to GPU
    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    // 4. Launch the CUDA kernel
    int threadsPerBlock = 4;
    int blocks = (N + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocks, threadsPerBlock>>>(&d_A, &d_B, &d_C, N);

    // 5. Wait for the GPU to finish
    cudaDeviceSynchronize();

    // 6. Copy results from GPU to CPU
    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    // Print results
    printf("C = ");
    for (int i = 0; i < N; i++)
    {
        printf("%d ", C[i]);
    }
    printf("\n");

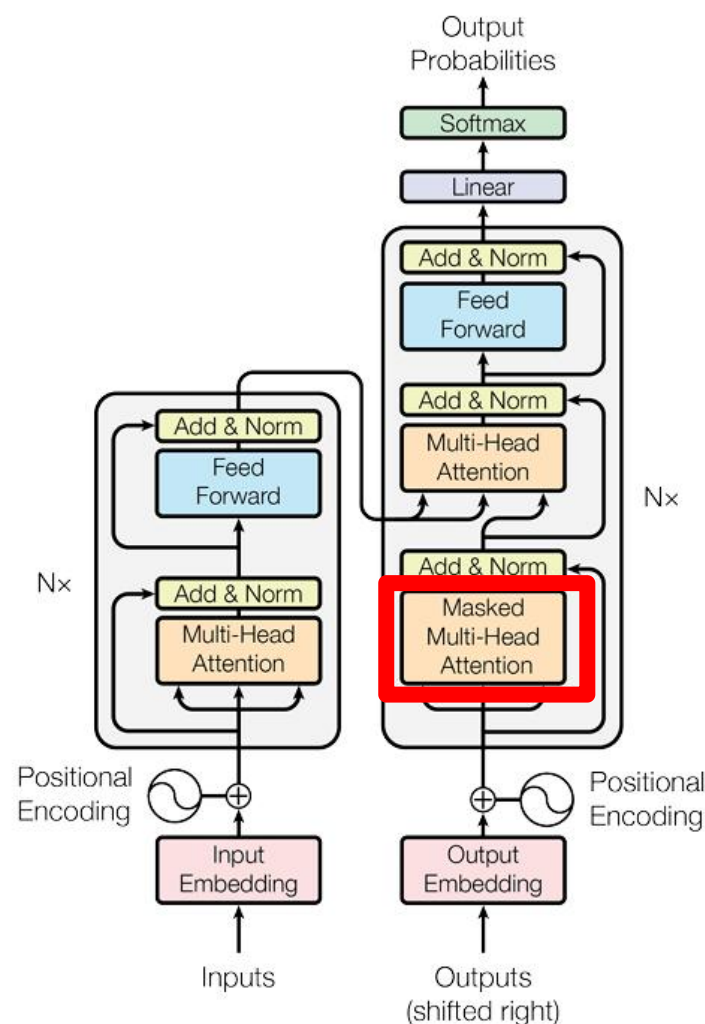
    // 7. Free GPU memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return 0;
}
```

1 Allocate GPU memory
(d_A , d_B , d_C)

2 Copy A, B to GPU
($A \rightarrow d_A$, $B \rightarrow d_B$)

4 Copy result back to CPU
($d_C \rightarrow C$)

2 D Matrix Addition



$$S'_{i,j} = S_{i,j} + M_{i,j}$$

$$S, M, S' \in (\mathbb{R} \cup \{\pm\infty\})^{L \times L}$$

- L : sequence length
- S : attention score matrix
- M : attention mask

$$\begin{bmatrix} S_{00} & S_{01} & S_{02} & S_{03} \\ S_{10} & S_{11} & S_{12} & S_{13} \\ S_{20} & S_{21} & S_{22} & S_{23} \\ S_{30} & S_{31} & S_{32} & S_{33} \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} S_{00} & -\infty & -\infty & -\infty \\ S_{10} & S_{11} & -\infty & -\infty \\ S_{20} & S_{21} & S_{22} & -\infty \\ S_{30} & S_{31} & S_{32} & S_{33} \end{bmatrix}$$

The same operation is applied independently to every (row, col).

Figure 9 The Transformer model architecture

Mapping CUDA Threads to 2D Data

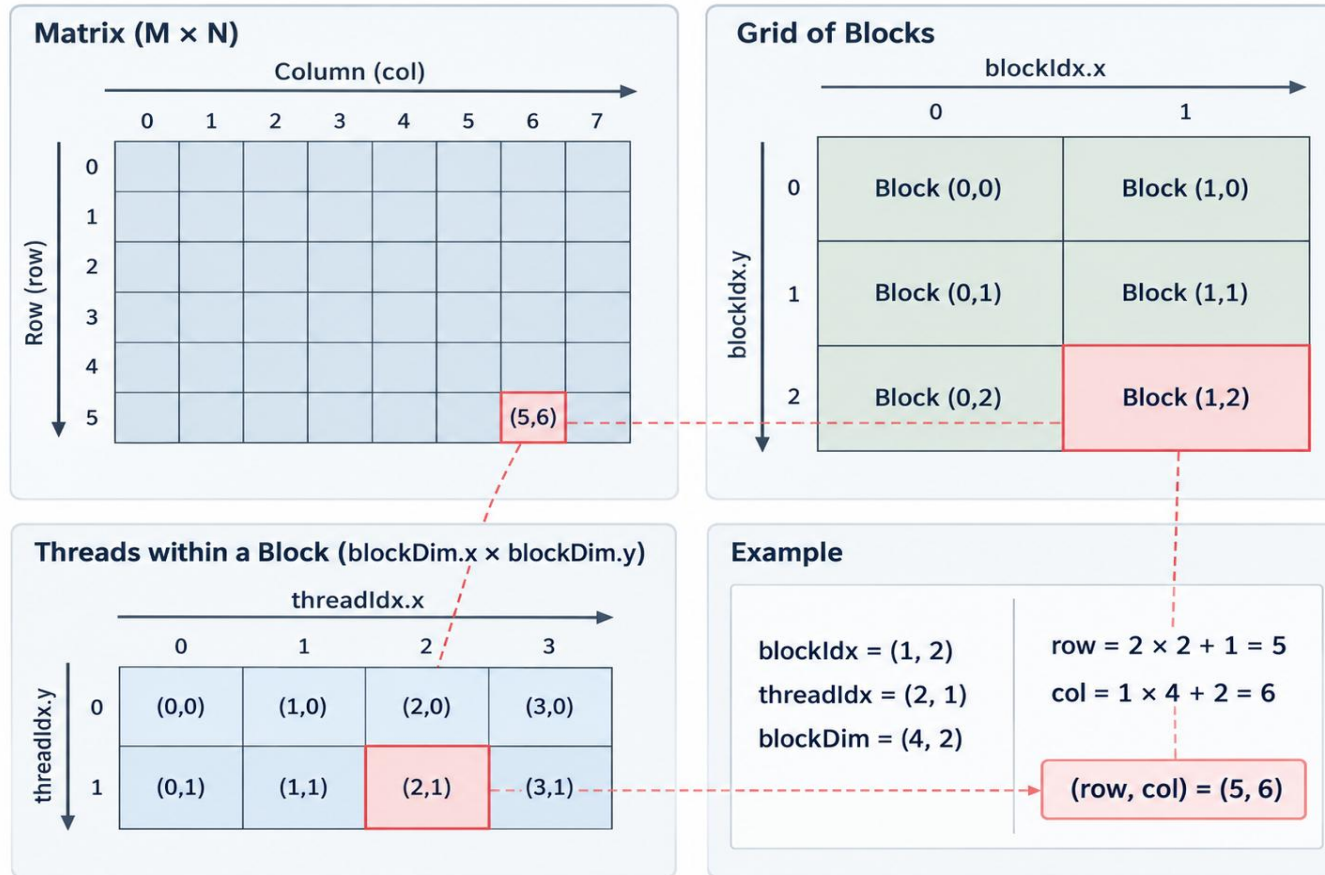


Figure 10 Mapping 2D Threads to a Matrix

$$col = blockIdx.x \times blockDim.x + threadIdx.x$$
$$row = blockIdx.y \times blockDim.y + threadIdx.y$$

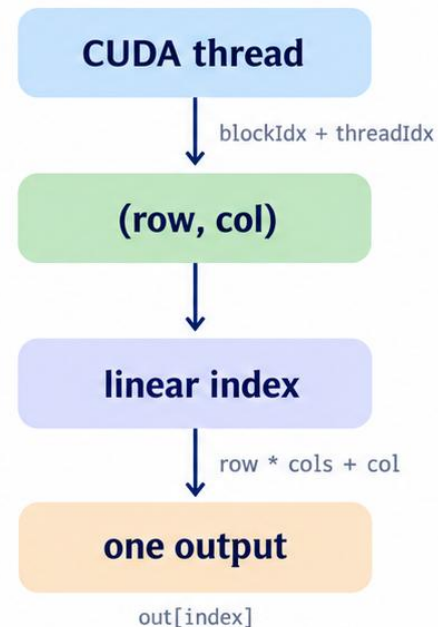
2D CUDA Kernel for Addition

```
__global__ void addMask(const float *S, const float *M, float *out, int rows, int cols)
{
    // 1. Find my row
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // 2. Find my column
    int col = blockIdx.x * blockDim.x + threadIdx.x;

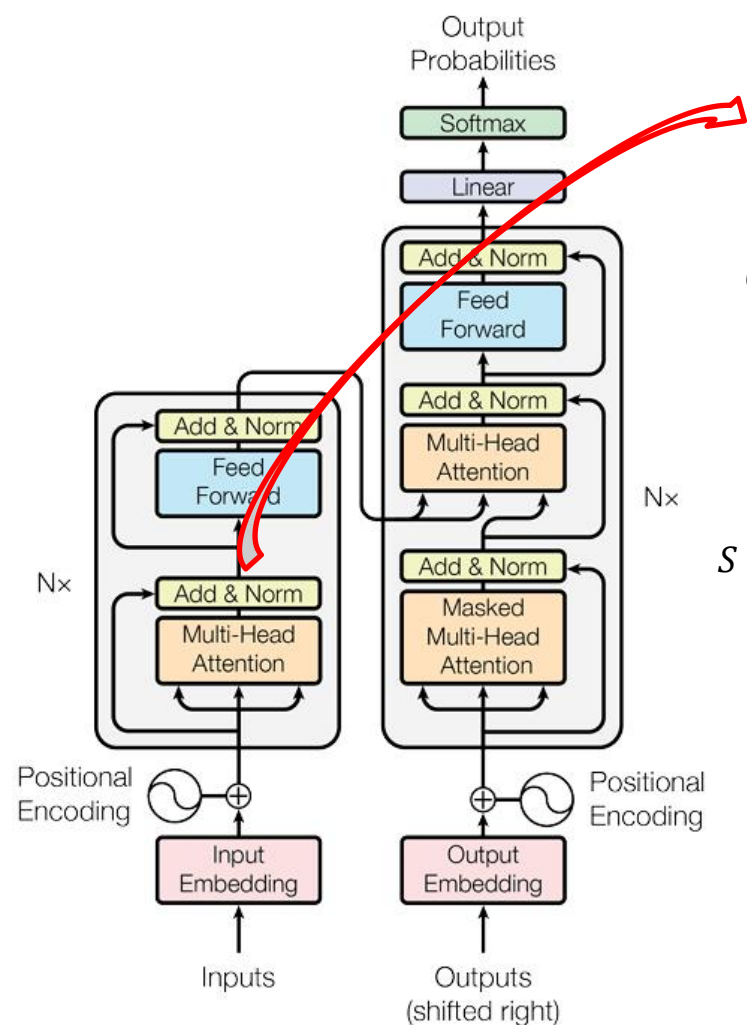
    // 3. Check whether this matrix element exists
    if (row < rows && col < cols)
    {
        // 4. Convert (row, col) to a linear memory index
        int index = row * cols + col;

        // 5. Compute one output element
        out[index] = S[index] + M[index];
    }
}
```



$(blockIdx.x, threadIdx.x)$	$\rightarrow col$
$(blockIdx.y, threadIdx.y)$	$\rightarrow row$
(row, col)	$\rightarrow row \times cols + col$
$index$	$\rightarrow out[index]$

2 D Matrix Multiplication



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1d_k} \\ q_{21} & q_{22} & \cdots & q_{2d_k} \\ \vdots & \vdots & \ddots & \vdots \\ q_{L1} & q_{L2} & \cdots & q_{Ld_k} \end{bmatrix}, \quad K^T = \begin{bmatrix} k_{11} & k_{21} & \cdots & k_{L1} \\ k_{12} & k_{22} & \cdots & k_{L2} \\ \vdots & \vdots & \ddots & \vdots \\ k_{1d_k} & k_{2d_k} & \cdots & k_{Ld_k} \end{bmatrix} \in \mathbb{R}^{d_k \times L}$$

$$S = QK^T = \begin{bmatrix} \sum_{r=1}^{d_k} q_{1r} k_{1r} & \sum_{r=1}^{d_k} q_{1r} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{1r} k_{Lr} \\ \sum_{r=1}^{d_k} q_{2r} k_{1r} & \sum_{r=1}^{d_k} q_{2r} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{2r} k_{Lr} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=1}^{d_k} q_{Lr} k_{1r} & \sum_{r=1}^{d_k} q_{Lr} k_{2r} & \cdots & \sum_{r=1}^{d_k} q_{Lr} k_{Lr} \end{bmatrix} \in \mathbb{R}^{L \times L}$$

$$S_{ij} = Q_{i,:} K_{j,:}^T = [q_{i1} \quad q_{i2} \quad \cdots \quad q_{id_k}] \begin{bmatrix} k_{j1} \\ k_{j2} \\ \vdots \\ k_{jd_k} \end{bmatrix} = \sum_{r=1}^{d_k} q_{ir} k_{jr} \in \mathbb{R}$$

Matrix Multiplication

$$(i, j) \mapsto S_{ij} = \sum_{r=1}^{d_k} Q_{ir} K_{jr}$$

Figure 9 The Transformer model architecture

Toward Efficient Matrix Computation

304040 + 505040 = ? 809080 easy to calculate

304040 * 505040 = ? difficult to do mental arithmetic

Standard block multiplication

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Needs 8 block multiplications + 4 block additions

$$\Theta(N^3)$$

Strassen's idea

1969

Seven intermediate products (only 7 block multiplications):

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Recombine to get the blocks of $C = AB$:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Needs 7 block multiplications + more additions/subtractions

$$\Theta(n^{\log_2^7}) = \Theta(n^{2.807})$$

Less expensive multiplications

More cheaper addition

Toward Efficient Matrix Computation

$$\begin{bmatrix} 4 & 0 & 4 & 0 & 3 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \\ 5 \\ 0 \end{bmatrix} = ?$$

$$4 \times 4 + 0 \times 0 + 4 \times 5 + 0 \times 0 + 3 \times 5 + 0 \times 0 = 51$$

6 multiplications + 5 additions

$$4 \times 4 + 4 \times 5 + 3 \times 5 = 51$$

3 multiplications + 2 additions **FASTER**

1. Sparse matrix

$$A = \begin{bmatrix} 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Most entries are zero

Only a small fraction are nonzero
(e.g., 5 nonzeros out of 25 entries = 20%)

Dense view: store every entry.

Sparse view: store only nonzero
values + positions.

2. Special storage formats

Coordinate format (COO)

Store a list of (row, column, value) for each nonzero.

values	4	-2	3	5	1
row index	1	2	3	4	5
col index	3	5	1	4	2

Compressed Sparse Row (CSR)

Store values and column indices, with a row pointer.

values	4	-2	3	5	1	
col index	3	5	1	4	2	
row pointer	0	1	2	3	4	5

Store only what matters

3. Special computation

Sparse matrix-vector multiplication: $y = Ax$

$$A = \begin{bmatrix} 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \rightarrow y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

- Skip multiplications by zero (use only nonzeros)
- Fewer memory accesses
- Common kernels: SpMV (Ax), SpMM (AB)

Performance depends on sparsity pattern
and memory access regularity.

Reduce both storage and computation

A simple learning path

1. Find independent work



2. Map work to threads



3. Compute the index



4. Write one output

1D or 2D, the key question is always the same:

For these introductory element-wise kernels, each thread owns one output element.

Examples

1D example
vectorAdd

$$i = blockIdx.x \times blockDim.x + threadIdx.x$$

2D example
addMask

$$\begin{aligned} col &= blockIdx.x \times blockDim.x + threadIdx.x \\ row &= blockIdx.y \times blockDim.y + threadIdx.y \end{aligned}$$