System for Artificial Intelligence

GPU Architecture & CUDA Programming

Siyuan Feng
Shanghai Innovation Institute

Recap: Overview of Machine Learning Systems

ML Models

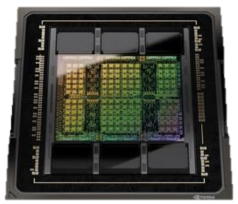
Automatic Differentiation

Graph Optimization

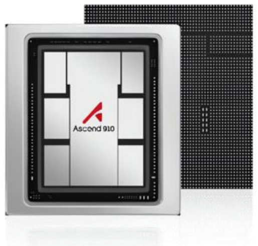
Parallelism / Distributed

Hardware Acceleration

This Lecture



NVIDIA GPU



HUAWEI NPU

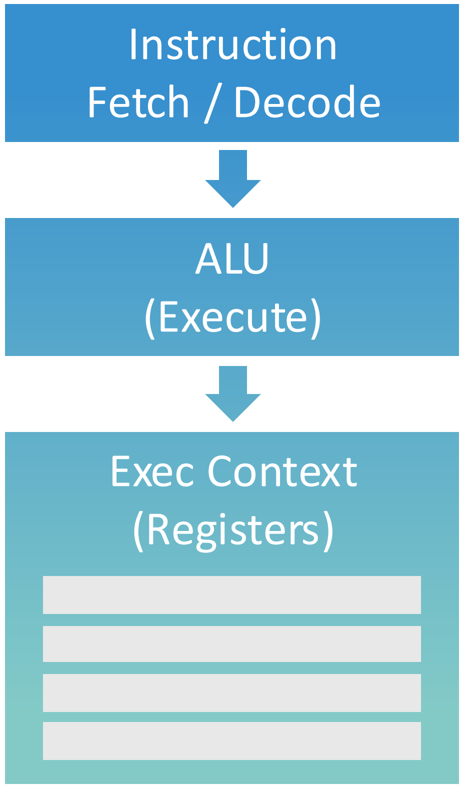
Mobile devices

OUTLINE

01

Parallel Computing

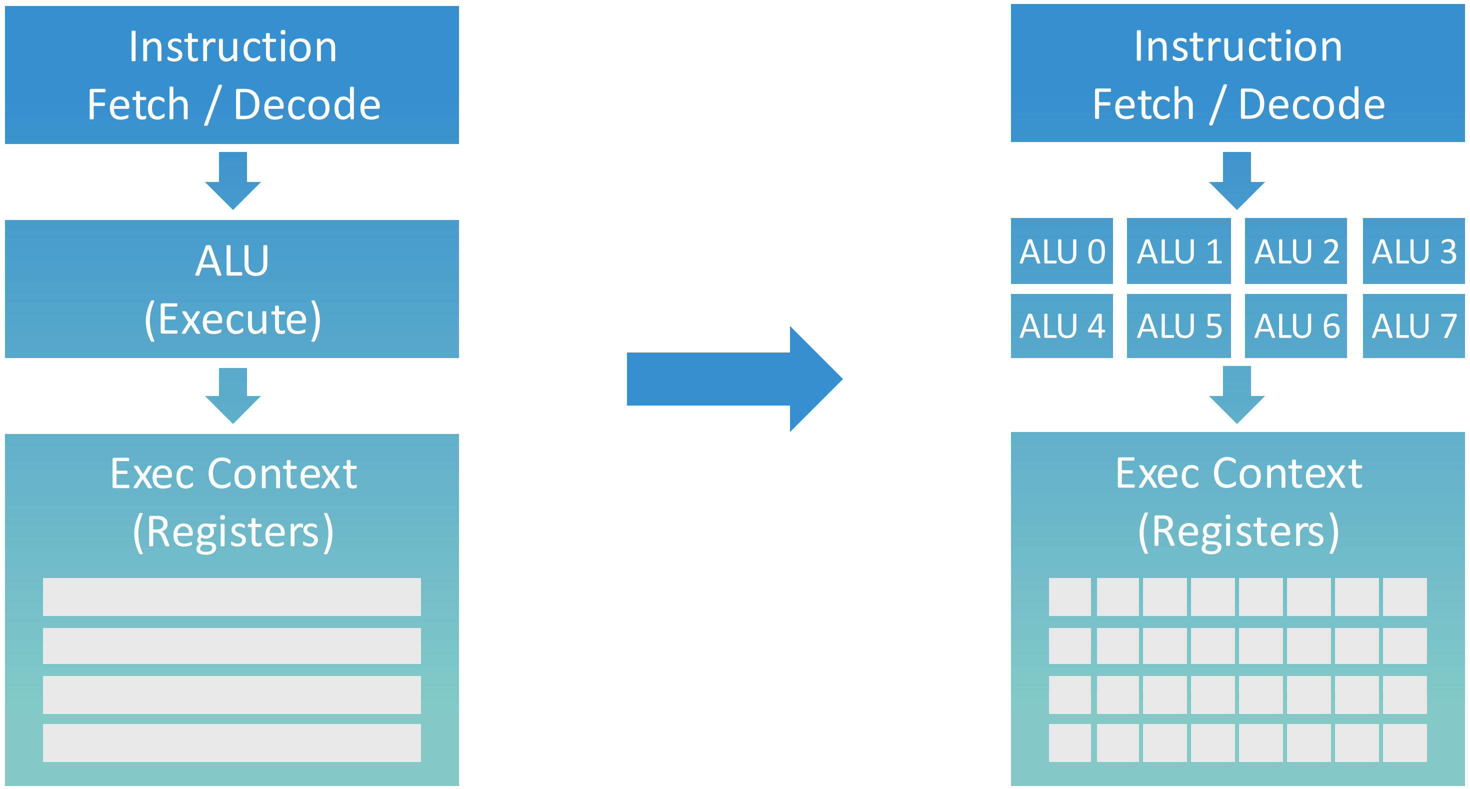
SISD: Single Instruction, Single Data



Conventional single instruction, single data processor

SIMD: Single Instruction, Multiple Data

上海创智学院 Shanghai Innovation Institute

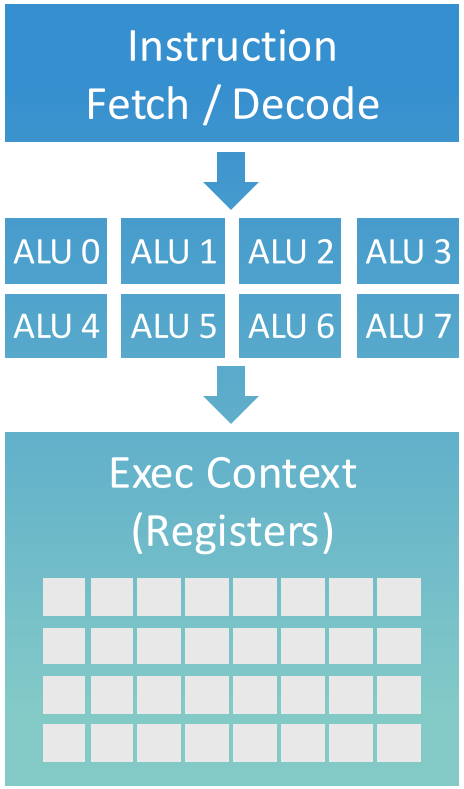


Conventional single instruction, single data processor

Modern single instruction, multiple data processor

SIMD: Single Instruction, Multiple Data

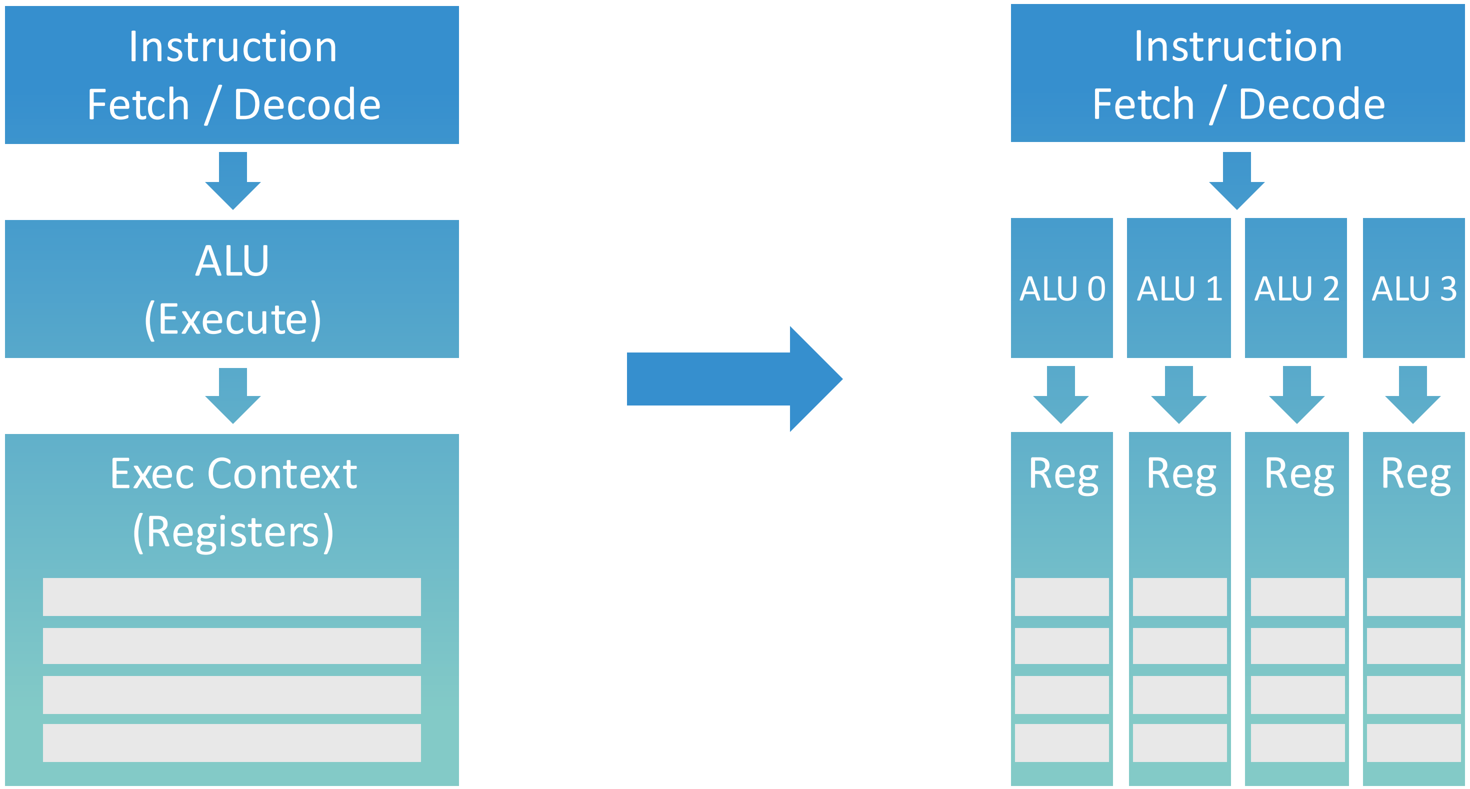
- Same instruction broadcast and executed in parallel on all ALUs
- Add ALUs to increase compute capability
- Usually used as **Vectorize**



Modern single instruction, multiple data processor

SIMT: Single Instruction, Multiple Thread

上海创智学院 Shanghai Innovation Institute

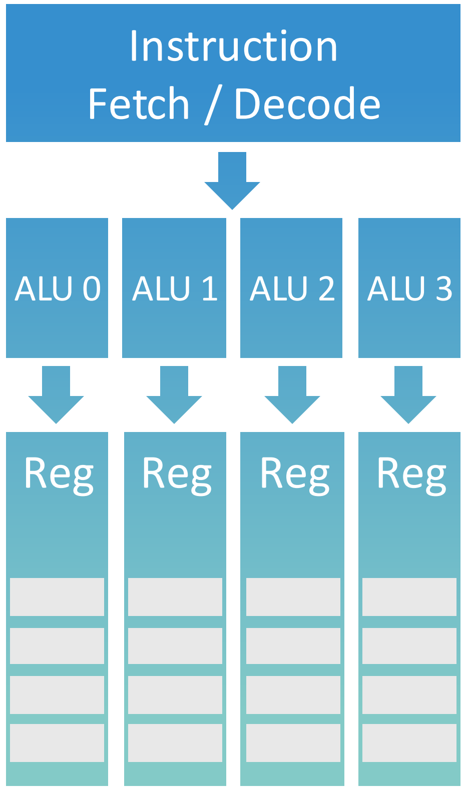


Conventional single instruction, single data processor

Modern single instruction, multiple thread GPUs

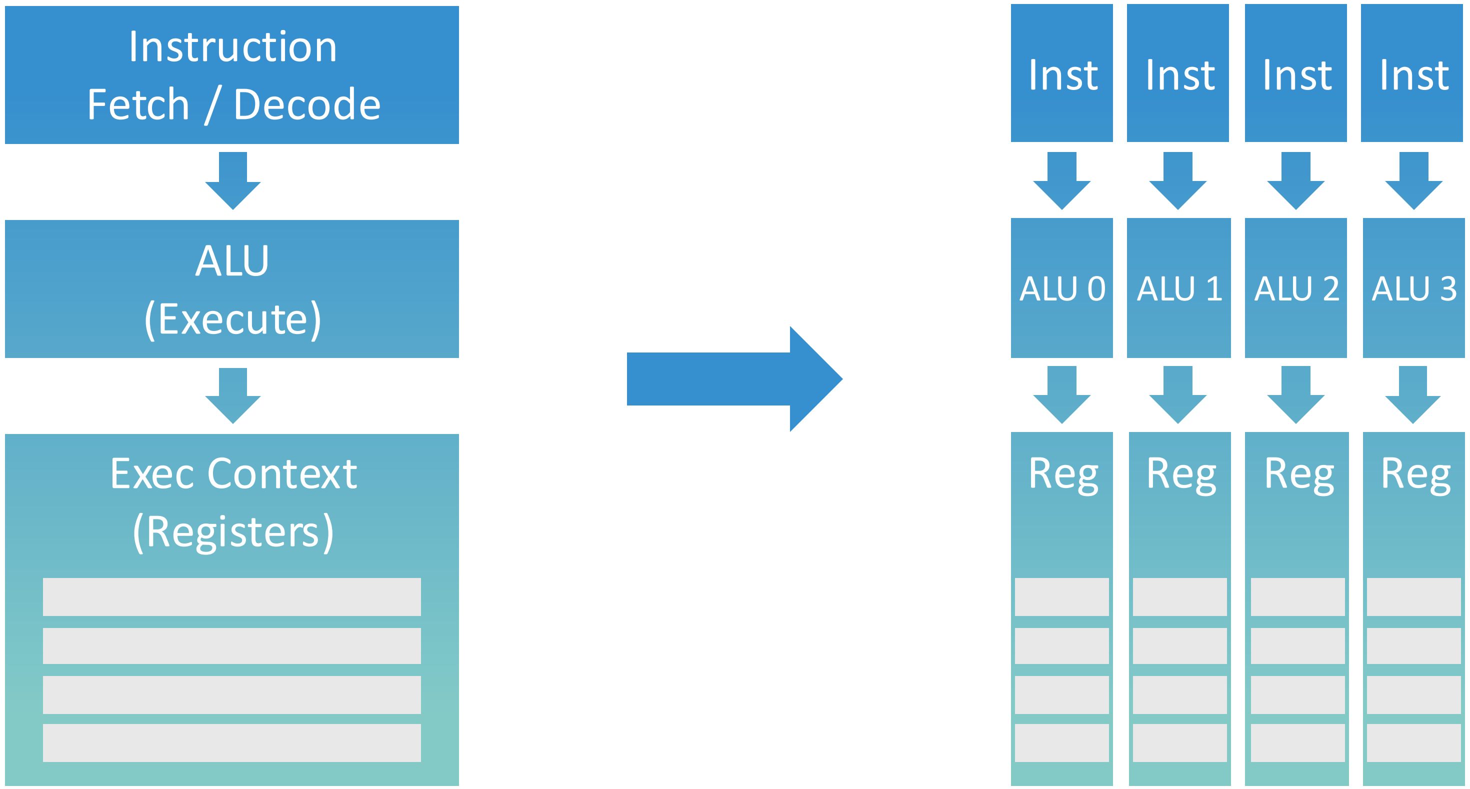
SIMT: Single Instruction, Multiple Thread

- One of the **subcategories** of SIMD
- Each ALU has its own separate register file
- Usually used in modern **GPGPU**



Modern single instruction, multiple thread GPUs

MIMD: Multiple Instruction, Multiple Data

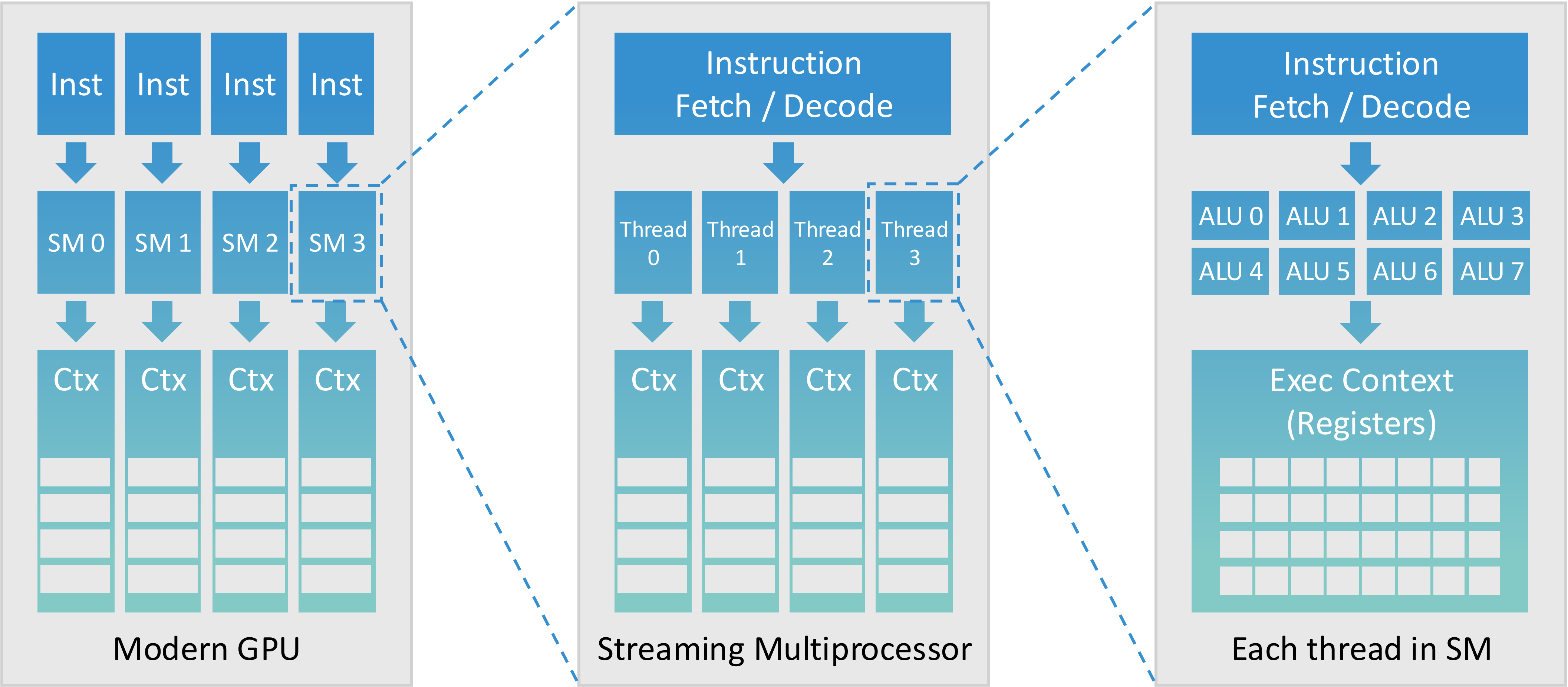


Conventional single instruction, single data processor

Modern multiple instruction, multiple data processor

Massive Parallel Computing Units in GPGPU

上海创智学院 Shanghai Innovation Institute



MIMD multi-Core processor

SIMT streaming multiprocessor

SIMD thread

02

GPU Architectures

H100 Architecture with Tensor Cores

上海创智学院 Shanghai Innovation Institute

One SM

GH100 Full GPU with 144 SMs, while H100 SMX has only 132 SMs

H100 Streaming Multiprocessor

上海创智学院
Shanghai Innovation Institute

Tensor Memory Accelerator (TMA), highly efficient, asynchronous, and bi-directional transfer of multi-dimensional tensors between global memory and shared memory

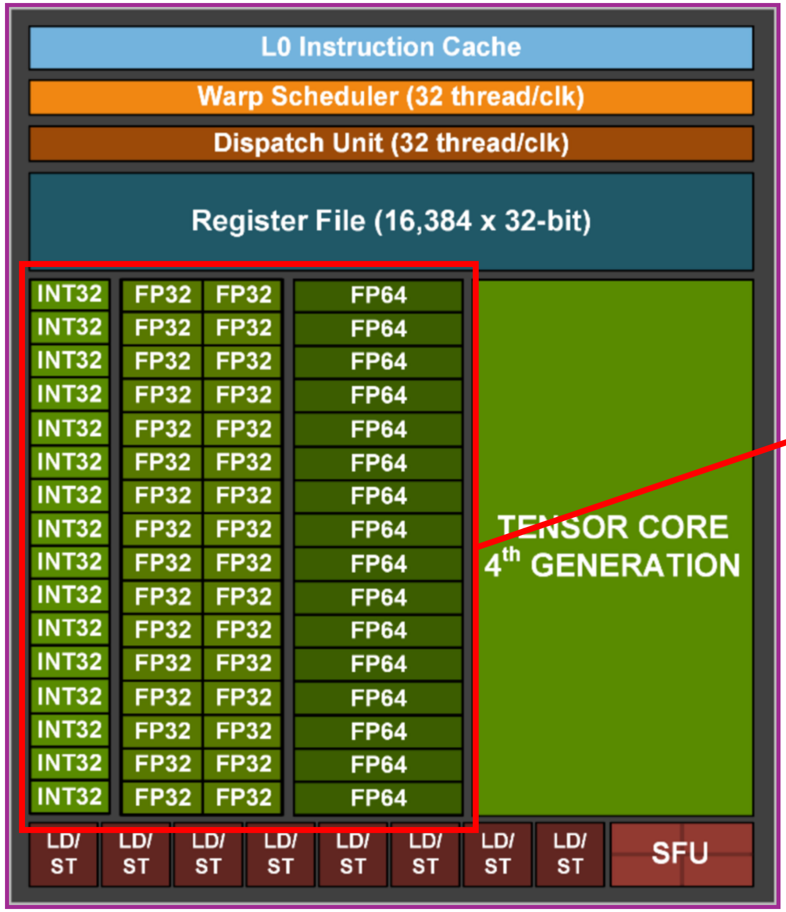
Shared Memory / L1 Data Cache

H100 Streaming Multiprocessor

上 海 创 智 学 院
Shanghai Innovation Institute

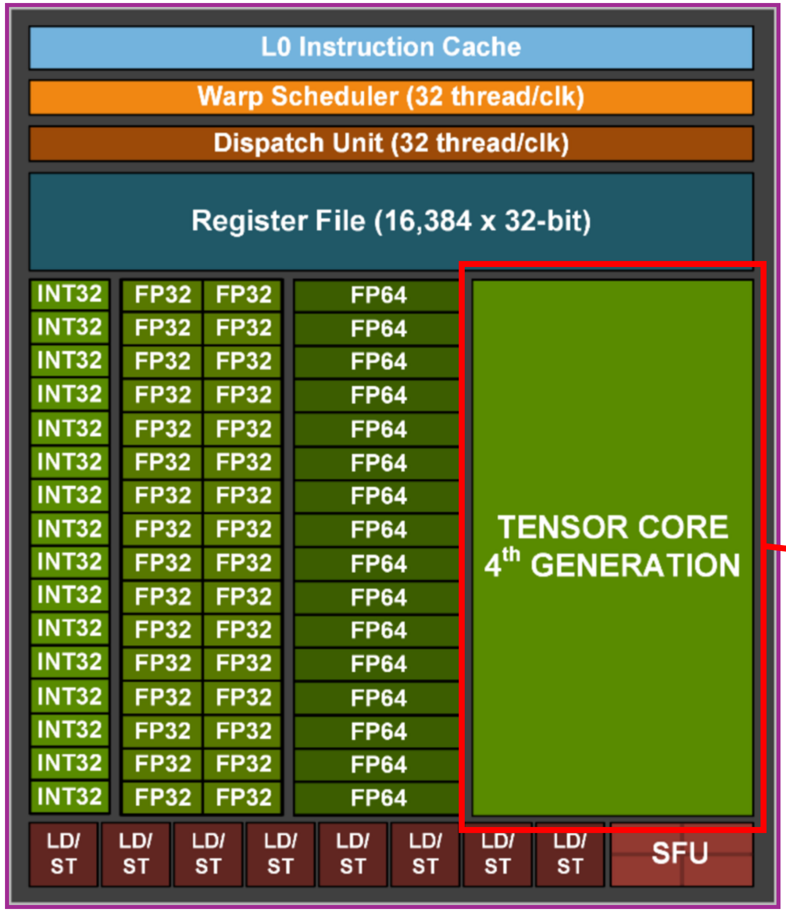
H100 Streaming Multiprocessor

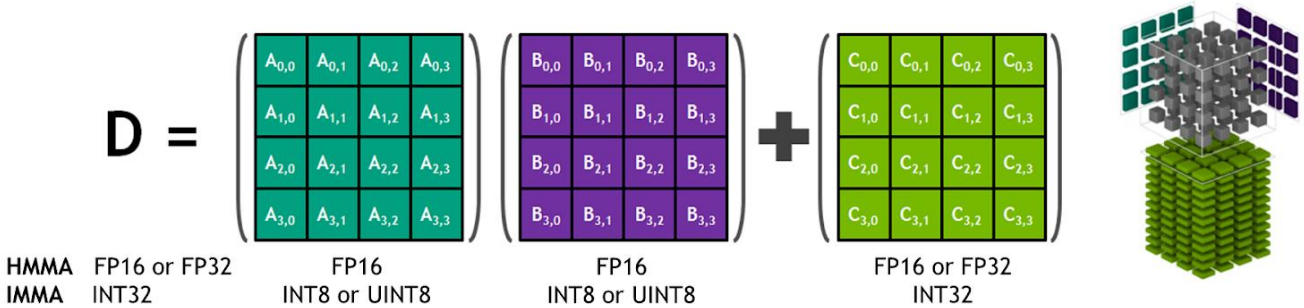
上海创智学院 Shanghai Innovation Institute



CUDA Cores (Scalar ALUs) managed by 32 CUDA threads

H100 Streaming Multiprocessor





Tensor Cores, used for matrix multiplication

03

CUDA Programming Abstractions

Basic CUDA syntax

上海创智学院 Shanghai Innovation Institute

Host program: running as part of normal C/C++ application on CPU

Host

```
const int Nx = 12;
const int Ny = 6;
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each

matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Bulk launch of many CUDA threads
"launch a grid of CUDA thread blocks"
Call returns when all threads have terminated

Device

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

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

__global__ denotes a CUDA kernel runs on GPU

Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

CUDA kernel: executed in parallel on multiple SMs

Clear Separation of Host and Device Code

上海创智学院 Shanghai Innovation Institute

Host

```
const int Nx = 12;
const int Ny = 6;
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each

matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Device

```
__device__ float doubleValue(float x)
{
  return 2 * x;
}
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

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

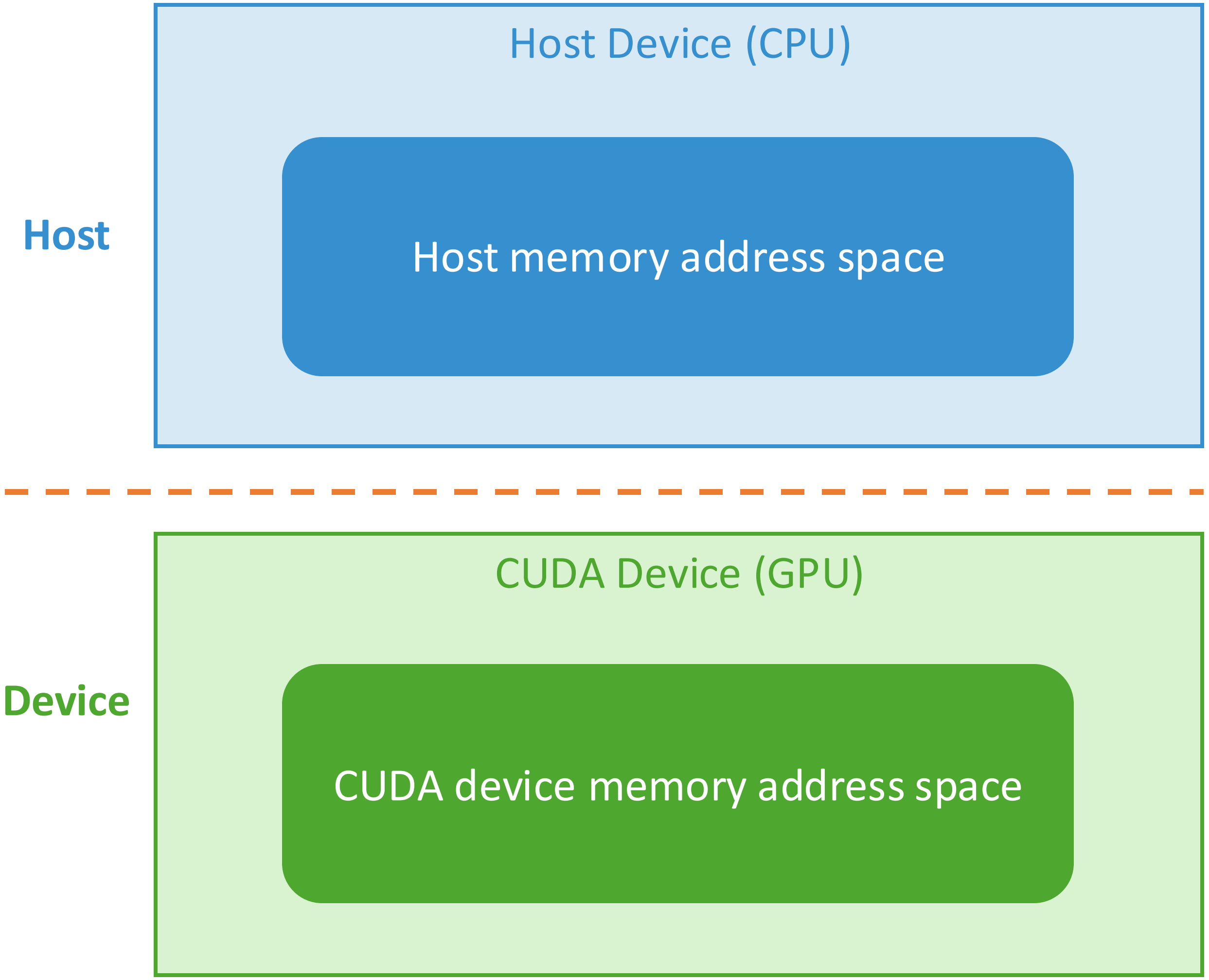
Separation of execution into host and device code is performed statically by the programmer

Function without any attribute runs on **Host** as common C++ program

__global__ denotes a CUDA kernel runs on GPU

__device__ denotes a CUDA function that can be called from **device** or **global** function

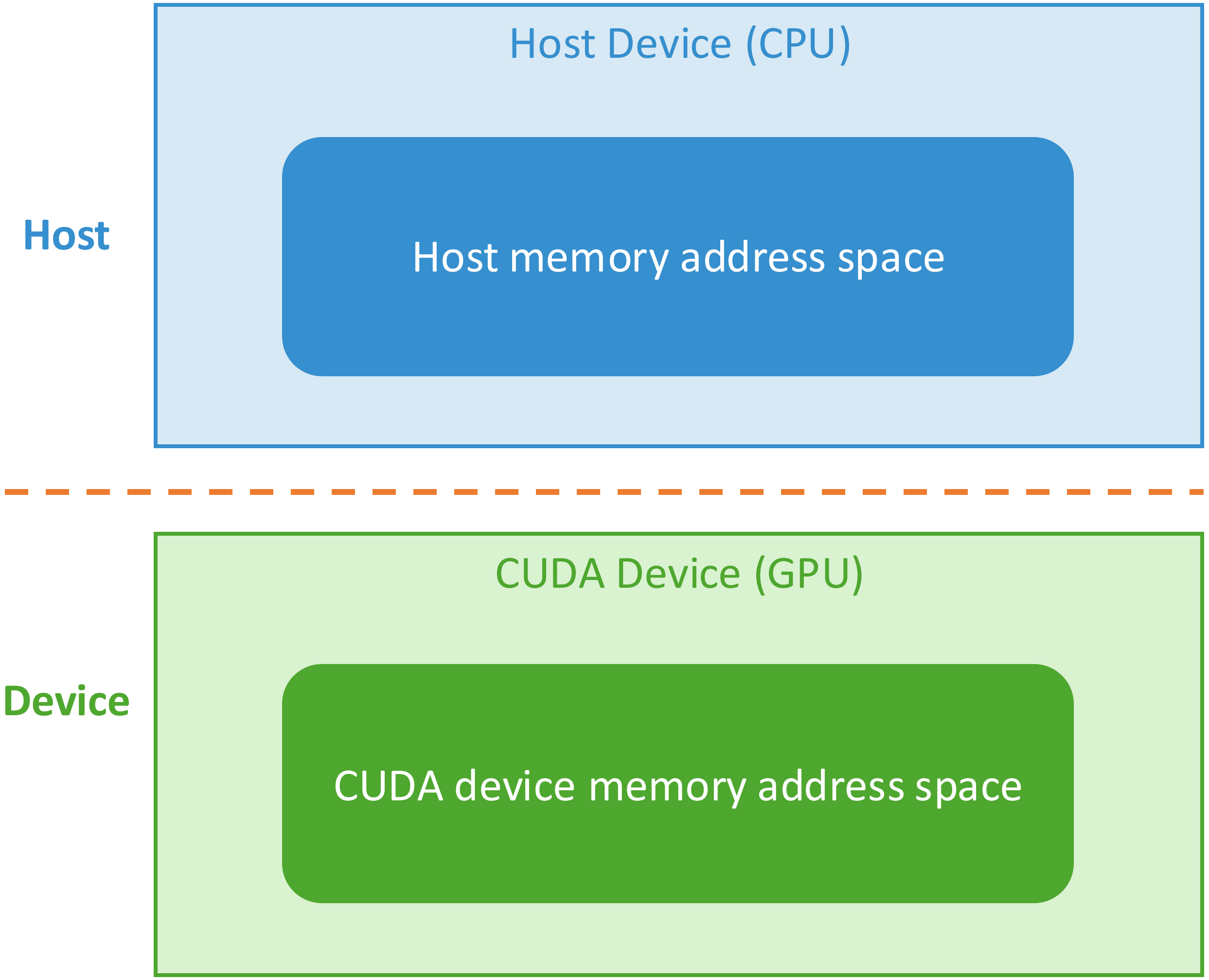
CUDA Memory Model



Distinct host and device address spaces:

- Cannot access host memory from device
- Cannot access device memory from host

Case Study: Minimal Example

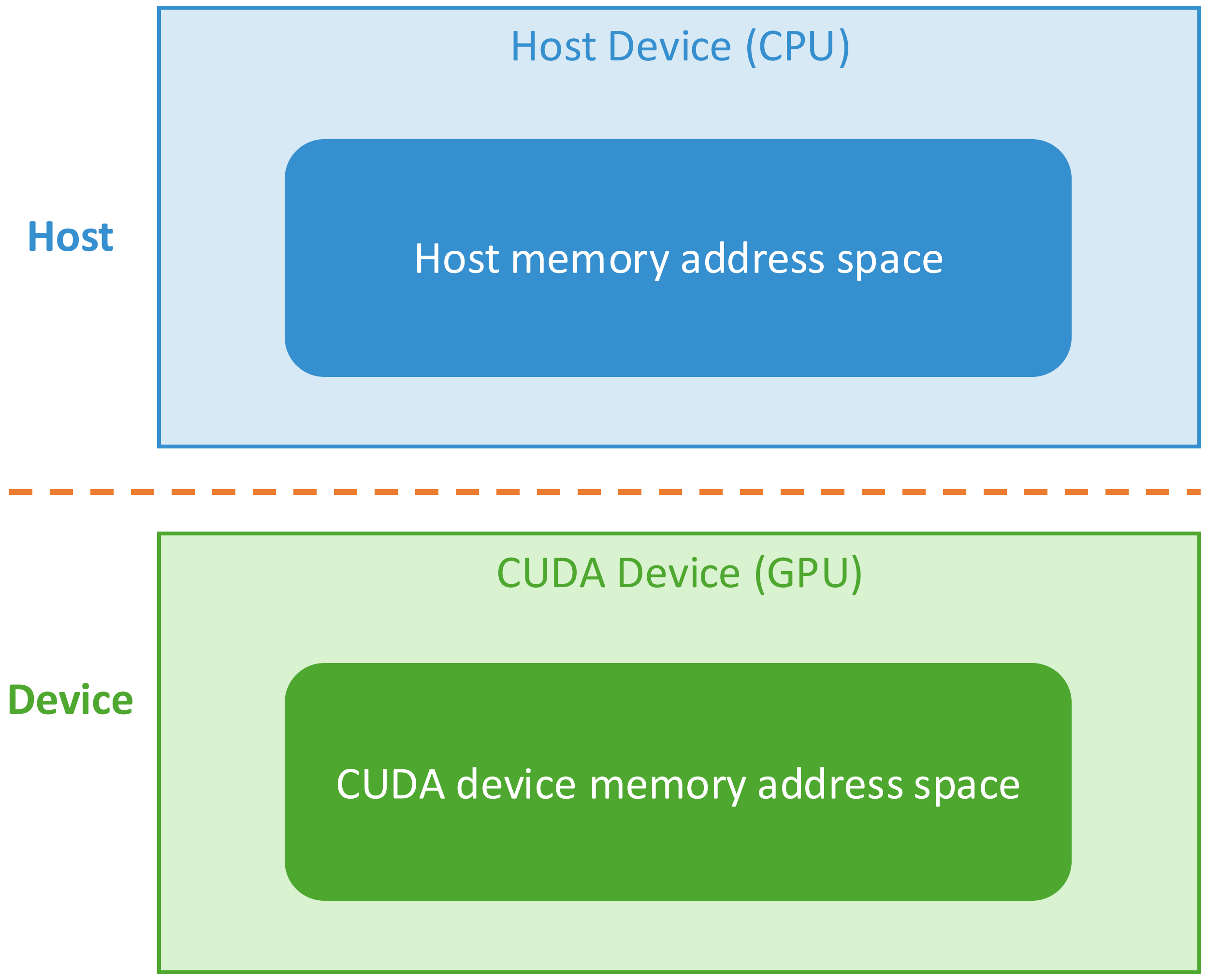



```
float* A = new float[N];

for (int i=0 i<N; i++)
  A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;
cudaMalloc(&deviceA, bytes);

cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```

Init data on host device

cudaMalloc: Allocate Memory Space on Device

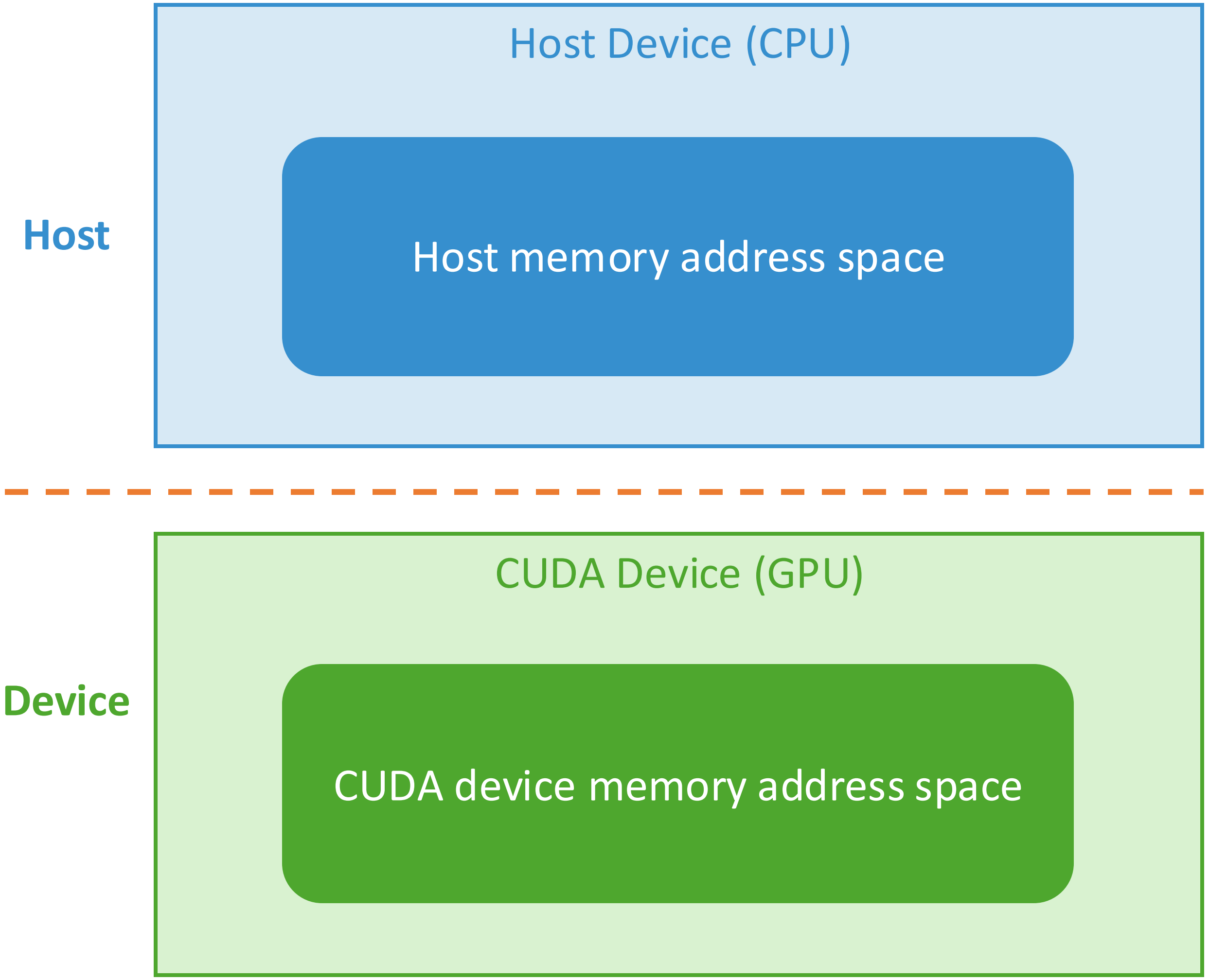
上海创智学院 Shanghai Innovation Institute




```
float* A = new float[N];

for (int i=0 i<N; i++)
  A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;
cudaMalloc(&deviceA, bytes);

cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```

Allocate memory with specific size on device, and store the pointer to `deviceA`

cudaMemcpy: Move Data Between Host and Device




```
float* A = new float[N];

for (int i=0 i<N; i++)
  A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;
cudaMalloc(&deviceA, bytes);

cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```

Copy data from host to device

NOTE:

1. deviceA[i] is an invalid operation on host side
2. cudaMemcpy also support copy from device to host

Basic CUDA Syntax

上海创智学院 Shanghai Innovation Institute

Host program: running as part of normal C/C++ application on CPU

Host

```
const int Nx = 12;
const int Ny = 6;
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each

matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Bulk launch of many CUDA threads
"launch a grid of CUDA thread blocks"
Call returns when all threads have terminated

Device

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

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

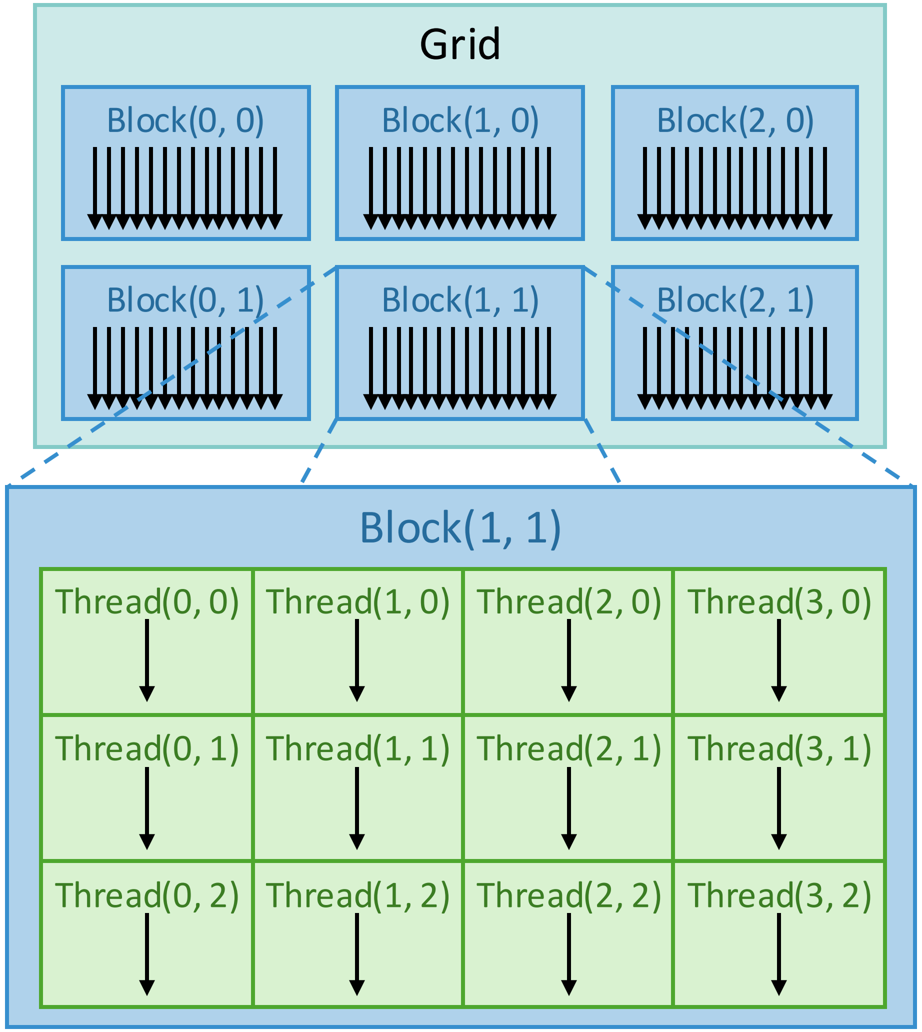
__global__ denotes a CUDA kernel runs on GPU

Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

CUDA kernel: executed in parallel on multiple SMs

CUDA Programs Consist of a Hierarchy of Threads

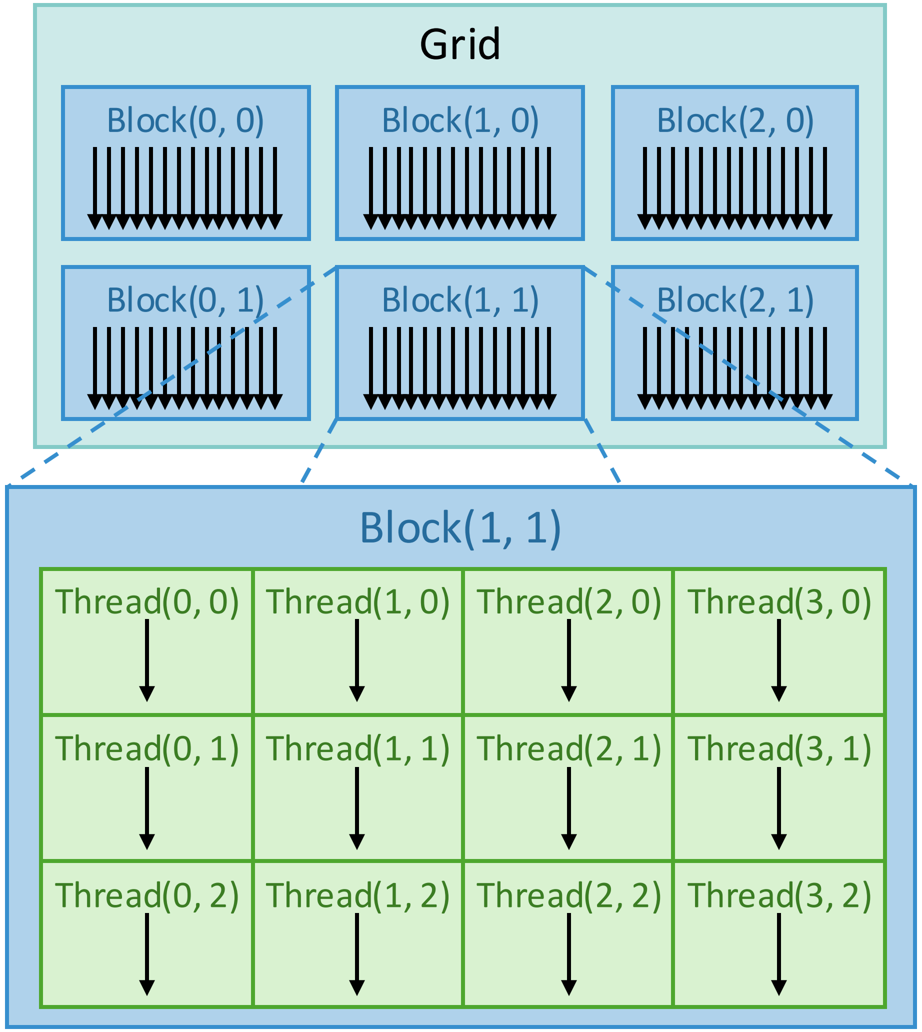
`threadIdx` and `blockIdx` are up to 3-dimensional (2D example below)

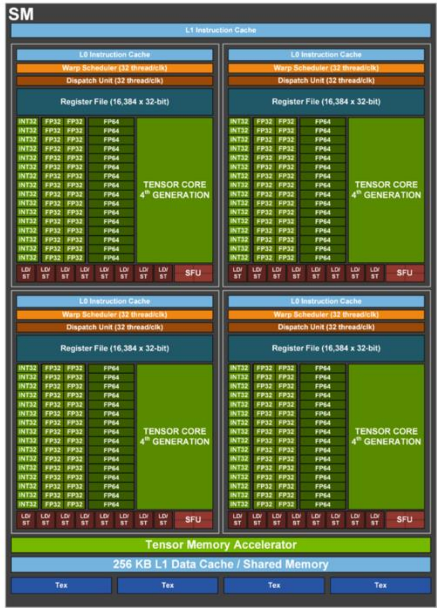



```
const int Nx = 12;
const int Ny = 6;
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each

matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA Blocks Map to GPU SM

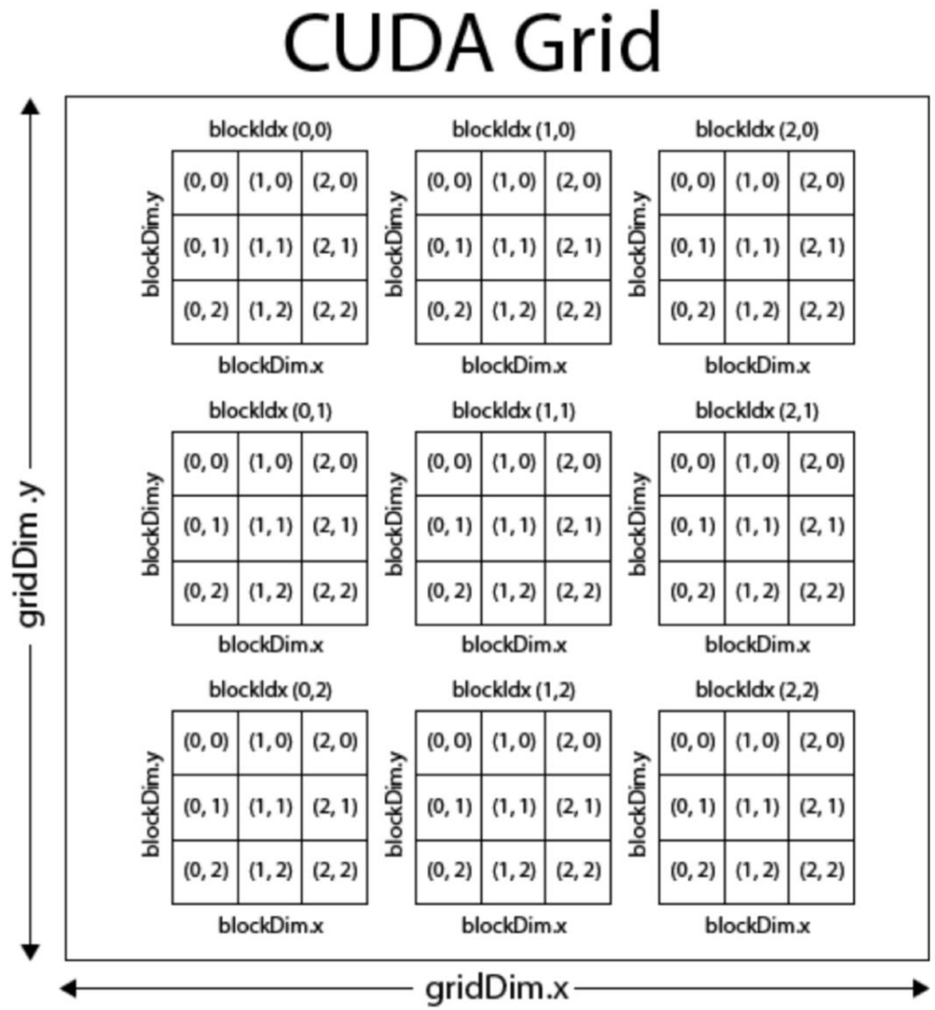
The whole CUDA program runs on whole GPU, while a block runs on a single SM





Grid, Block, and Thread

- gridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of a block
- threadIdx: The thread index within a block



SIMT: Divergent Execution Overhead

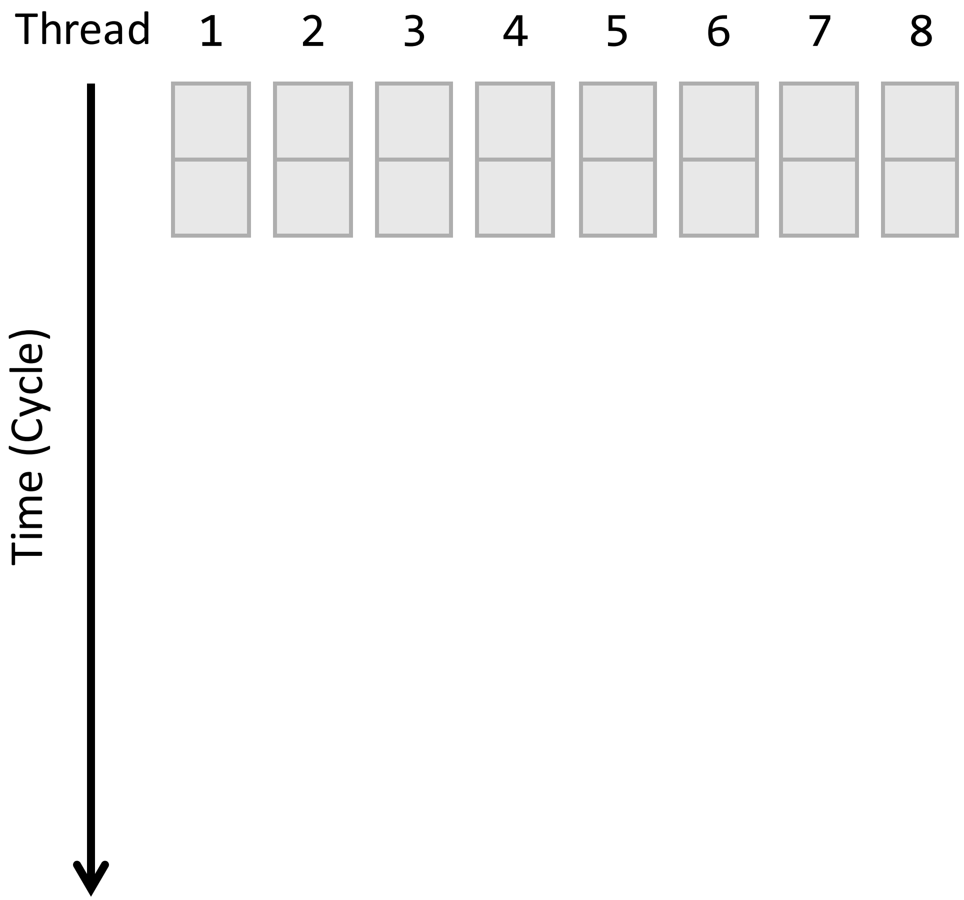
Thread 1 2 3 4 5 6 7 8

Time (Cycle)

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

Kernel function

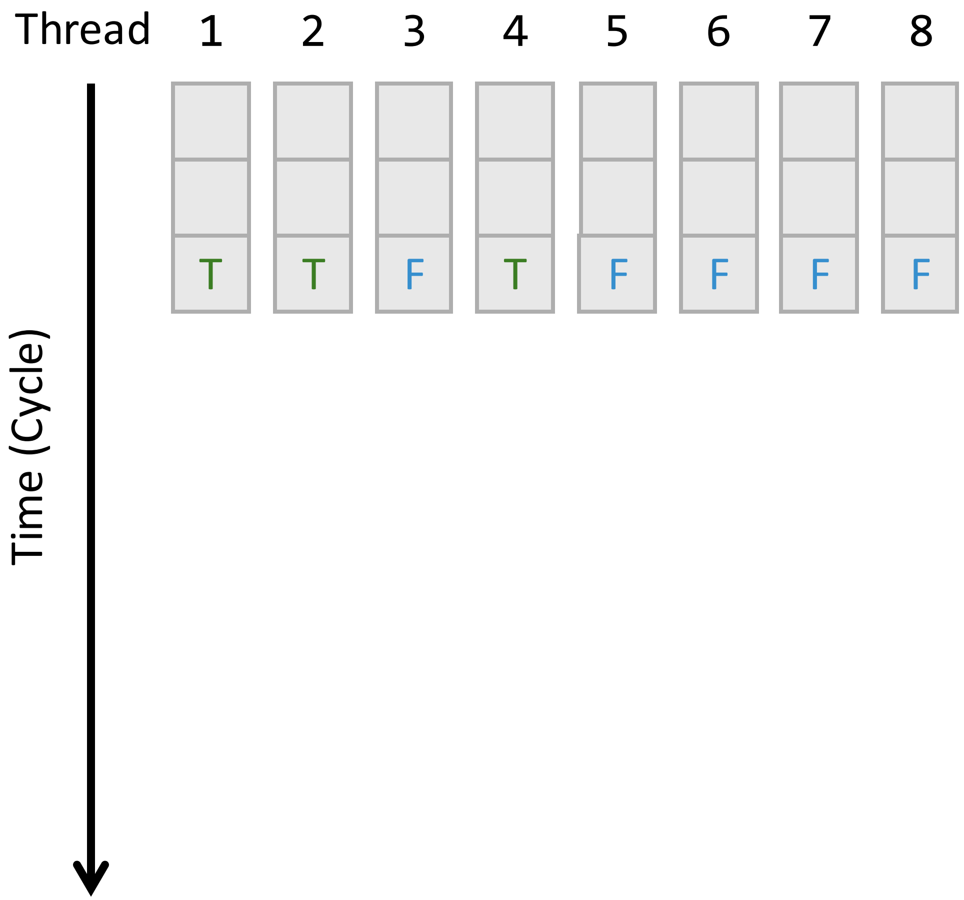
SIMT: Divergent Execution Overhead




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

Kernel function

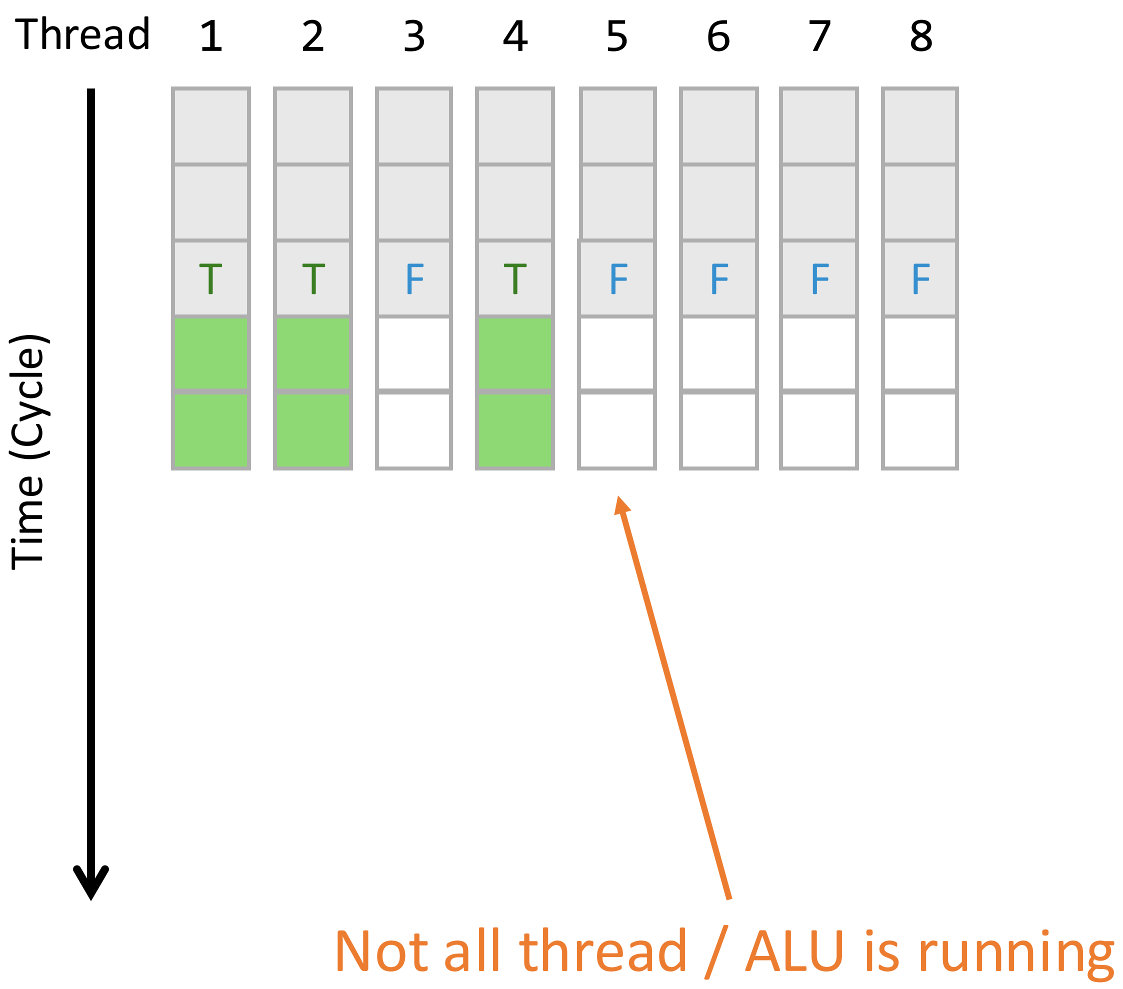
SIMT: Divergent Execution Overhead




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

Kernel function

SIMT: Divergent Execution Overhead

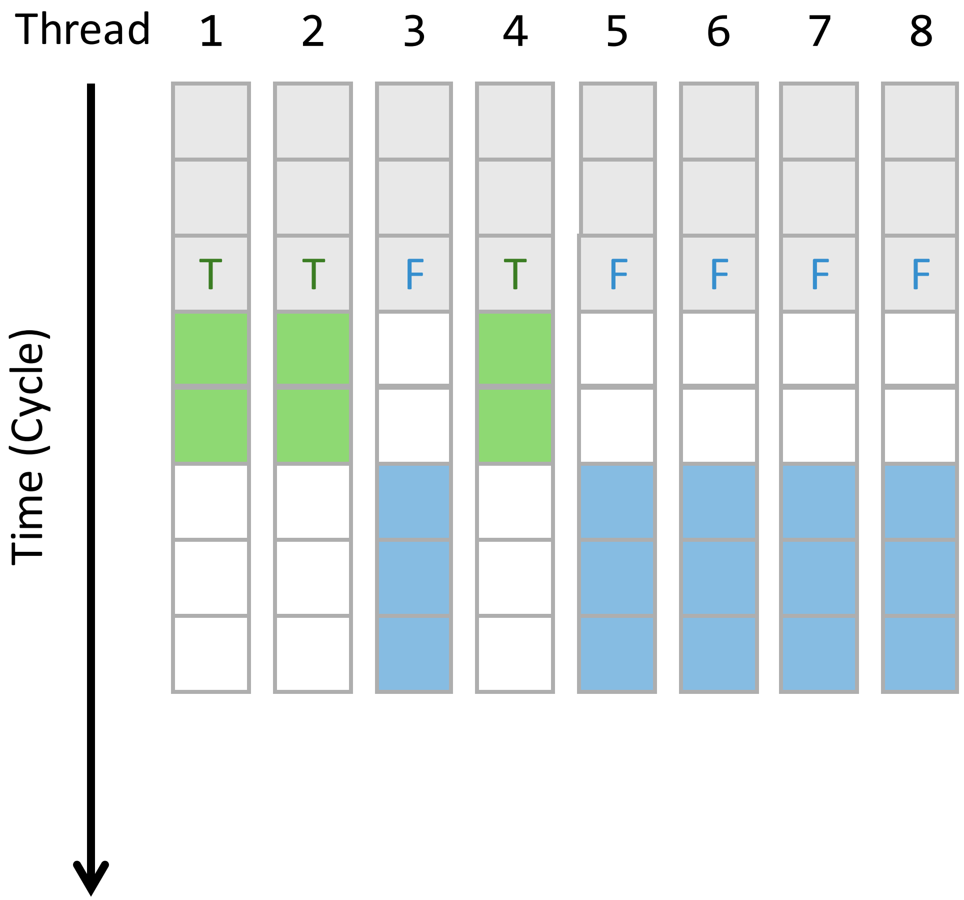



```
__global__ void f(float A[N]) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
⇒   x = 2.0f * x;
  } else {
    x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

Kernel function

SIMT: Divergent Execution Overhead

上海创智学院 Shanghai Innovation Institute

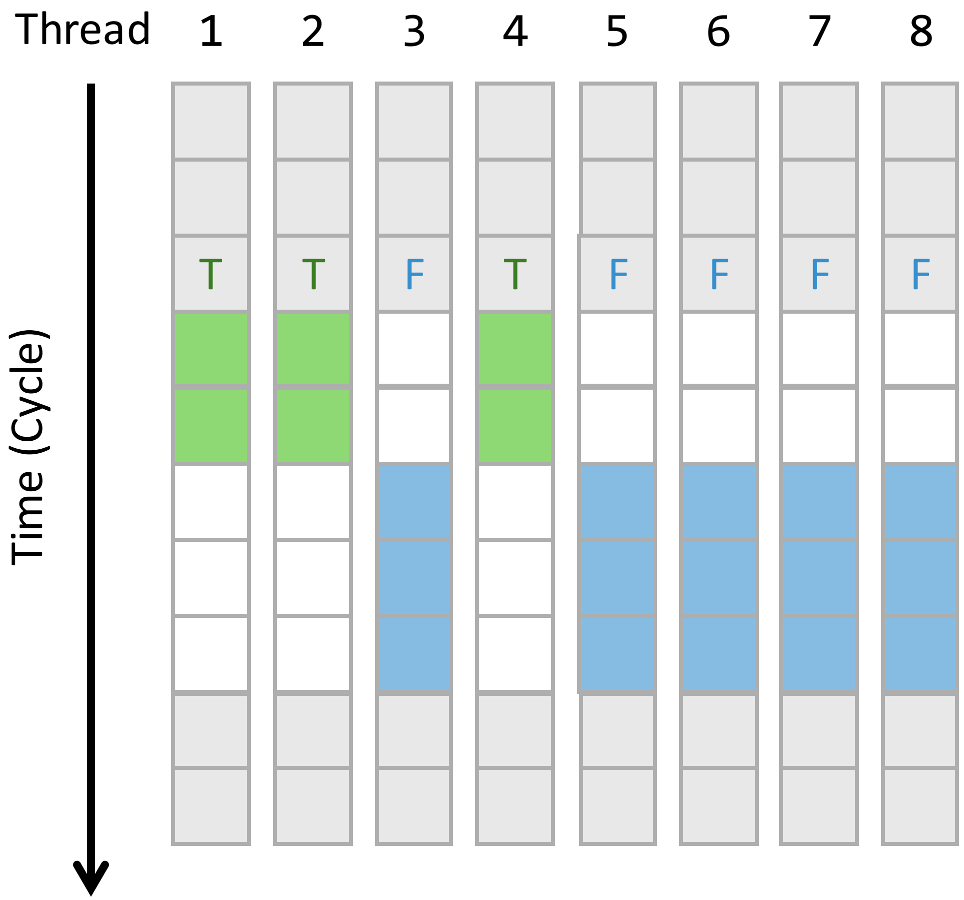



```
__global__ void f(float A[N]) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
    x = 2.0f * x;
  } else {
    x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

Kernel function

SIMT: Divergent Execution Overhead

上海创智学院 Shanghai Innovation Institute




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

Kernel function

Terminology

Coherence execution

- Same instruction sequence applies to all elements
- Necessary for efficient use of GPUs

Divergent execution

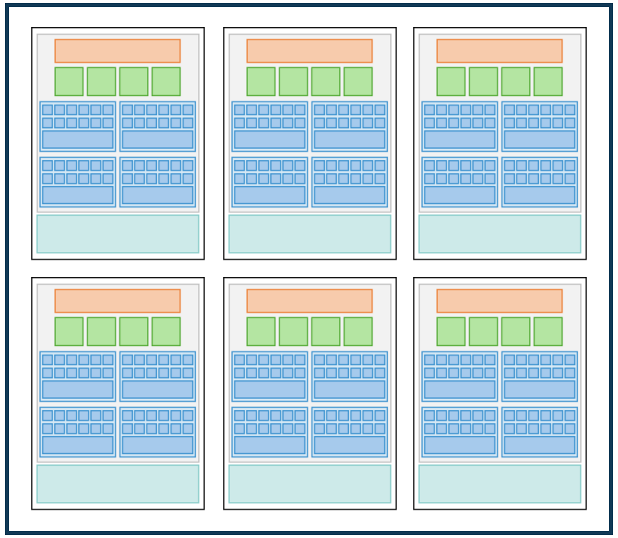
- A lack of coherence execution
- Should be minimized in CUDA programs

04

CUDA Program Execution

CUDA Compilation

- Goal: run the same CUDA program on various GPUs



Mid-range GPU (6 cores)

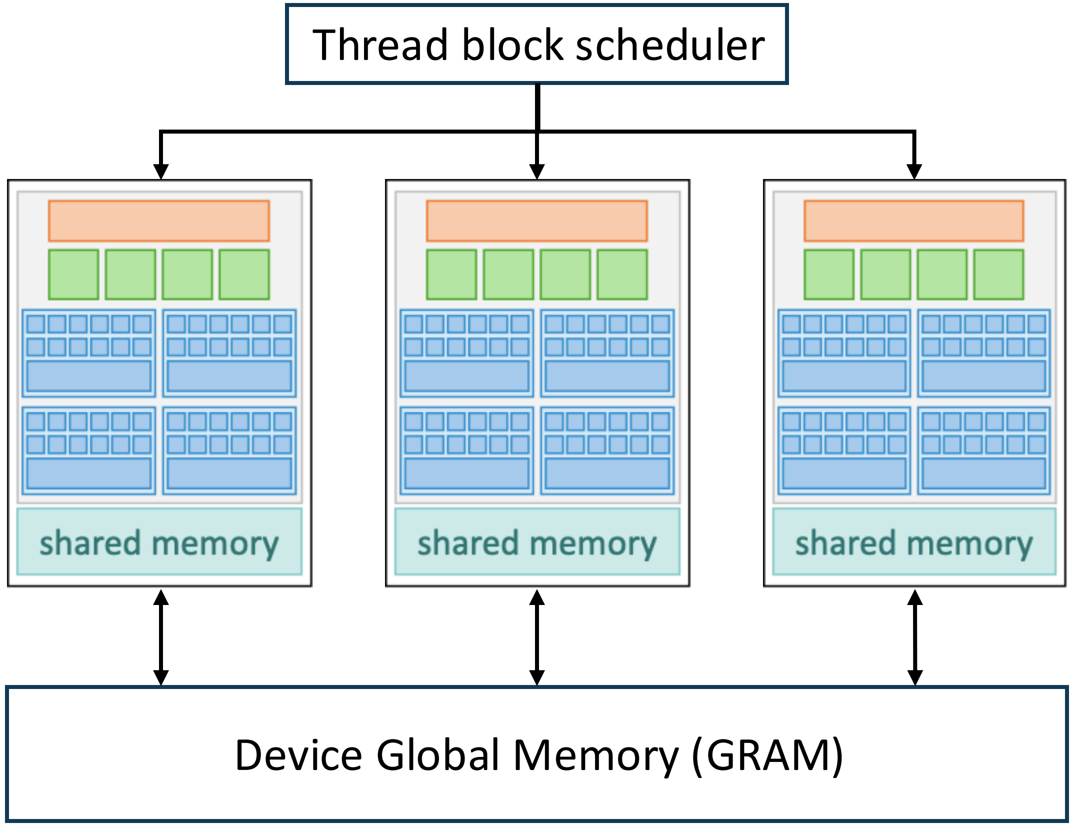
High-end GPU (12 cores)

CUDA Thread Block Scheduling

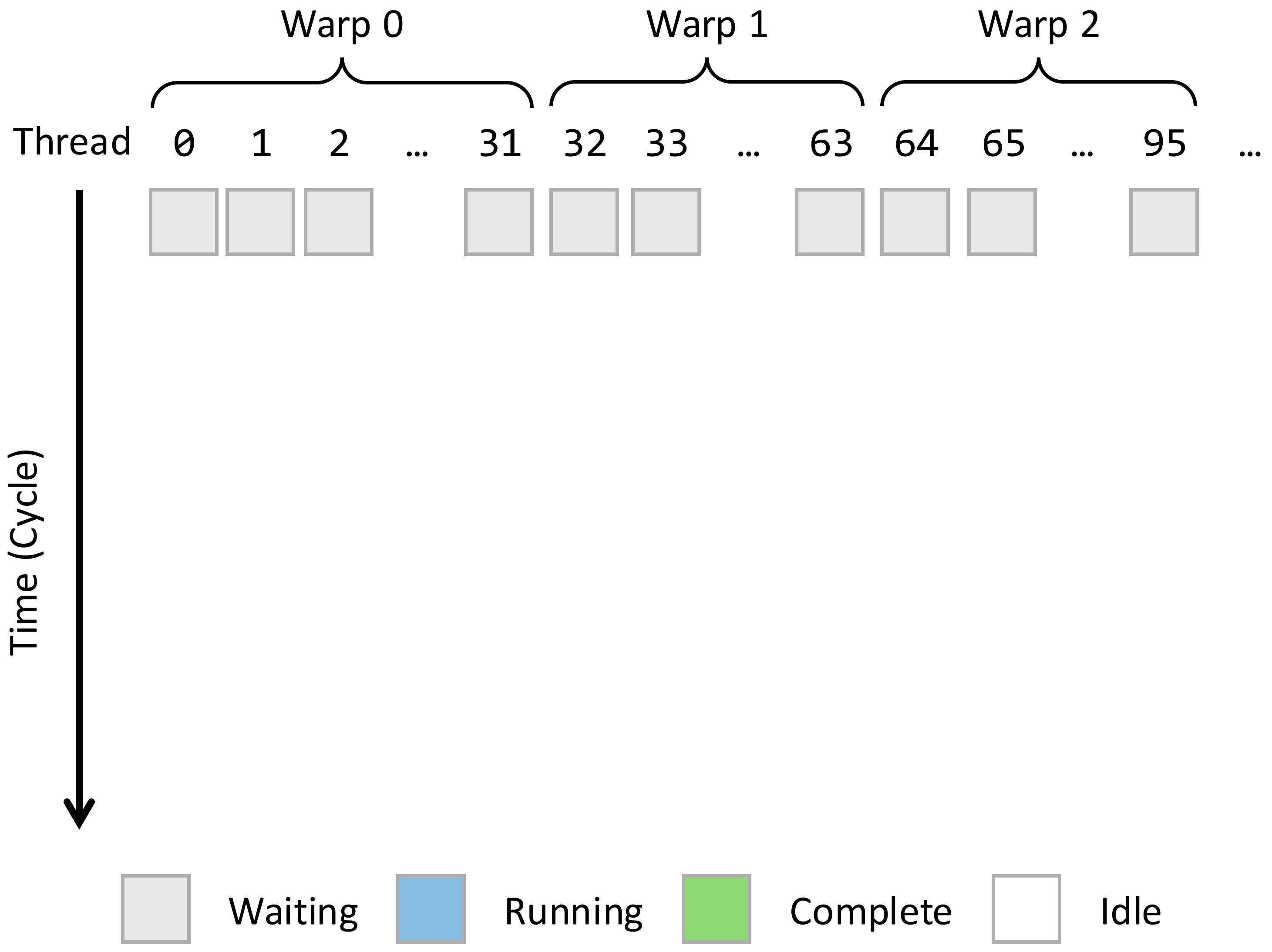
上海创智学院 Shanghai Innovation Institute

- **Major CUDA assumption:** thread blocks can be executed in any order (no dependencies between thread blocks)

- GPU maps thread blocks to cores using a dynamic scheduling policy that respects resource requirements.

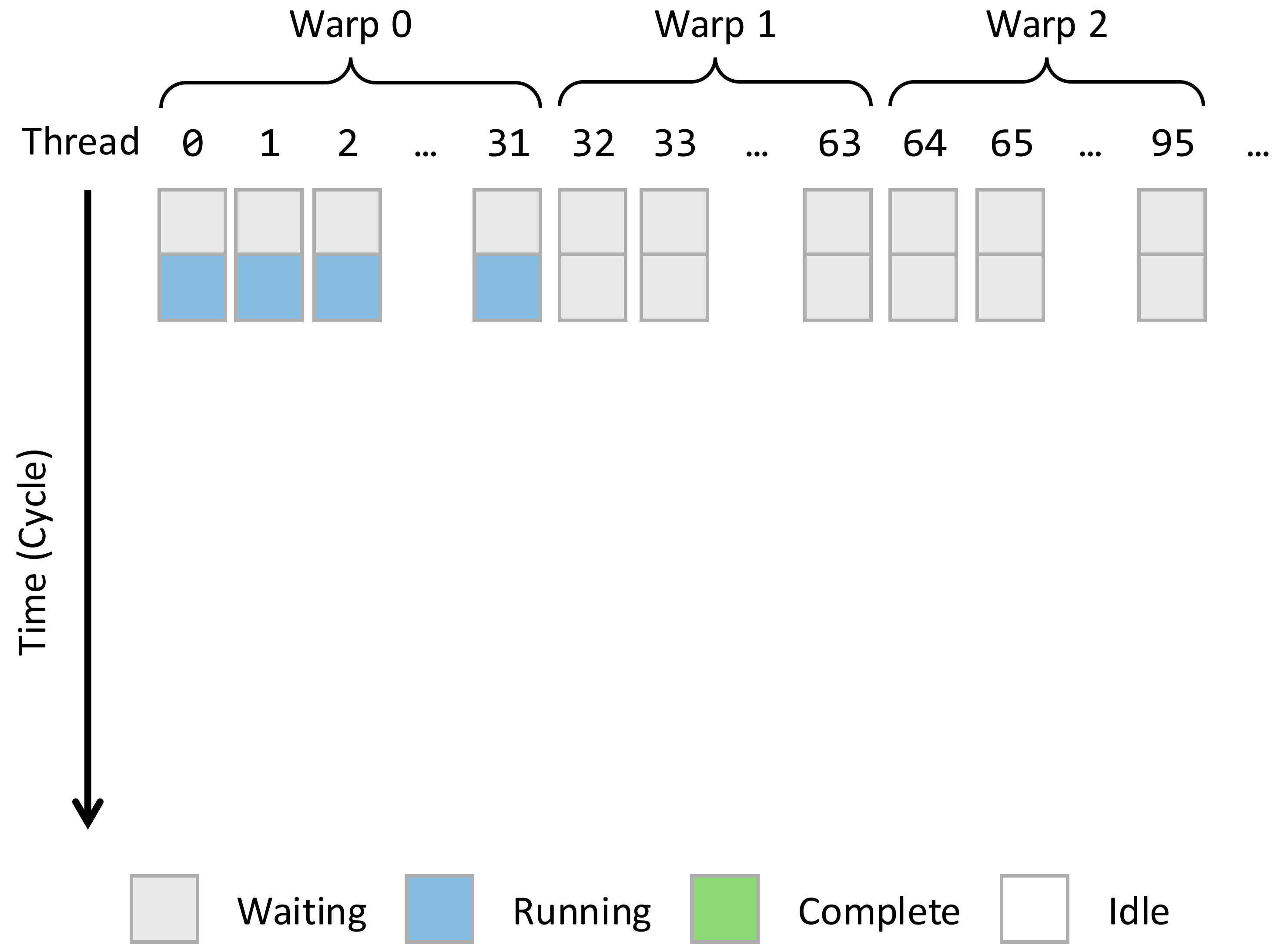


CUDA Thread Scheduling



Warp: Groups of 32 CUDA threads in a thread block are executed simultaneously

CUDA Thread Scheduling



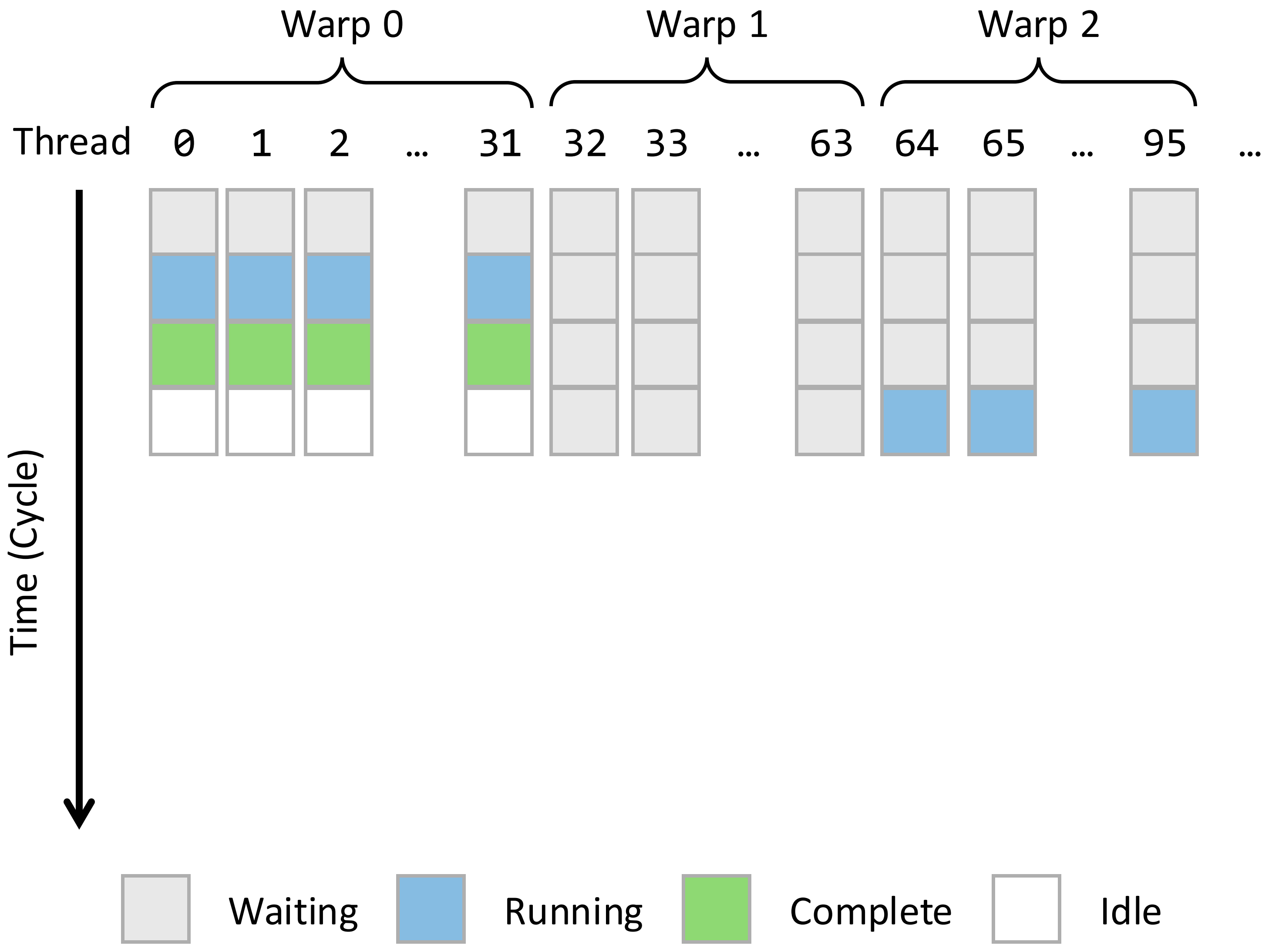
At most 32 threads can run simultaneously, while other threads need to wait until running threads finished

CUDA Thread Scheduling

The previous threads are all finished, now GPU can schedule another warp

CUDA Thread Scheduling

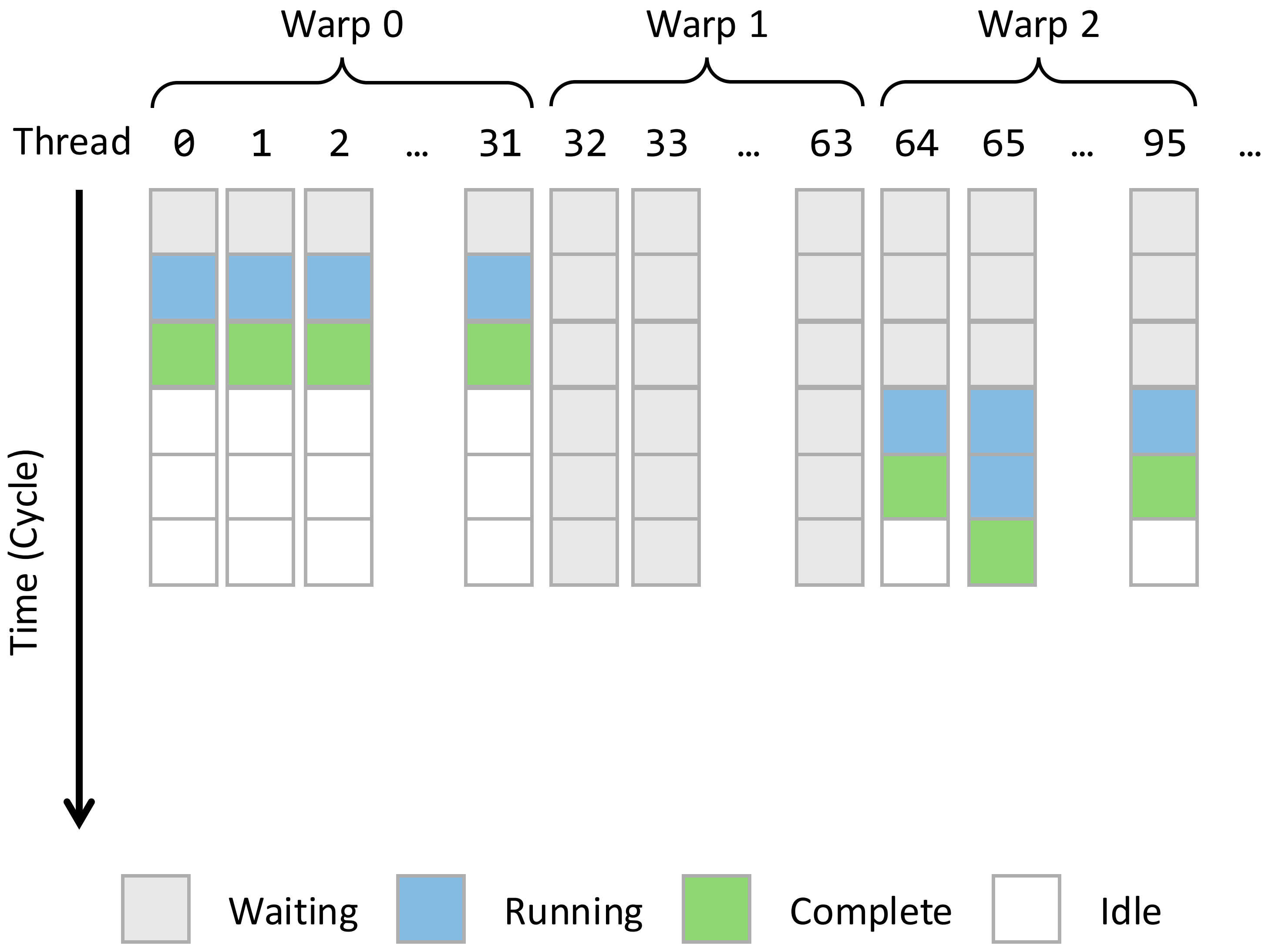
上海创智学院
Shanghai Innovation Institute



Like thread block scheduling, thread inside a block can be executed in any order.

The warp scheduler can pick any waiting warps. Here it pick warp 2

CUDA Thread Scheduling

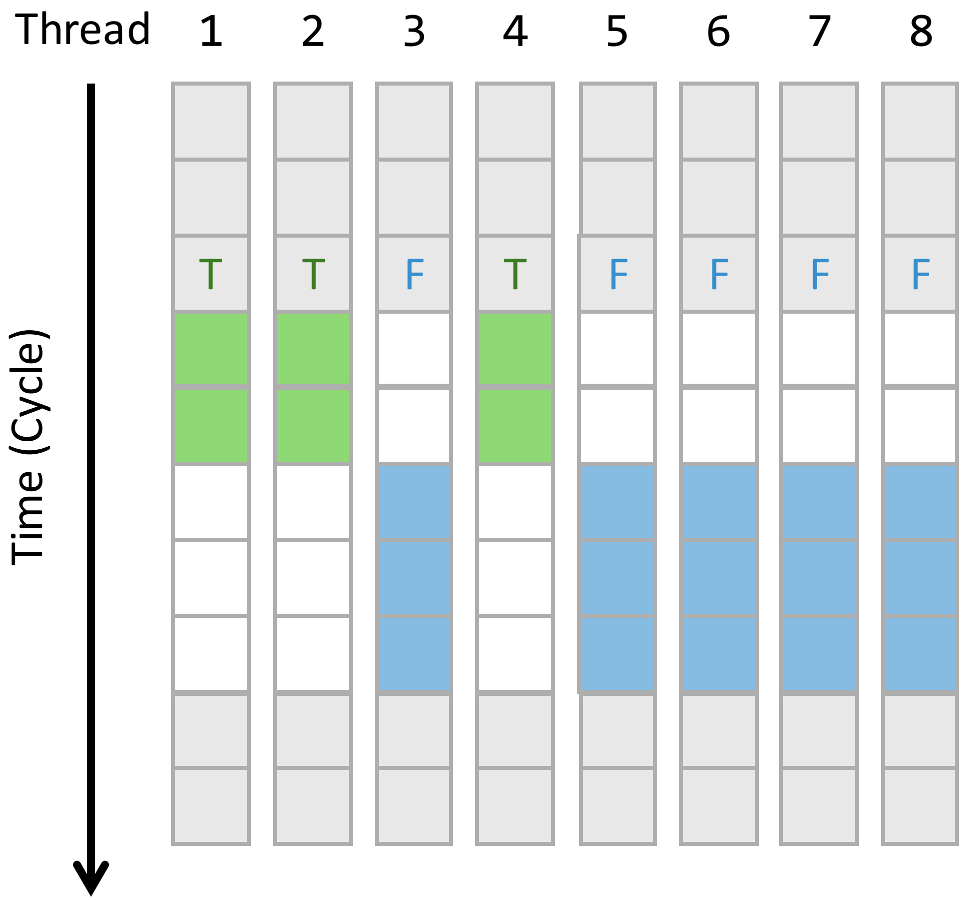


A warp is complete if and only if all 32 threads are complete.

CUDA Thread Scheduling

Recap: Divergent Execution Overhead

上海创智学院 Shanghai Innovation Institute



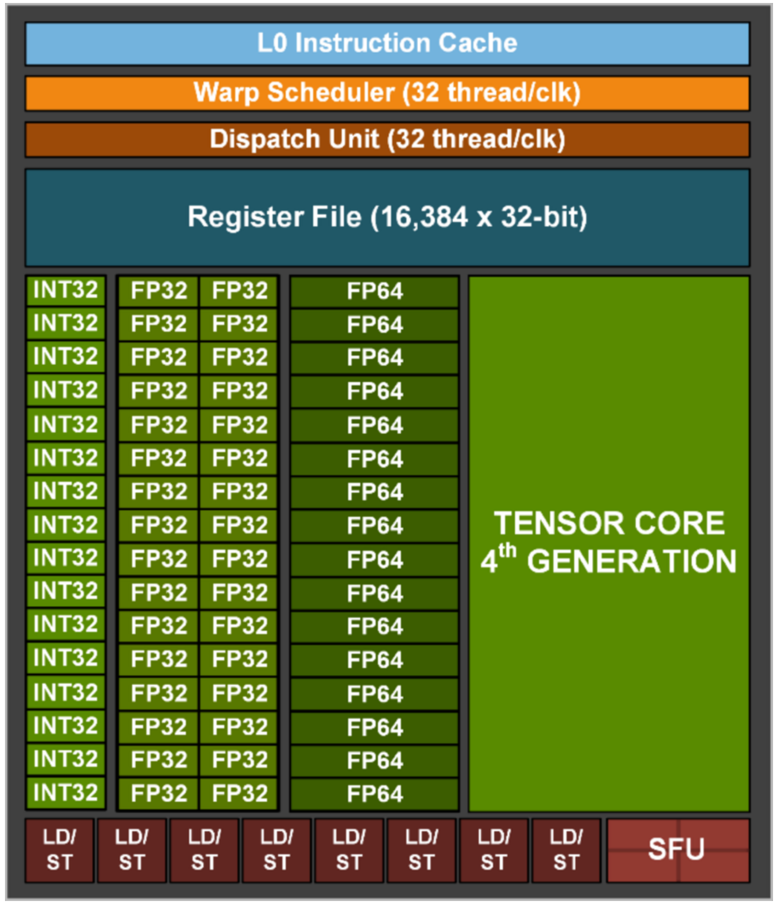

```
__global__ void f(float A[N]) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float x = A[i];
  if (x > 0) {
    x = 2.0f * x;
  } else {
    x = exp(x, 5.0f);
  }
  A[i] = x;
}
```

Kernel function

The divergent overhead only happens **inside a warp**, but not across all threads

More Complicated Scheduling in Modern GPUs

上 海 创 智 学 院
Shanghai Innovation Institute



Different kinds on ALUs support
out-of-order execution

4 **Warp** can be running on a SM at the same time

Acknowledgement

上海创智学院
Shanghai Innovation Institute

The development of this course, including its structure, content, and accompanying presentation slides, has been significantly influenced and inspired by the excellent work of instructors and institutions who have shared their materials openly. We wish to extend our sincere acknowledgement and gratitude to the following courses, which served as invaluable references and a source of pedagogical inspiration:

- Machine Learning Systems[15-442/15-642], by **Tianqi Chen** and **Zhihao Jia** at **CMU**.
- Advanced Topics in Machine Learning (Systems)[CS6216], by **Yao Lu** at **NUS**

While these materials provided a foundational blueprint and a wealth of insightful examples, all content herein has been adapted, modified, and curated to meet the specific learning objectives of our curriculum. Any errors, omissions, or shortcomings found in these course materials are entirely our own responsibility. We are profoundly grateful for the contributions of the educators listed above, whose dedication to teaching and knowledge-sharing has made the creation of this course possible.

System for Artificial Intelligence

Thanks

Siyuan Feng
Shanghai Innovation Institute