Embedded Intelligent System and Novel Computer Architecture

Lecture 03(c) – Understanding Modern Processor: GPGPU Arch & Massively Parallel Programming

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

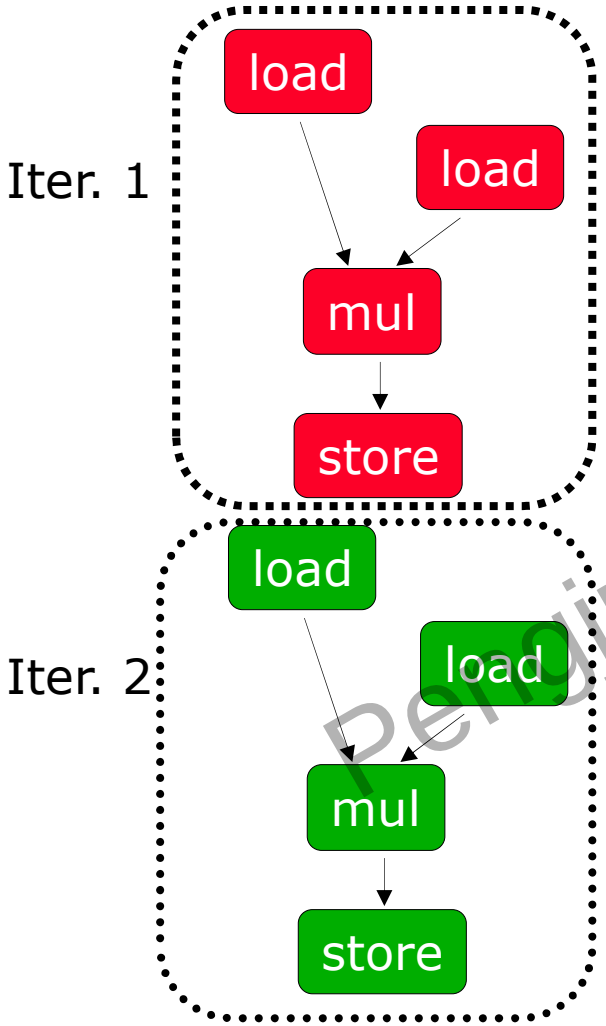
http://gr.xjtu.edu.cn/web/pengjuren

Contents

- **Programming Model and Interface**
- Hardware Implementation
- Optimization through S/H Cooperation

How can you exploit Parallelism ?

Scalar Sequential Code



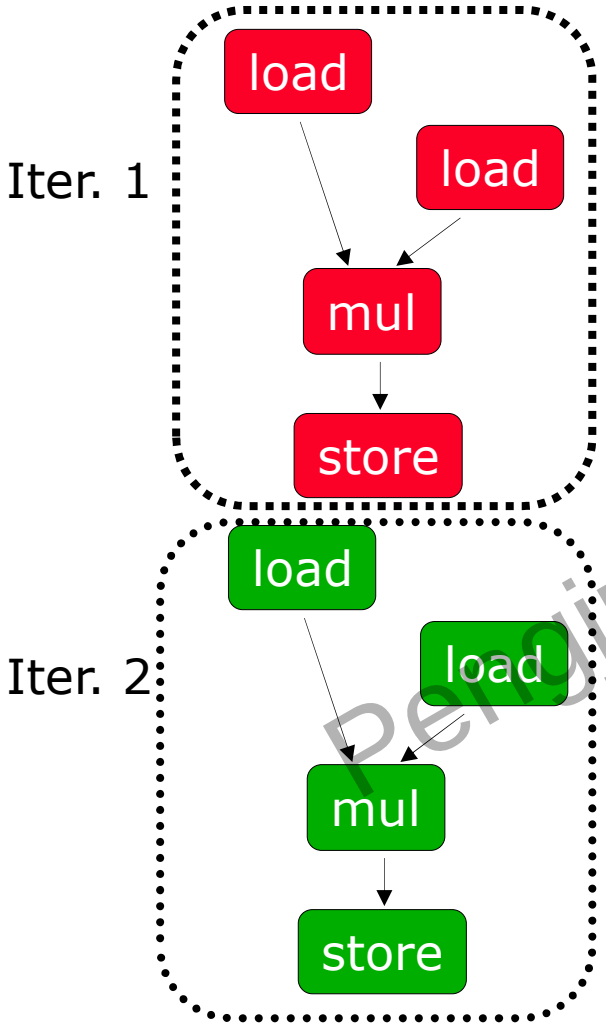

```
for (i=0; i < N; i++)
    A[i] = B[i] * C[i];
```

Let's examine three programming options to exploit parallelism present in this sequential code:

1. **Sequential (SISD)**
2. **Data-Parallel (SIMD)**
3. **Multithreaded (MIMD/SPMD)**

Prog. Model 1: Sequential (SISD)

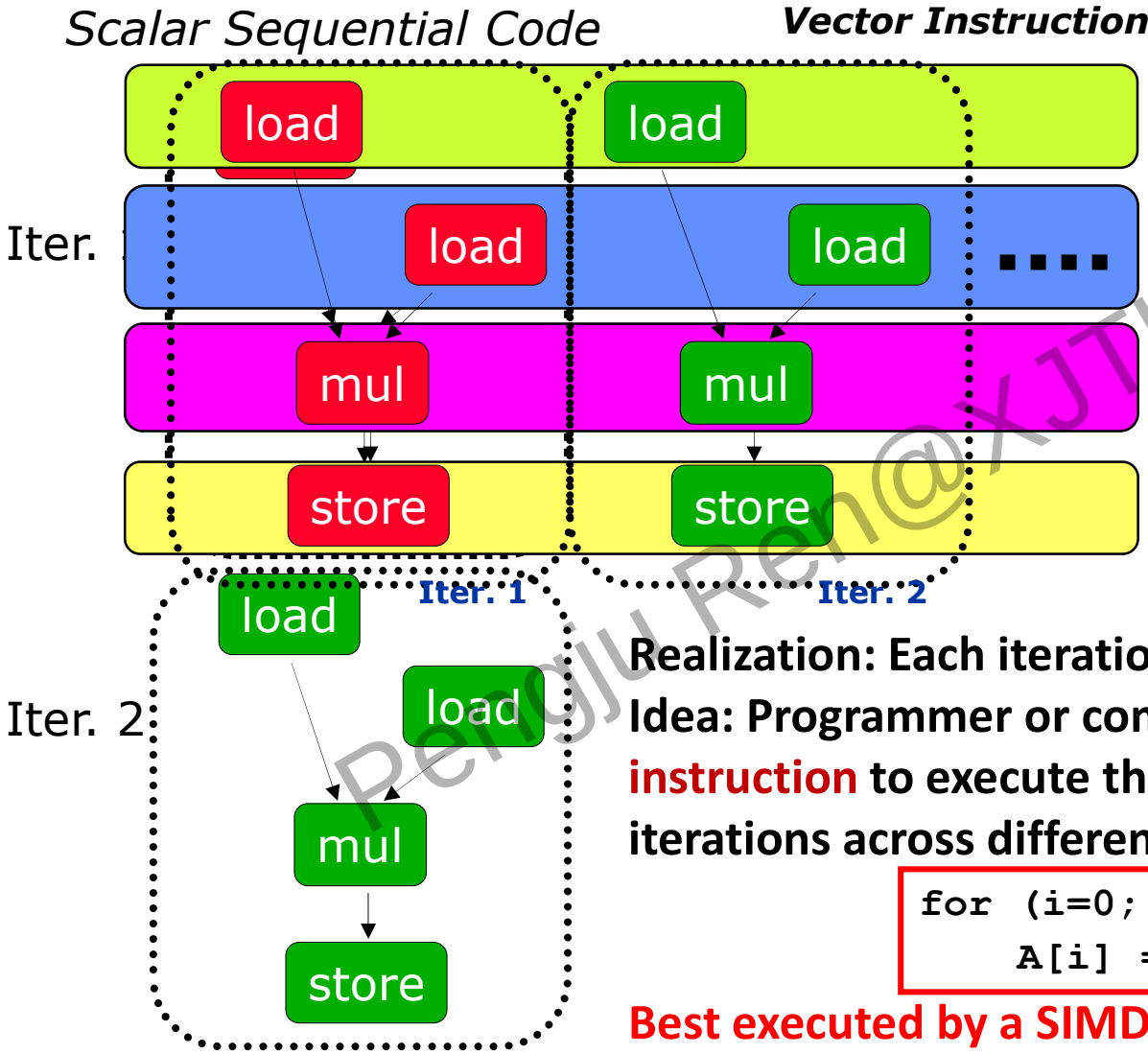
Scalar Sequential Code




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

- **Can be executed on a Pipelined processor**
- **Out-of-order execution processor**
  - **Independent instructions executed when ready**
  - **Different iterations are present in the instruction window and can execute in parallel in multiple functional units**
  - **In other words, the loop is dynamically unrolled by the hardware**
- **Superscalar or VLIW processor**
  - **Can fetch and execute multiple instructions per cycle**

Prog. Model 2: Data Parallel (SIMD)



VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

Realization: Each iteration is independent

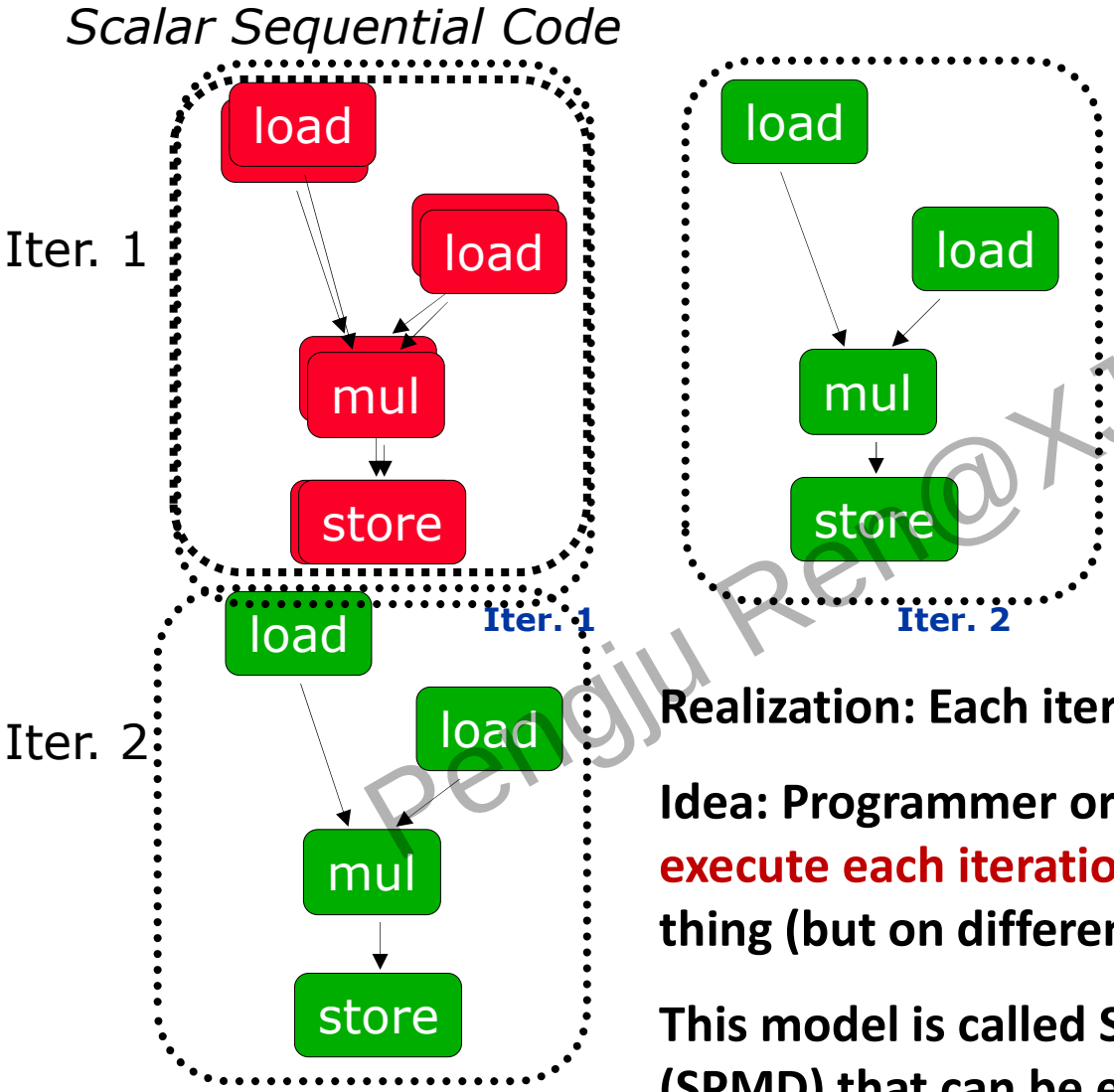
Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

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

Best executed by a SIMD processor (vector, array)

Prog. Model 3: Multithreaded (SPMD)

Scalar Sequential Code




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

....

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

This model is called Single Program Multiple Data (SPMD) that can be executed on a SIMT machine

Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice
- New applications, such as machine vision, speech recognition, machine learning and machine intelligence, etc. all require large numerical computations that are often trivially data parallel
- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

GPU Computing Applications

A Major Paradigm Shift

In the 20th Century, we were able to understand, design, and manufacture what we can measure

– Physical instruments and computing systems allowed us to see farther, capture more, communicate better, understand natural processes, control artificial processes…

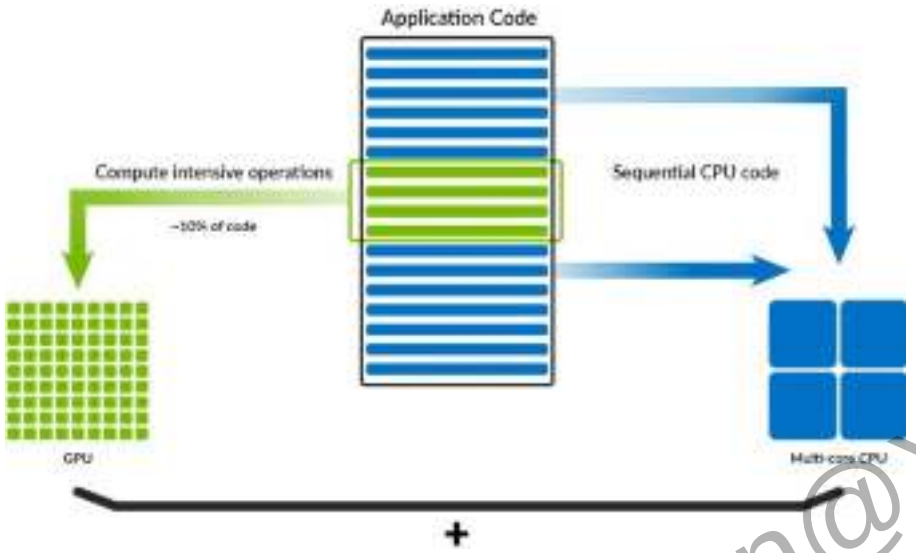
In the 21st Century, we are able to understand, design, and create what we can compute

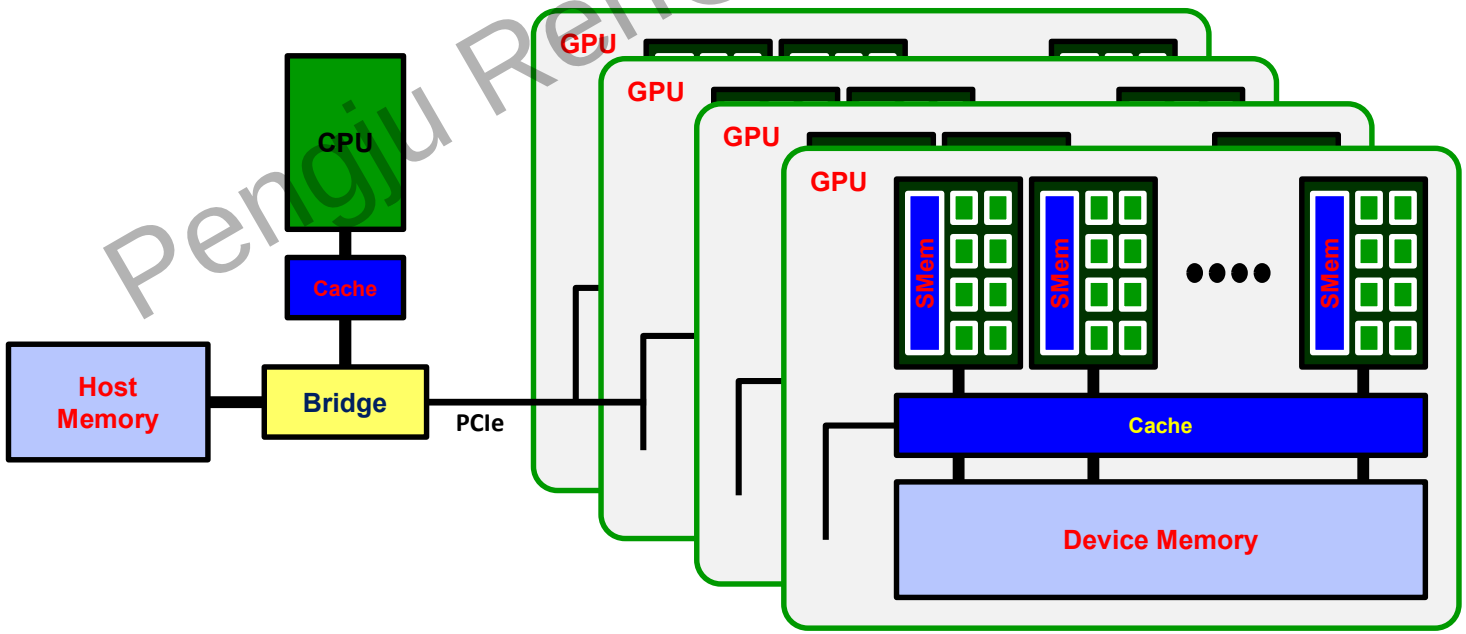
– Computational models are allowing us to see even farther, going back and forth in time, learn better, test hypothesis that cannot be verified any other way, create safe artificial processes…

The Leading Actor: Graphical Processing Units(GPU)

- Basic idea:
  - **Heterogeneous execution model**
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
    - (GeForce 8800, since 2006)
  - Programming model is "Single Instruction（Program） Multiple Thread"
    - **SPMD is from programmers' perspective**
    - **SIMT is the architecture view**
  - GPU hardware handles thread management, not applications or OS

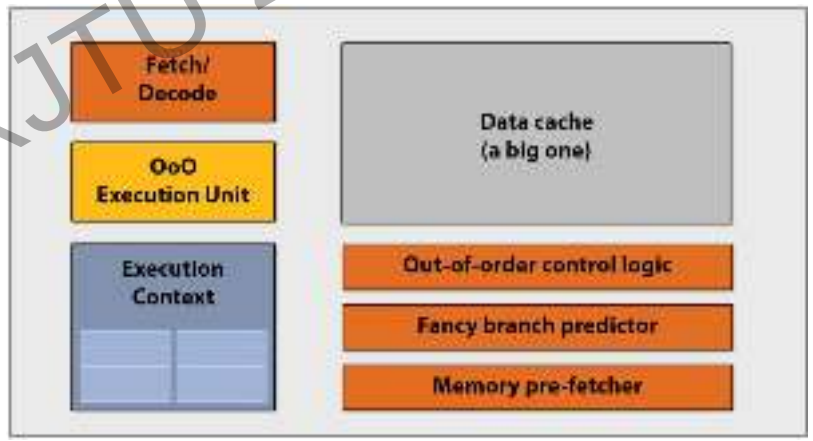
Heterogeneous CPU-GPU System Architecture





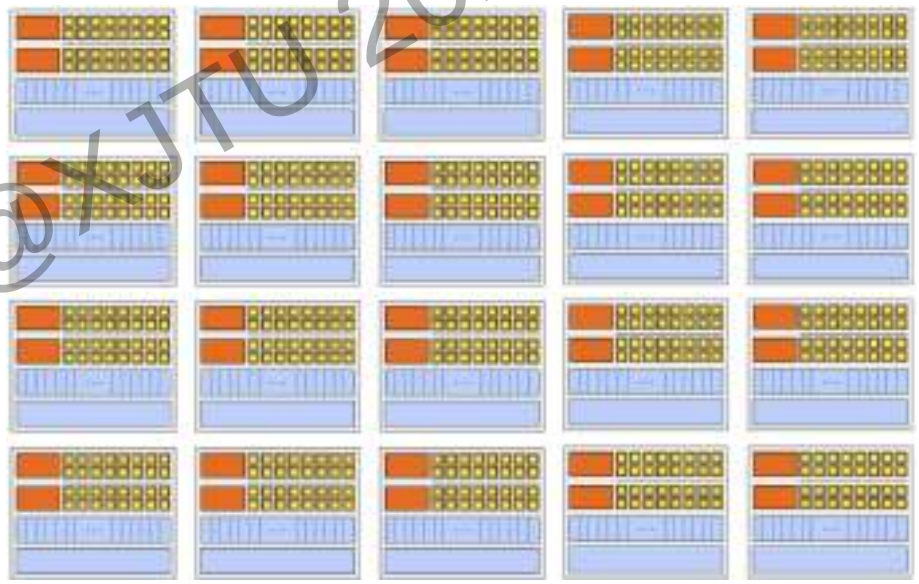
CPUs: Latency Oriented Design

- **High clock frequency (~3.0GHz)**
- **Large caches (L1 ~KBs, L2/3 ~MBs)**
  - Convert long latency memory accesses to short latency cache accesses
- **Sophisticated control**
  - SuperScalar & OOO Executions
  - Branch prediction for reduced branch penalty
  - Data forwarding for reduced data latency
- **Powerful ALU**
  - Reduced operation latency
  - SIMD Units for High-end CPUs



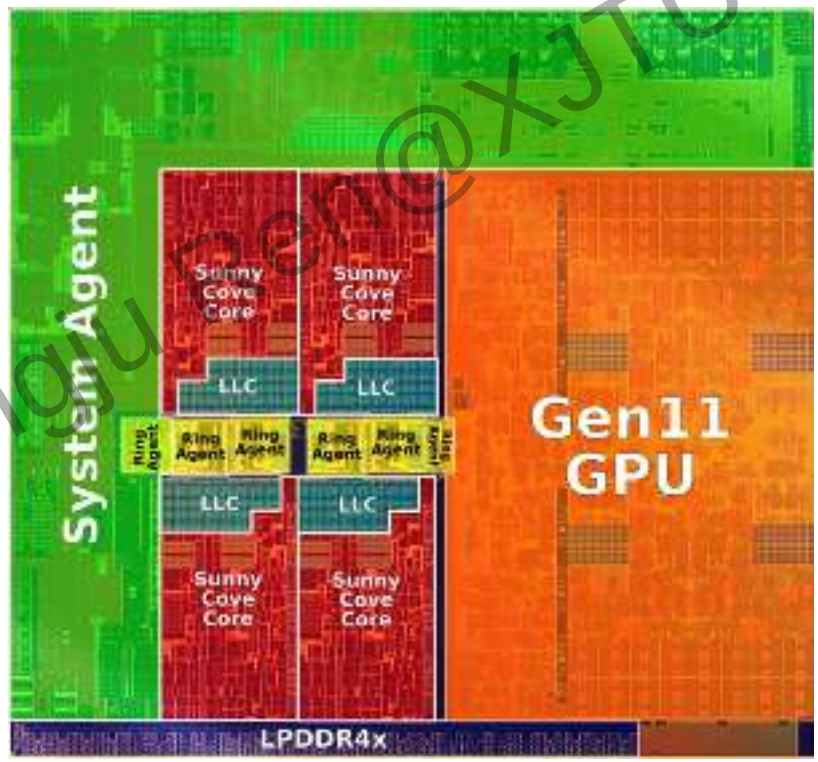
GPUs: Throughput Oriented Design

- **Moderate clock frequency(~1Ghz)**
- **Small caches (Scratchpad)**
  - To boost memory throughput
- **Simple control**
  - No branch prediction
  - No data forwarding
- **Energy efficient ALUs**
  - Many, long latency but heavily pipelined for high throughput
- **Require massive number of threads to tolerate latencies**



Integrate CPU & GPU together

- CPUs for sequential parts where latency hurts
  - CPUs can be +10x faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be +10x faster than CPUs for parallel code

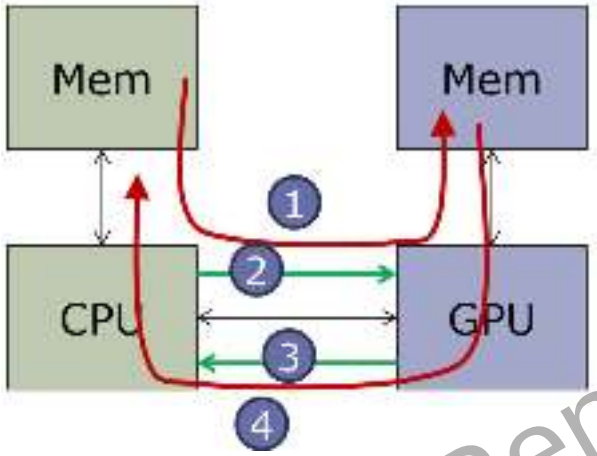


Ice Lake is branded as 10th Generation Core i3, i5, and i7 processors

Programming work flow of GPU

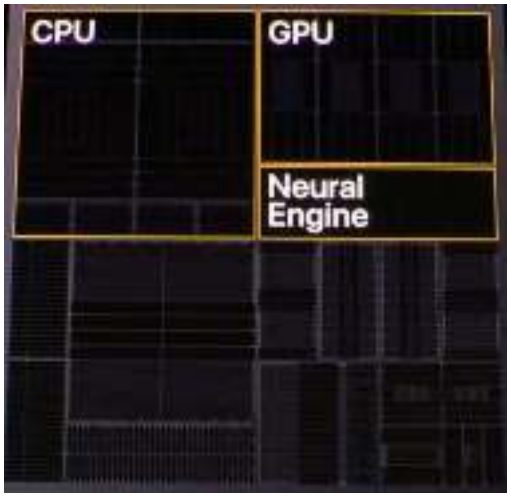
- **Learning how GPUs are programmed is relevant to a computer architect interested in GPU computing to gain a better understanding of the *hardware/software interface***
  - Identify compute intensive parts of an application
  - Adopt/create scalable algorithms
  - Optimize data arrangements to maximize locality
  - Performance Tuning
  - Pay attention to code portability, scalability, and maintainability
- **Three key abstractions—a hierarchy of thread groups, shared memories, and barrier synchronization**

GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- **Separate DRAM memory spaces for CPU(often called system memory) and GPU(often called device memory)**
- **Data transfers can dominate execution time**
- **Integrated GPUs with unified address space (CPU & GPU contend for memory)**



Apple A13 @Iphone 11

Example 1 : Vector Addition

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}
int main()
{
   size_t size = N*sizeof(float);

  // Memory allocation for A_h, B_h, and C_h
  float* A_h = (float*)malloc(size);
  float* B_h = (float*)malloc(size);
  float* C_h = (float*)malloc(size);
   // I/O to read A_h and B_h, N elements
…
   vecAdd(A_h, B_h, C_h, N);
}
```

Traditional C Code

Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
int size = n* sizeof(float);
float *A_d, *B_d, *C_d;

cudaMalloc((void **) &A_d, size);
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
cudaMalloc((void **) &B_d, size);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
cudaMalloc((void **) &C_d, size);

vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

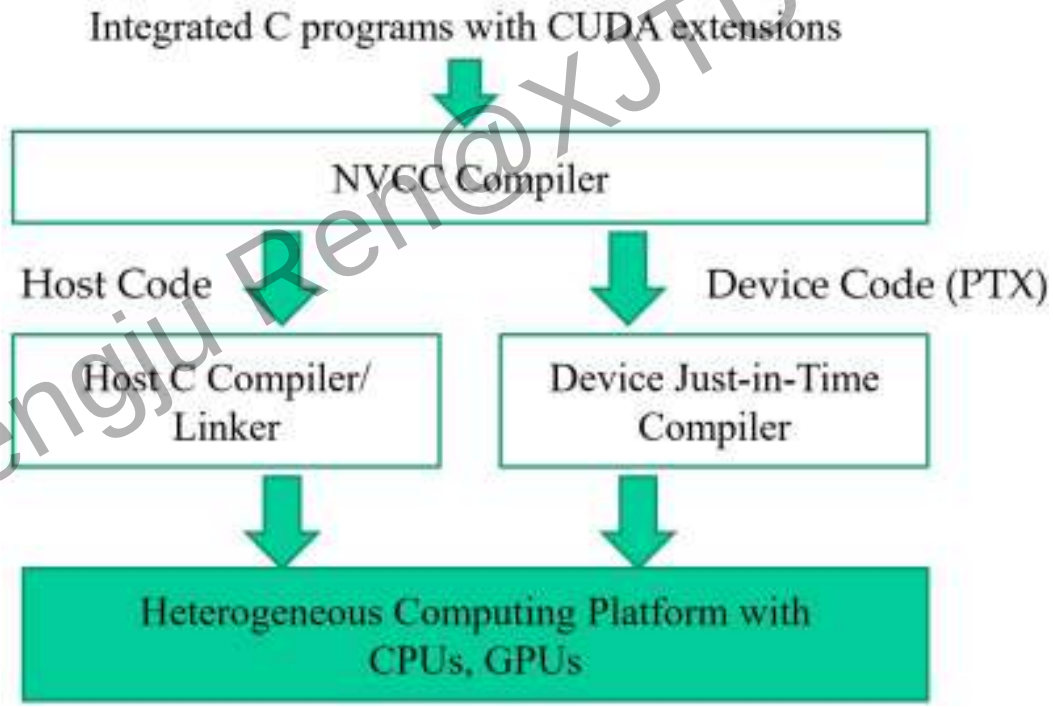
```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i= blackDim.x*blockIdx.x+threadIdx.x;
    if (i<n)  C[i] = A[i] + B[i];
}
```

Device Code

CUDA C Code （Heterogeneous Computing)

CUDA Function Declarations and Compiling

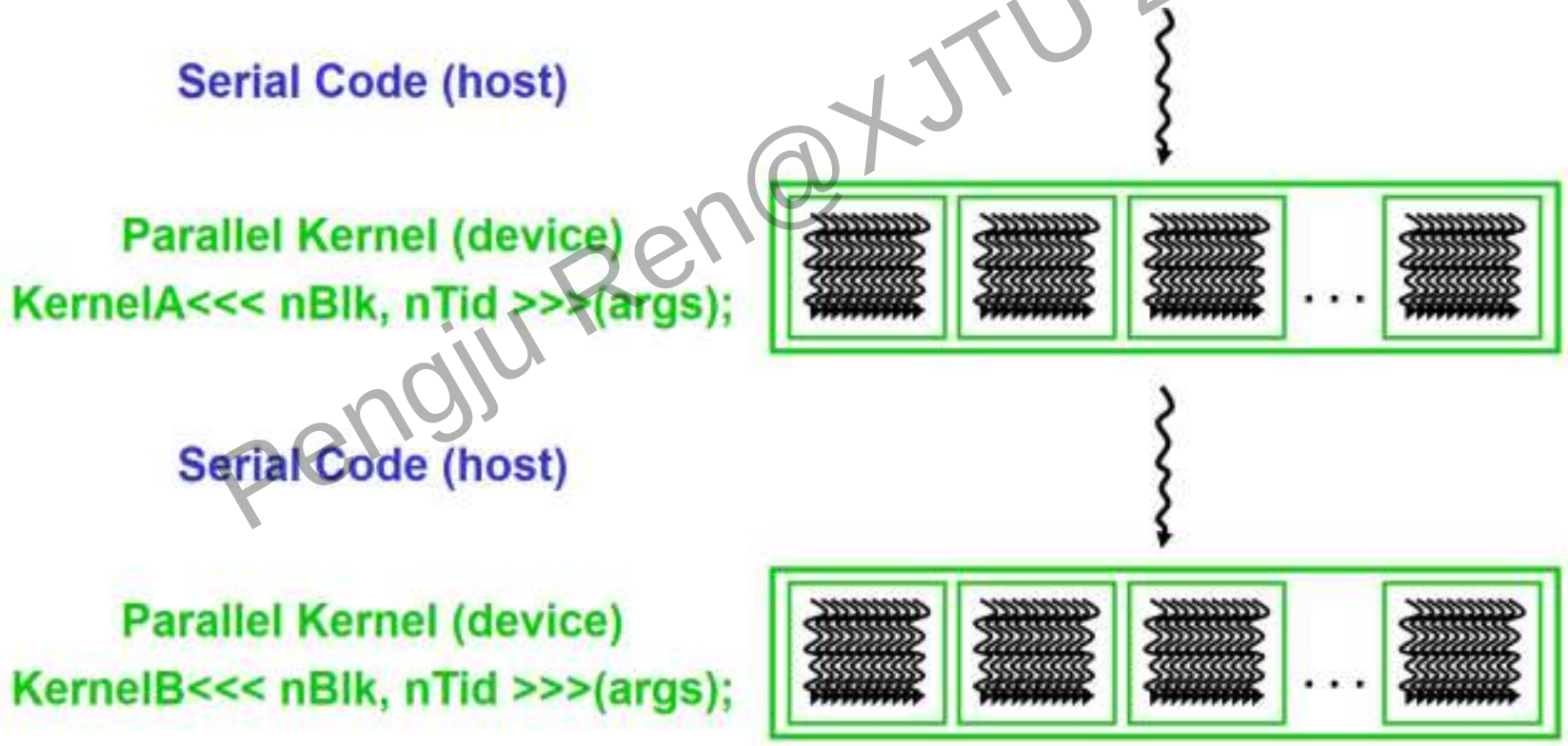
	Executed on	Only callable from
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host



Source files compiled with NVCC can include a mix of host code and device code

CUDA/OpenCL – Execution Model

Integrated **host(CPU)+device(GPU)** app C program
– Serial or modestly parallel parts in host C code
– Highly parallel parts in device SPMD kernel C code



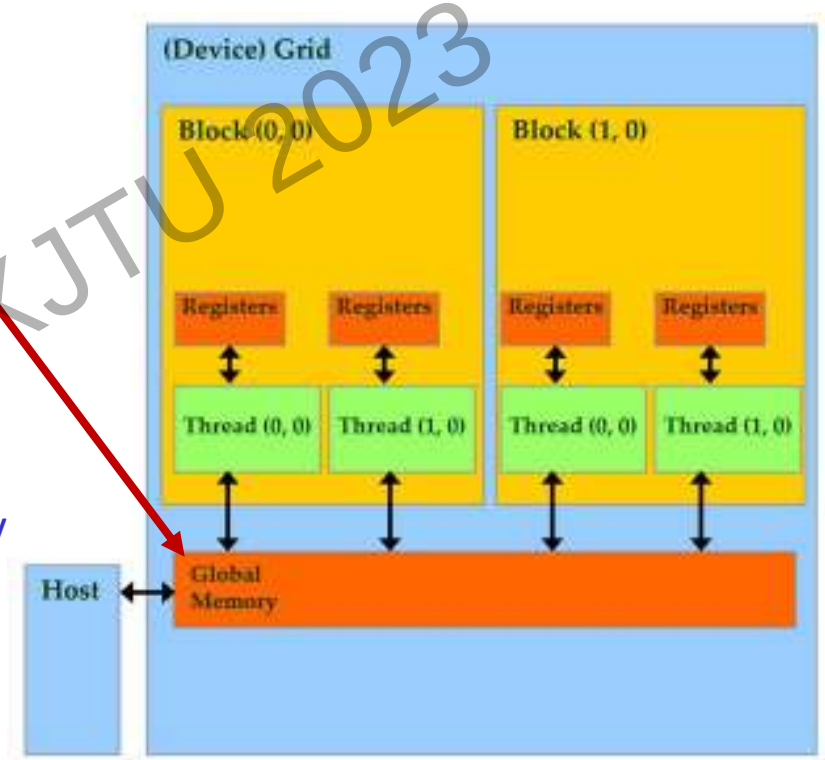
Partial Overview of CUDA Memories
(CUDA Device Memory Management API)

cudaMalloc()
- Allocates object in the device global memory
- Two parameters
  - **Address of a pointe**r to the allocated object
  - **Size of** the allocated object in terms of bytes

cudaFree()
- Frees object from device global memory
  - **Pointer** to freed object

Host code can
- Transfer data to/from per grid global memory

cudaMemcpy()
- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer



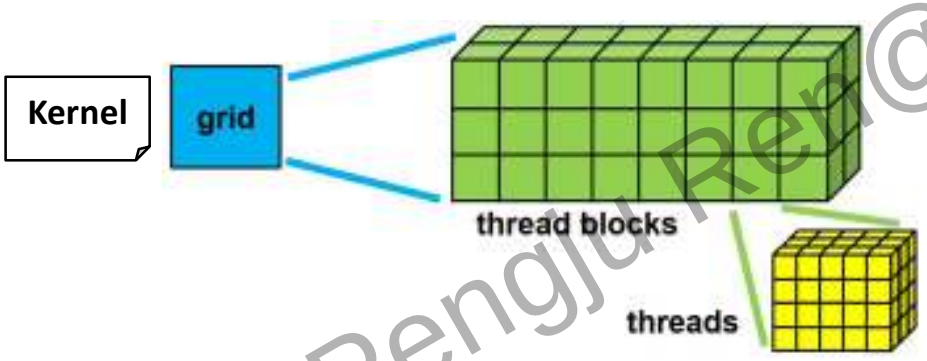
Arrays of Parallel Threads

A CUDA kernel is executed as a **grid** (array) of threads:

– All threads in a grid run the same **kernel** code, the Single Program Multiple Data (SPMD model)

– Each thread has **a unique index** that it uses to compute memory addresses and make control decisions

dim3 dimGrid(8,3,2) //girdDim.x, gridDim.y, gridDim.z)



Thread block and thread organization
- **simplifies memory addressing**
- **when processing multidimensional data**

e.g. Image\Video Processing, Vectors\Matrices\Tensors

dim3 dimBlock(5,4,3) //blockDim.x, blockDim.y, blockDim.z)

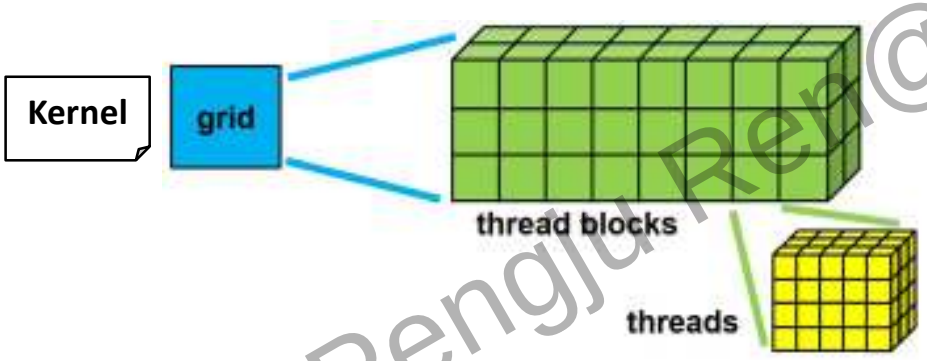
Each CUDA kernel
- is executed by a **grid**
- a 3D array of **thread blocks**, which are 3D arrays of **threads**.

Each thread
- executes the **same program**
- on **distinct data inputs**
- a single-program, multiple-data (**SPMD**) model

Blockidx and Threadidx are unique

- **Each block has a unique index tuple**
  - blockIdx.x (from 0 to (gridDim.x – 1))
  - blockIdx.y (from 0 to (gridDim.y – 1))
  - blockIdx.z (from 0 to (gridDim.z – 1))
- **Each thread has a unique index tuple**
  - threadIdx.x (from 0 to (blockDim.x – 1))
  - threadIdx.y (from 0 to (blockDim.y – 1))
  - threadIdx.z (from 0 to (blockDim.z – 1))

dim3 dimGrid(8,3,2) //girdDim.x, gridDim.y, gridDim.z)



Thread block and thread organization
- **simplifies memory addressing**
- **when processing multidimensional data**

e.g. Image / Video Processing, Vectors\Matrices\Tensors

dim3 dimBlock(5,4,3) //blockDim.x, blockDim.y, blockDim.z)

Example Vector Addiction Kernel

A: DimGrid B: DimBlock

```
vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
```

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i= blackDim.x*blockIdx.x+threadIdx.x;
    if (i<n)  C[i] = A[i] + B[i];
}
```

A: Number of blocks per dimension

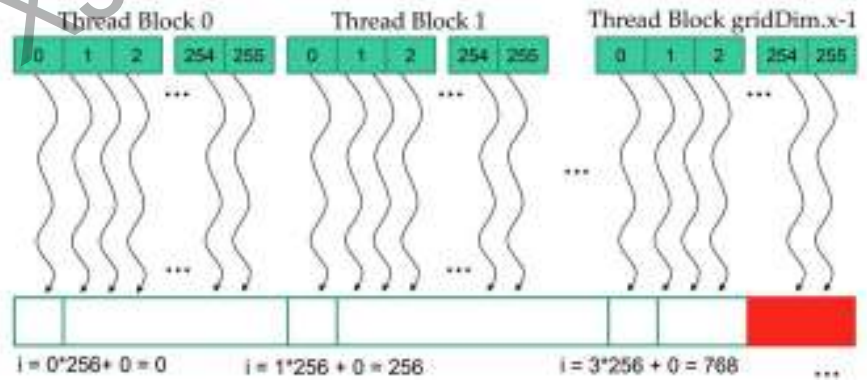
B: Number of threads per dimension in a block

C: Number of threads per block in x dimension

D: Unique block # in x dimension

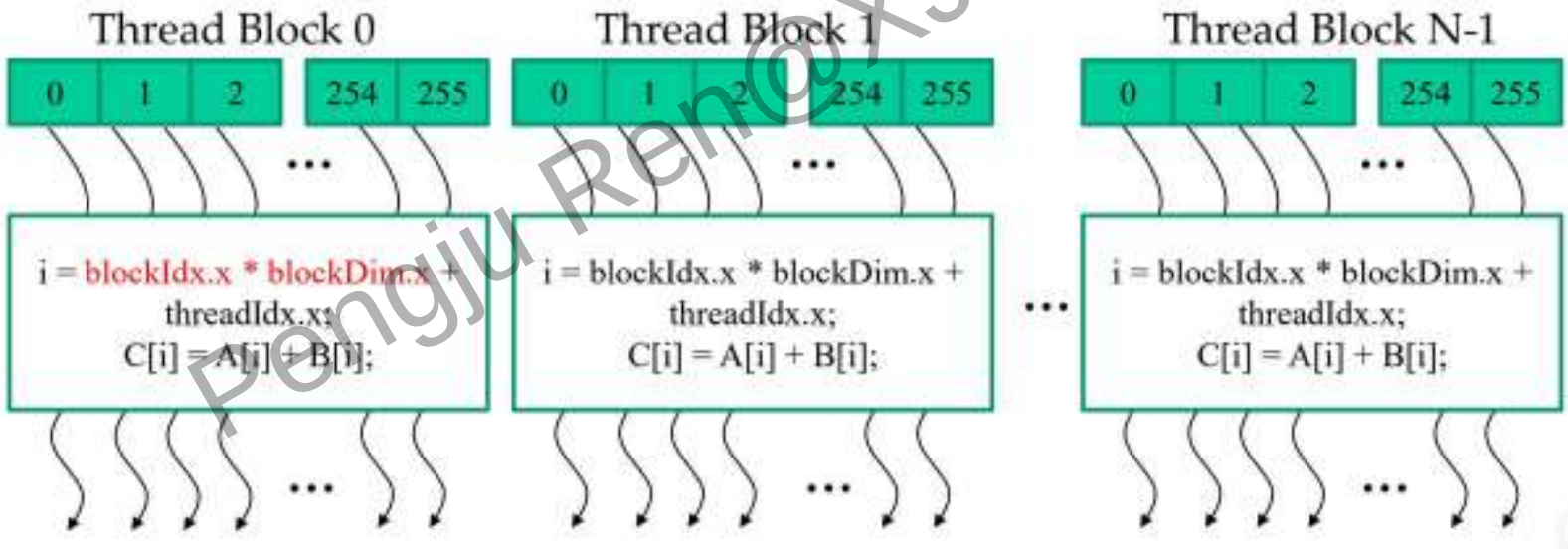
E: Unique thread # in x dimension in the block

How many blocks and threads in total will be executed in this example? (if n =1000)



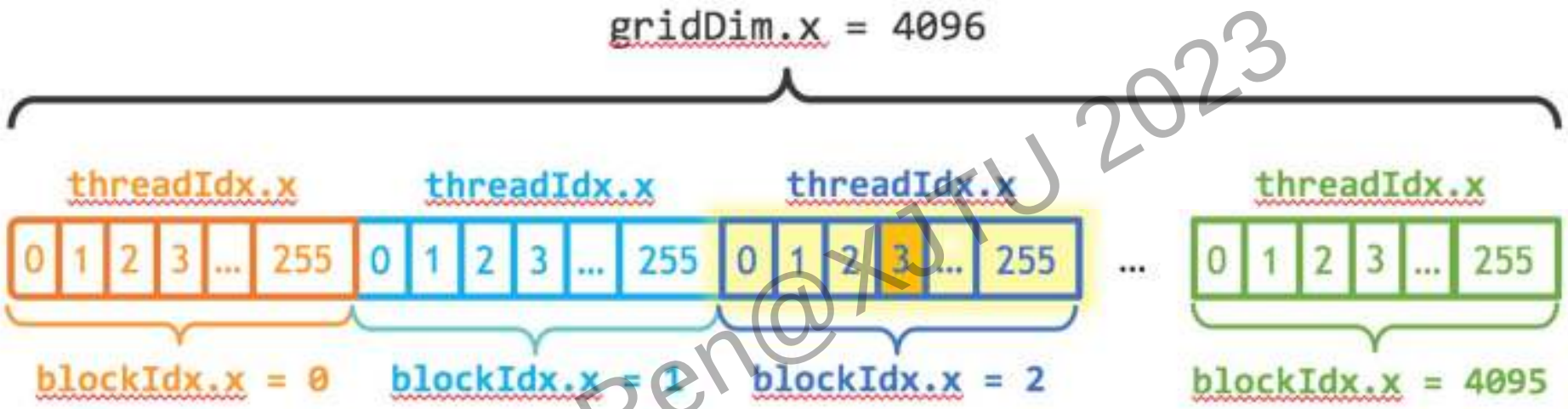
CUDA Execution Model: Threads Blocks

- Thread blocks are required to **execute independently**, should be executed in any order, in parallel or in series
- Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (to be covered later)
- Threads in different blocks cooperate less



For convenience, CUDA C provides a special shortcut for launching a kernel with one-dimensional (as default) grids an blocks

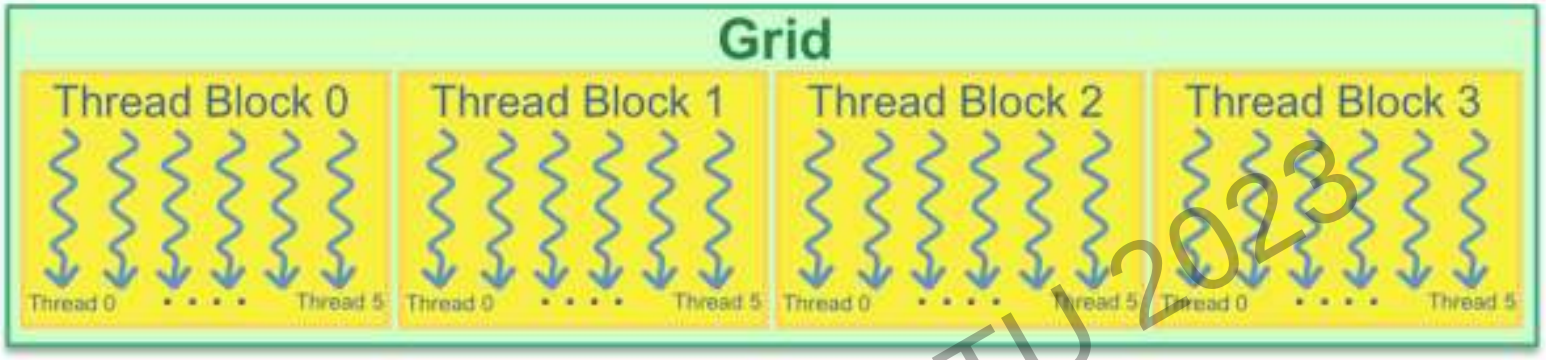
Index of Grids, Blocks and Threads



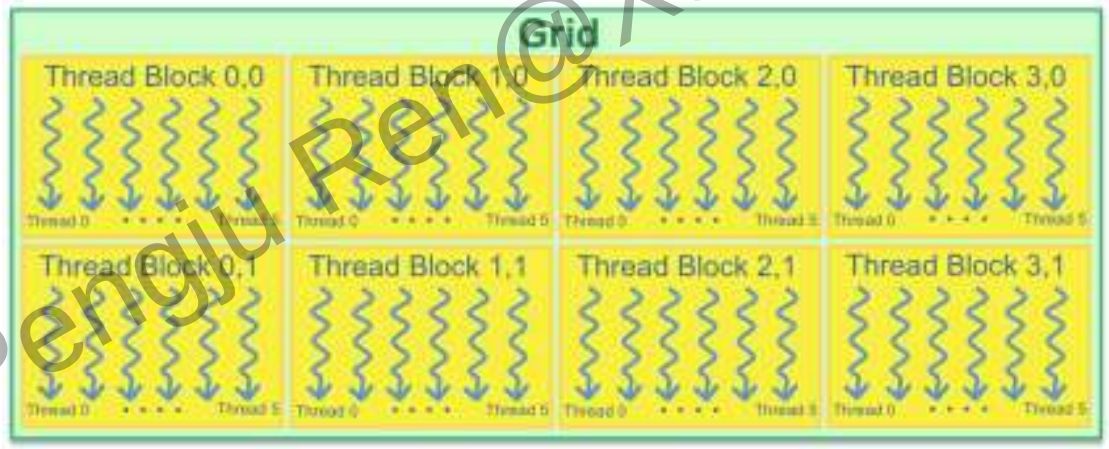

```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *    (256)    +     (3)      = 515
```

$$n = 4096 \times 256 = 1048576$$

Grids and Thread Blocks



1D Grid of 1D Blocks: blockIdx.x *blockDim.x + threadIdx.x;

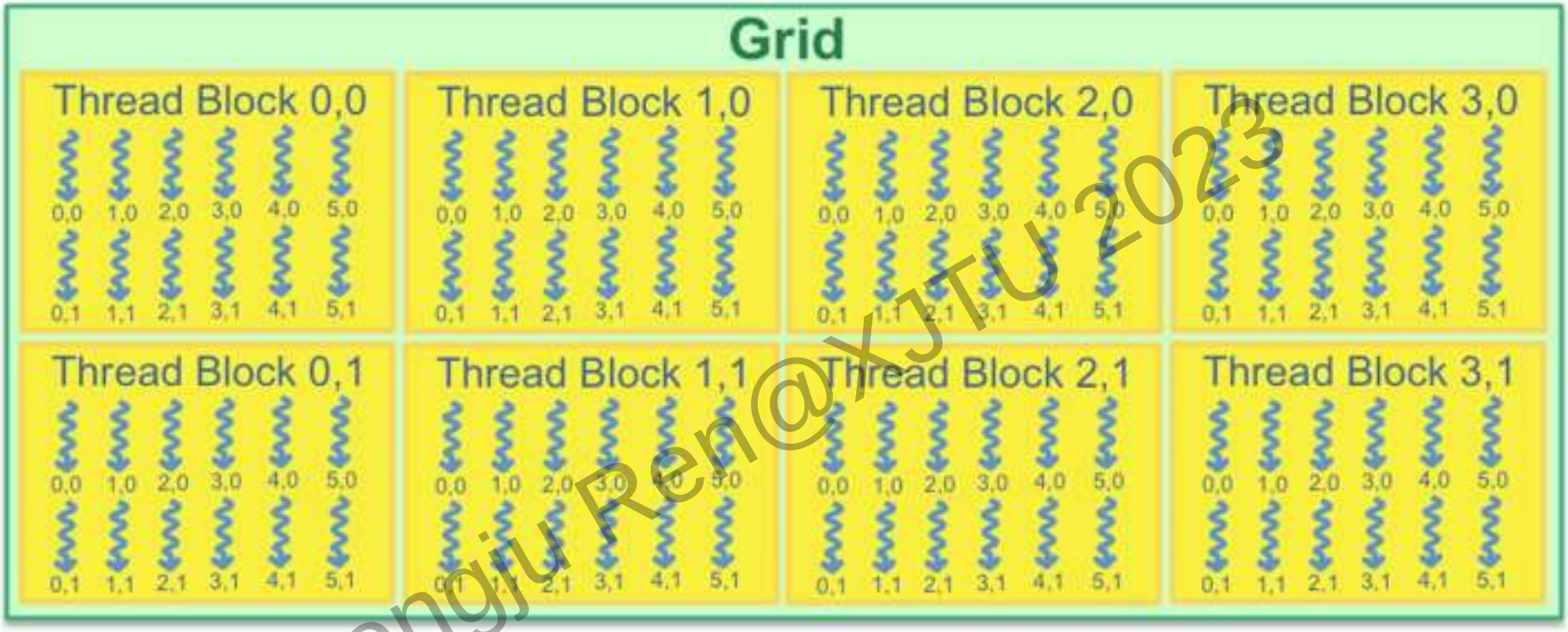


A 2D Grid of 1D Blocks:

int blockId = blockIdx.y * gridDim.x + blockIdx.x;

int threadId = blockId * blockDim.x + threadIdx.x;

Grids and Thread Blocks



A 2D Grid of 2D Blocks:

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y)
             + (threadIdx.y * blockDim.x) + threadIdx.x;
```

Indexing and Memory Access

- **Images are 2D data structures**
  - **height x width**
  - **Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width**

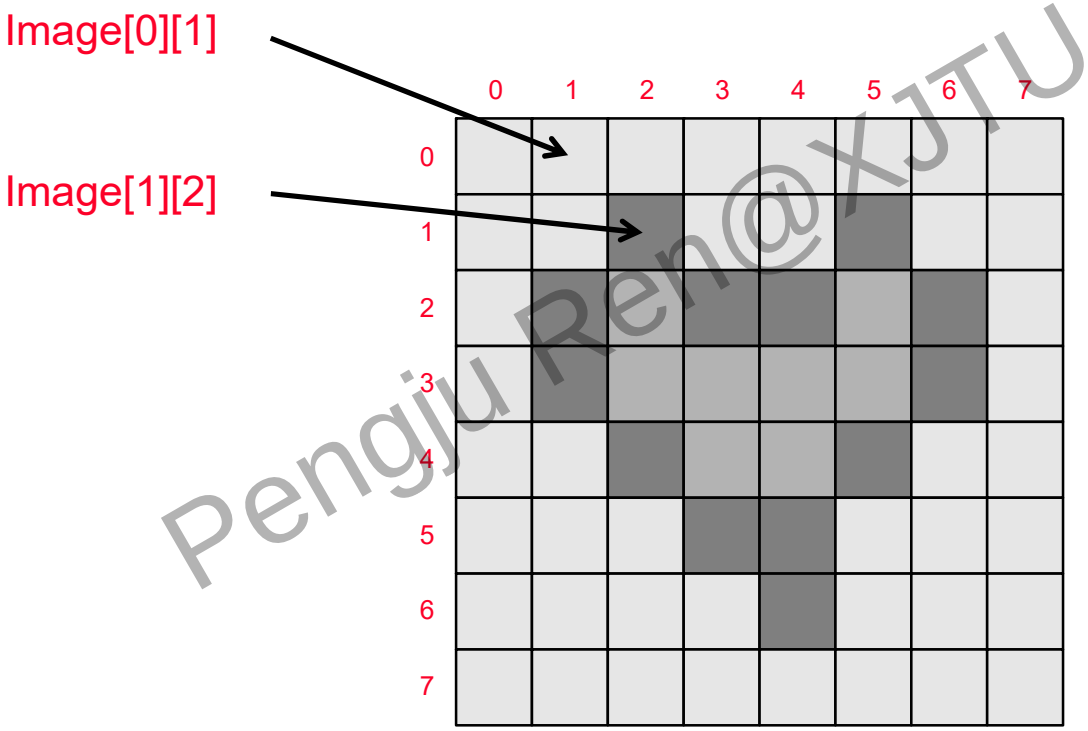
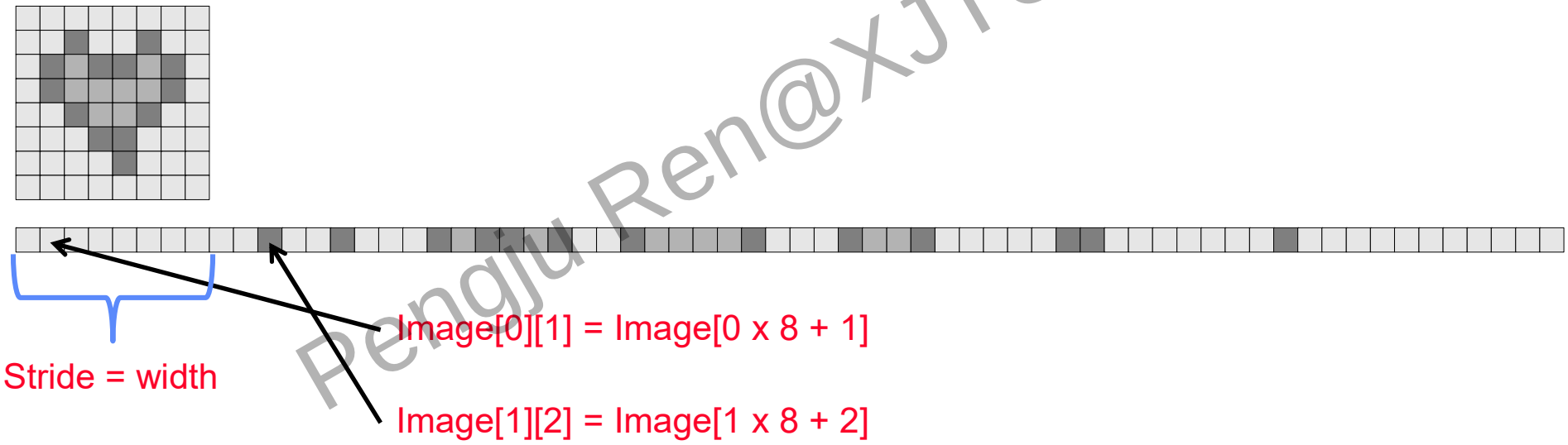


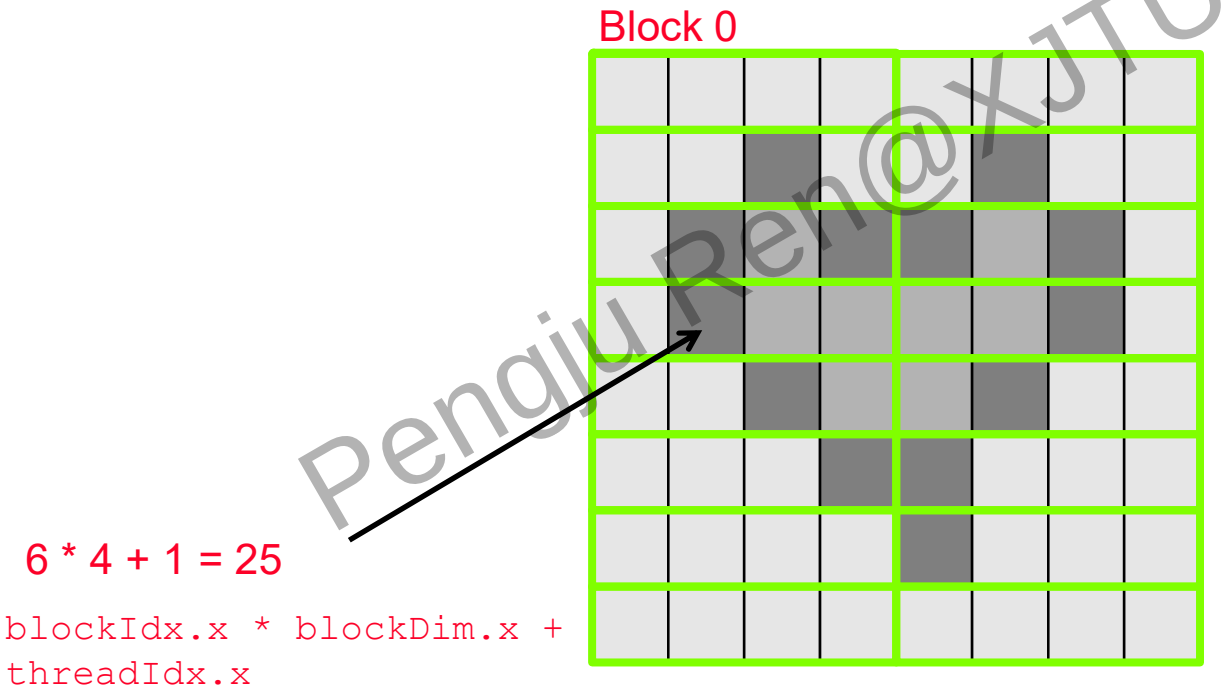
Image Layout in Memory

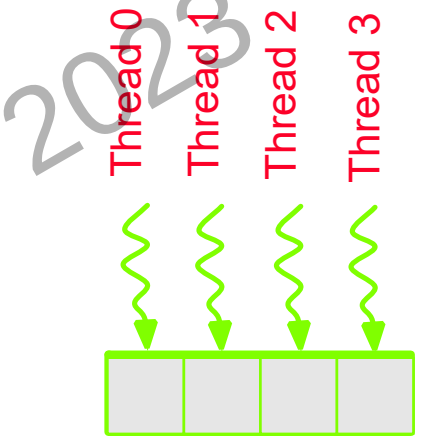
- **Row-major layout**
- **Image[j][i] = Image[j x width + i]**



Indexing and Memory Access: 1D Grid

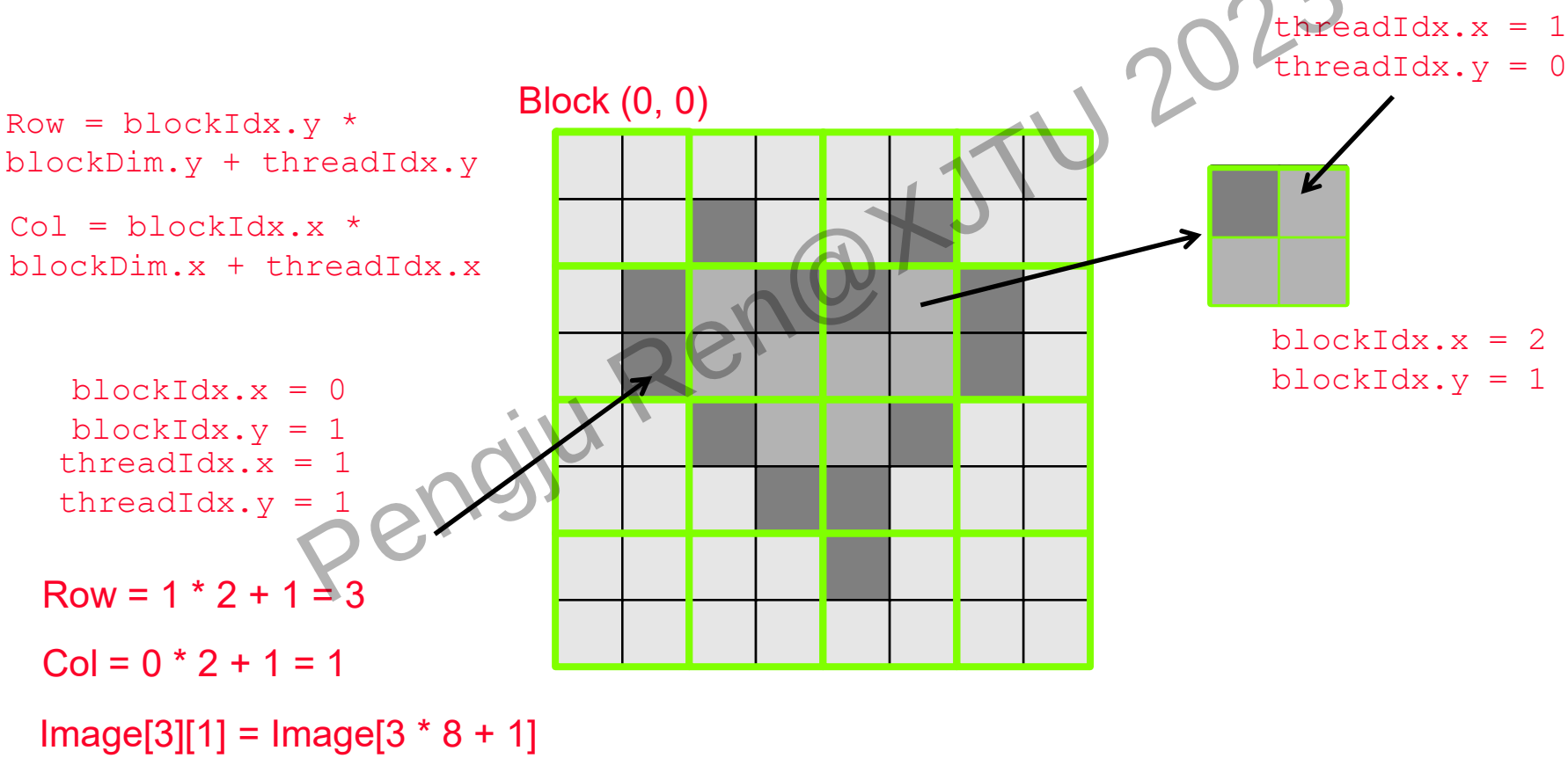
- One GPU thread per pixel
- Grid of Blocks of Threads
  - `gridDim.x, blockDim.x`
  - `blockIdx.x, threadIdx.x`



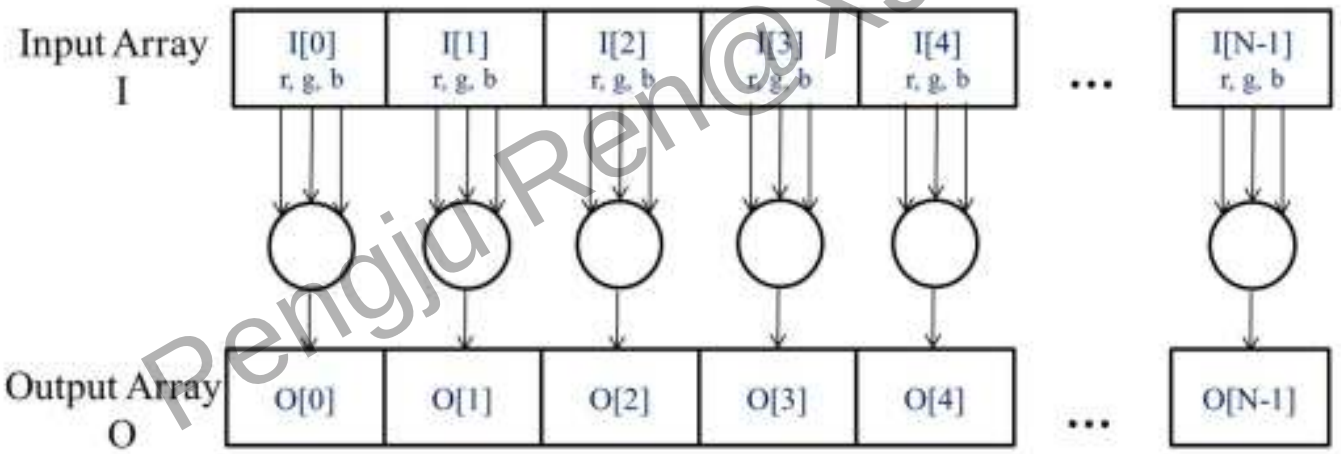


Indexing and Memory Access: 2D Grid

- **2D blocks**
  - **gridDim.x, gridDim.y**



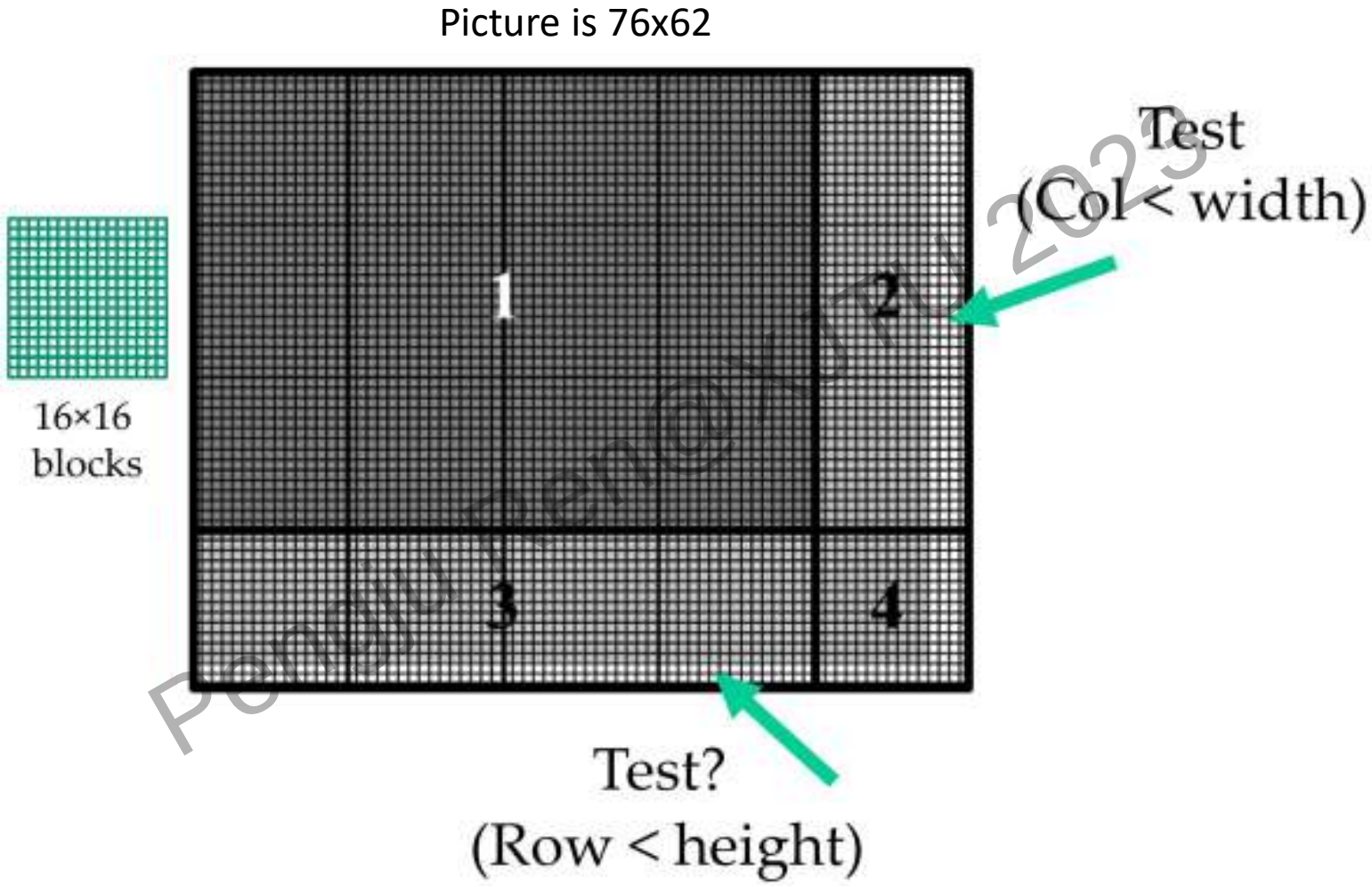
Example 2：Convert a Color image to a Gray one



Pixels can be calculated independently

Gray = 0.21*r + 0.71*g + 0.07*b

Example 2 : Conversion a Color image to a Gray one



Example 2：Conversion a Color image to a Gray one

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned ch                              e,
                int width, int height)
{
    int Col =  threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;

    if (Col < width && Row < height) {
        // get 1D coordinate for the grayscale image
        int greyOffset = Row*width + Col;
        // one can think of the RGB image having
        // THREE times as many columns of the gray scale image
        int rgbOffset = 3 * greyOffset;
        unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point const
        grayImage[grayOffset] = 0.21f*r +
    }
}
```

Indexing by ThreadId and BlockId

RGB(8-8-8)->Gray(8), So the width of the output is 1/3 of the original one

Example 3: Image Blurring

Patch is 3x3

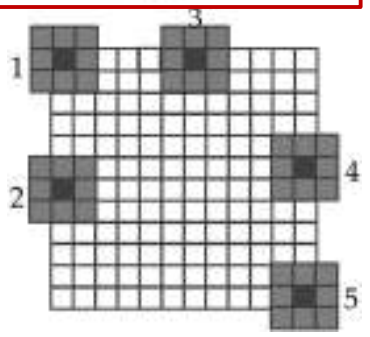
Output = average of 9 Pixels

Example 3: Image Blurring

```
__global__
 void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
   int Col  = blockIdx.x * blockDim.x + threadIdx.x;
   int Row  = blockIdx.y * blockDim.y + threadIdx.y;

   if (Col < w && Row < h) {
1.     int pixVal = 0;
2.     int pixels = 0;

     // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.     for(int blurRow = -BLUR_SIZE; blurRow <= BLUR_SIZE; ++blurRow) {
4.       for(int blurCol = -BLUR_SIZE; blurCol <= BLUR_SIZE; ++blurCol) {

5.         int curRow = Row + blurRow;
6.         int curCol = Col + blurCol;
         // Verify we have a valid image pixel
7.         if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.           pixVal += in[curRow * w + curCol];
9.           pixels++; // Keep track of number of pixels in the avg
           }
         }
       }
     // Write our new pixel value out
10.    out[Row * w + Col] = (unsigned char)(pixVal / pixels);
   }
 }
```

*The **BLUR_SIZE** is set such that **2xBLUR_SIZE** gives the number of pixels on each side of the patch*

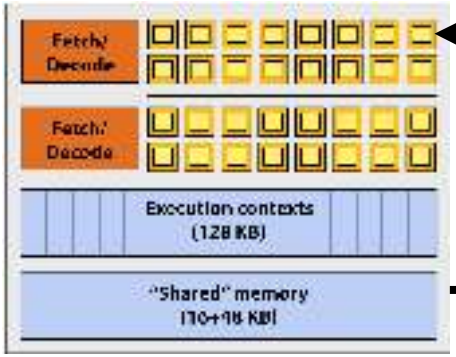
*Handing boundary conditions for **pixels near the edges** of the image*



Introduction of GPGPU (Program and Arch)

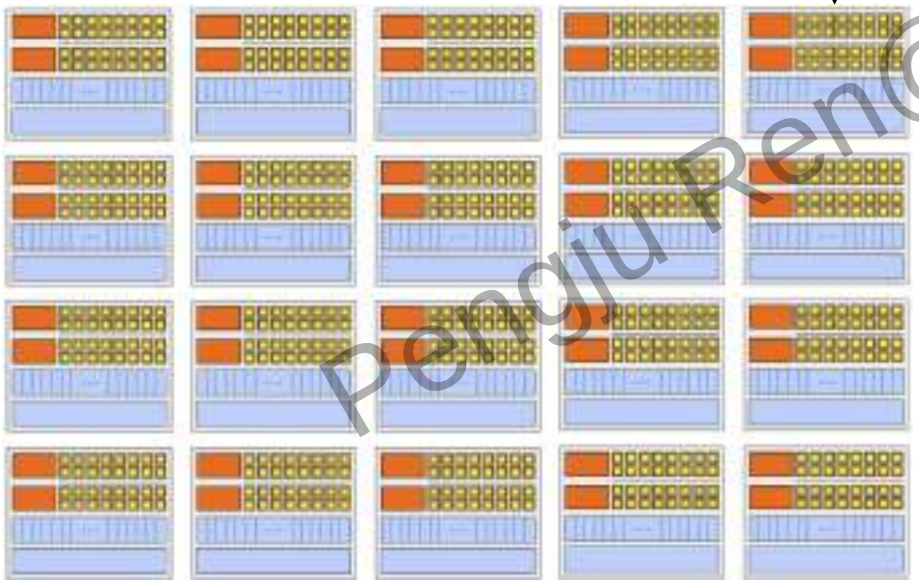
- Programming Model and Interface
- **Hardware Implementation**
- Optimization through S/H Cooperation

GPUs: Extreme throughput-oriented processors



SIMD function unit, control shared across tens of units (1 MUL-ADD per clock)

Streaming multiprocessors (SM)



68 cores (SM)

Each SM contains 2 FP64 64 FP32,
64INT32, 8 Tensor Core,
1 RT core

Special hardware for *ray tracing*

Special hardware for *neural nets*

Nvidia RTX 2080 (TU104 Chip)

Example：Nvidia Volta GV100

Volta GV100 Full GPU with 84 SM Units

NVIDIA Turing GPU architecture whitepaper

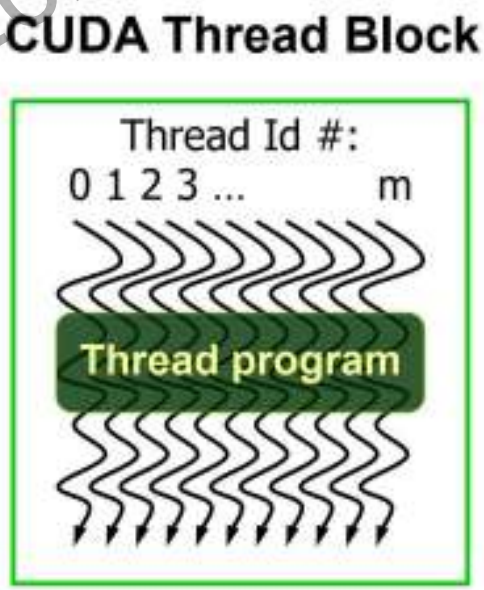
Example: Nvidia Volta GV100 – SM

- The GV100 SM is partitioned into 4 processing units
- Each unit with 16 FP32 Cores, 8 FP64 Cores, 16 INT32 Cores, two of the new mixed-precision Tensor Cores for deep learning matrix arithmetic
- Each unit has 16,384x32-bit Regs

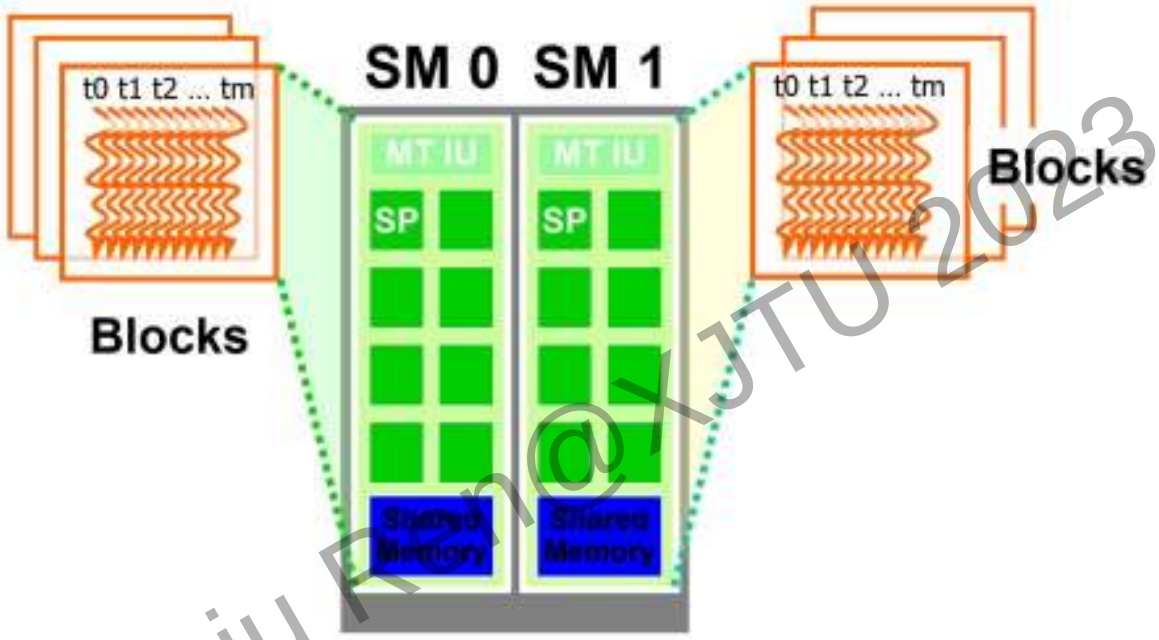
NVIDIA Turing GPU architecture whitepaper

Hardware Execution Model

- **CPU sends whole "*grid*" over to GPU, which distributes *thread blocks* among cores (each thread block executes on one core)**
  - **Programmer unaware of number of cores**
- Threads within ***block*** have ***thread index*** numbers
- Kernel code uses ***thread index*** and ***block index*** to select work and address shared data
- Threads in the same block **share data** and **synchronize** while doing their share of the work
- Threads in different blocks cannot cooperate
- Blocks **execute in arbitrary order**!



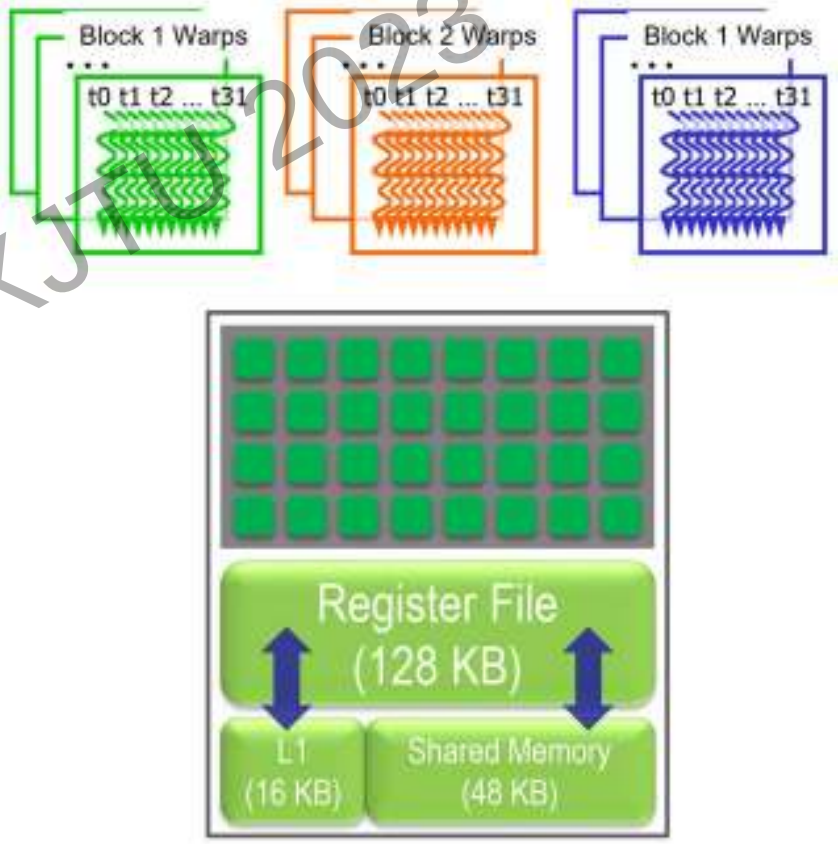
Executing Thread Blocks



- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to **32** blocks to each SM (e.g. resource limit for Maxwell)
  - Maxwell SM can take up to **2048** threads
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

Thread Scheduling

- Each block, groups of ***32***-threads(**warps)** in a thread block are executed simultaneously using *32-wide SIMD execution*.
  – An implementation decision, not part of the CUDA programming model
  – Warps are divided based on their *linearized thread index*
  – Threads 0-31(warp 0) Threads 32-63(warp 1)…
  – Warps are scheduling units in SM
- SM implements zero-overhead warp scheduling
  – Warps whose next instruction has its operands ready for consumption are eligible for execution
  – Eligible warps are selected for execution on a prioritized scheduling policy
  – **All threads in a warp execute the same instruction when selected**



Block Granularity Considerations

For color To Greyscale Conversion, should one use 8×8, 16×16 or 32×32 blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks.

- **For 8×8, we have 64 threads per block. Each SM can take up to 1,536 threads, which is 1,536/64=24 blocks. But each SM can only take up to 8 Blocks, so only 512 threads (16 warps) go into each SM!**
- **For 16×16, we have 256 threads per block. Each SM can take up to 1,536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus, we use the full thread capacity of an SM.**
- **For 32×32, we have 1,024 threads per Block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.**

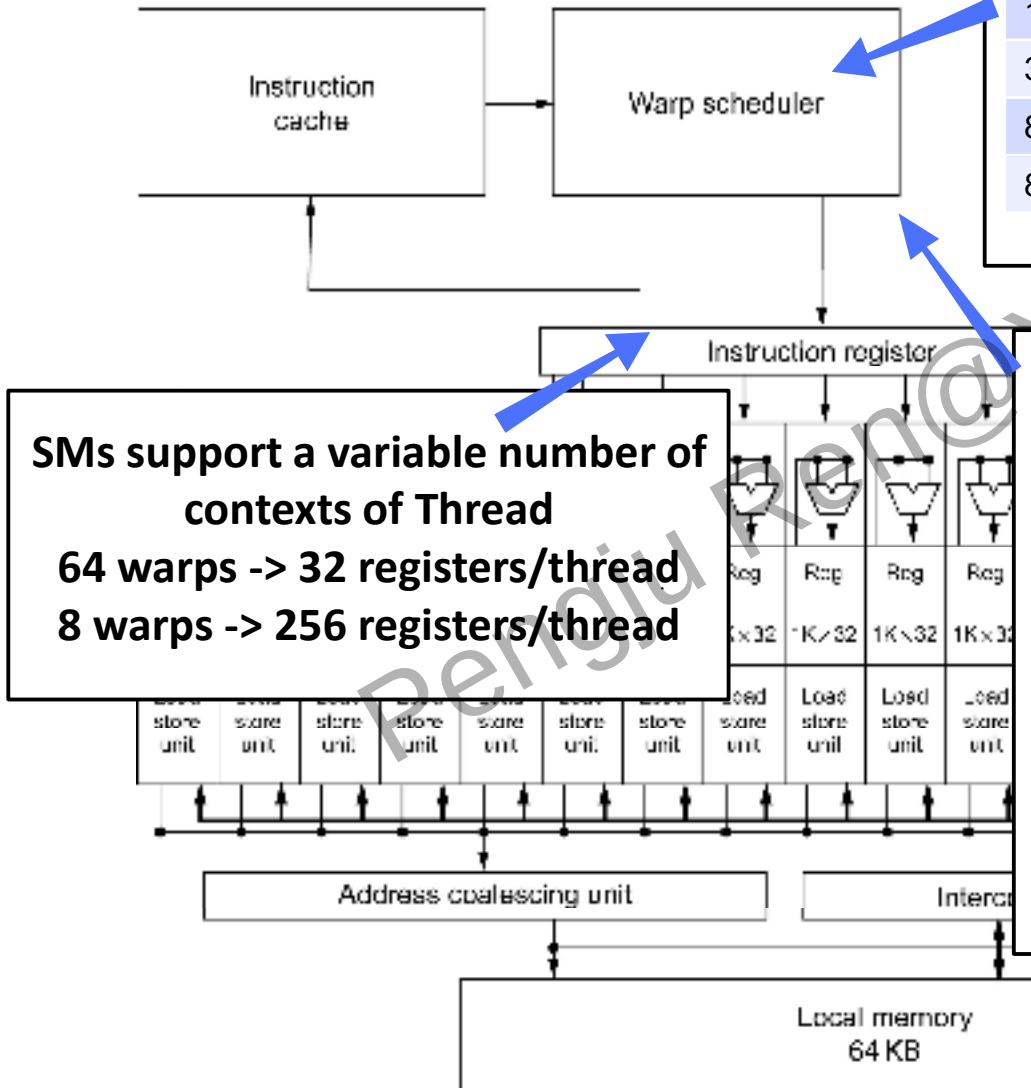
Streaming Multiprocessor(SM) Overview

Each SM supports tens of Warps
(e.g., 8～64 in Kepler) with 32 threads/warp

Warp Scheduler

Warp No.	Addr	SIMD instructions	Operands?
1	42	mul.f64	Ready
3	95	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Scoreboard



SMs support a variable number of contexts of Thread
64 warps -> 32 registers/thread
8 warps -> 256 registers/thread

Fine-grained multithreading (FGMT)

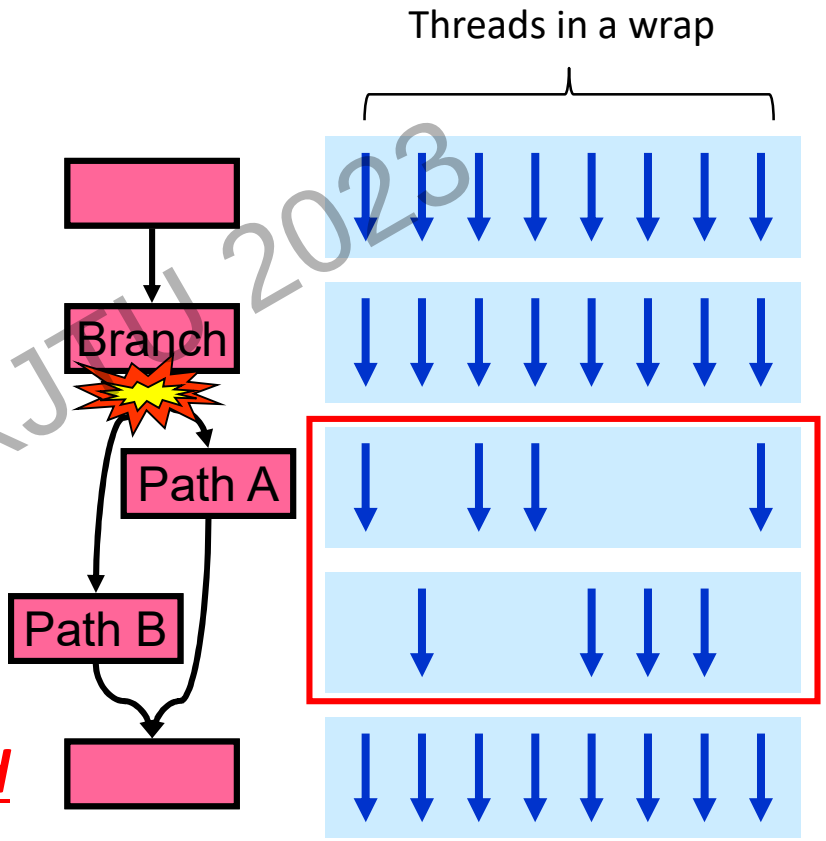
- **One instruction per thread in pipeline at a time (No interlocking)**
- **Interleave warp execution to hide latencies**

Register values of all threads stay in register file

FGMT enables long latency tolerance

Branch divergence

- If threads of a warp diverge via a *data-dependent conditional branch*
- Hardware tracks which µthreads take or don't take branch
  - If all go the same way, then keep going in SIMD fashion
  - If not, create mask vector indicating taken/not-taken
- *Keep executing one path under mask, push another branch PC+mask onto a hardware stack and execute later (SIMT stack)*
- When can execution of µthreads in warp reconverge?

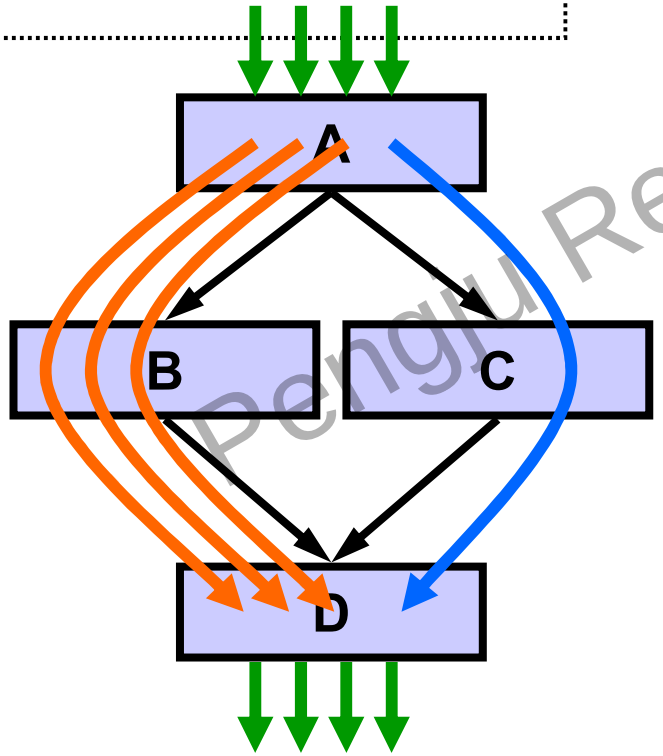


Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - » Entries consist of masks for each SIMD lane
    - » i.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - » Push on divergent branch
  - …and when paths converge
    - » Act as barriers
    - » Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Branch Divergence Handling

```
A;
if (some condition) {
    B;
} else {
    C;
}
D;
```

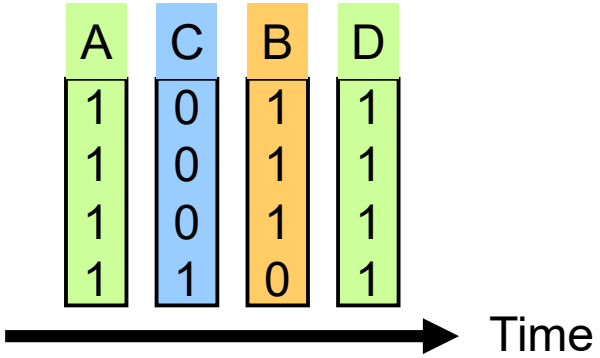


One per warp

Control Flow Stack (SIMT)

	Next PC	Recv PC	Active Mask
TOS →	D	--	1111
	B	D	1110
	C	D	0001

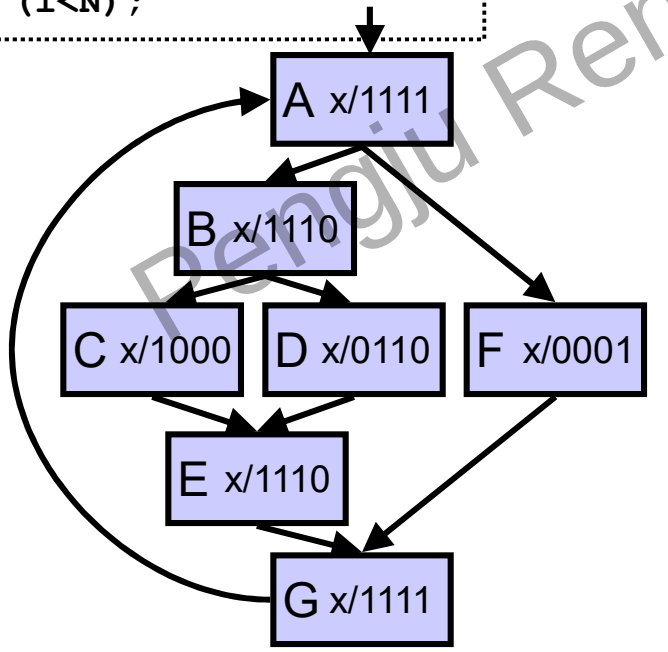
Execution Sequence



Assuming a GPU has 4 threads/warp

Branch Divergence Handling

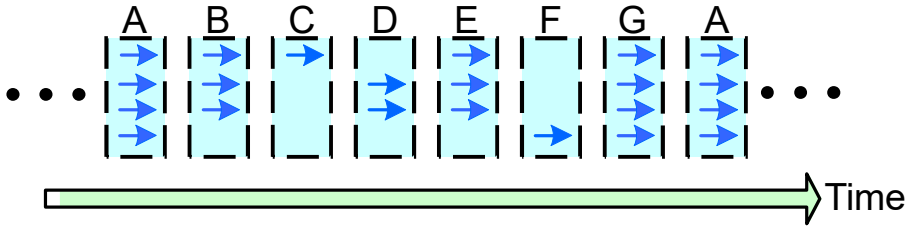
```
do {
A;
if (some condition1) {
   B;
     if (some condition2) {
         C;
    } else {
         D; // !condition2
    }
   E;
  }else F; // !condition1
}
i++;   // G;
}while (i<N);
```



One per warp

Control Flow Stack

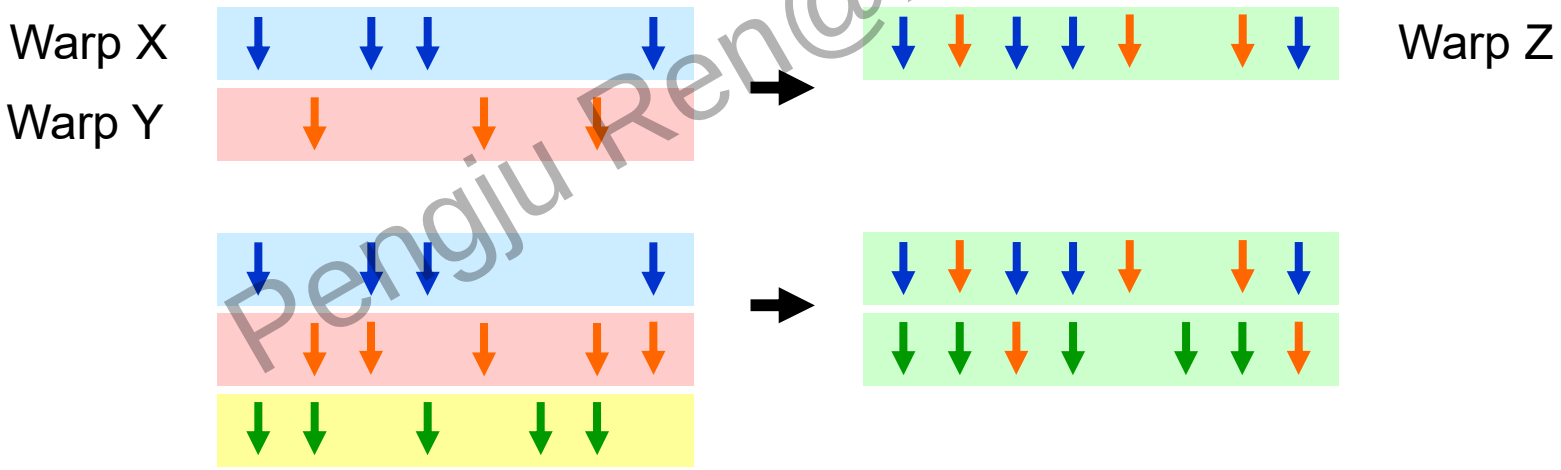
Next PC	Recv PC	Active Mask
G	--	1111
F	G	0001
E	G	1110
D	E	0110
C	E	1000



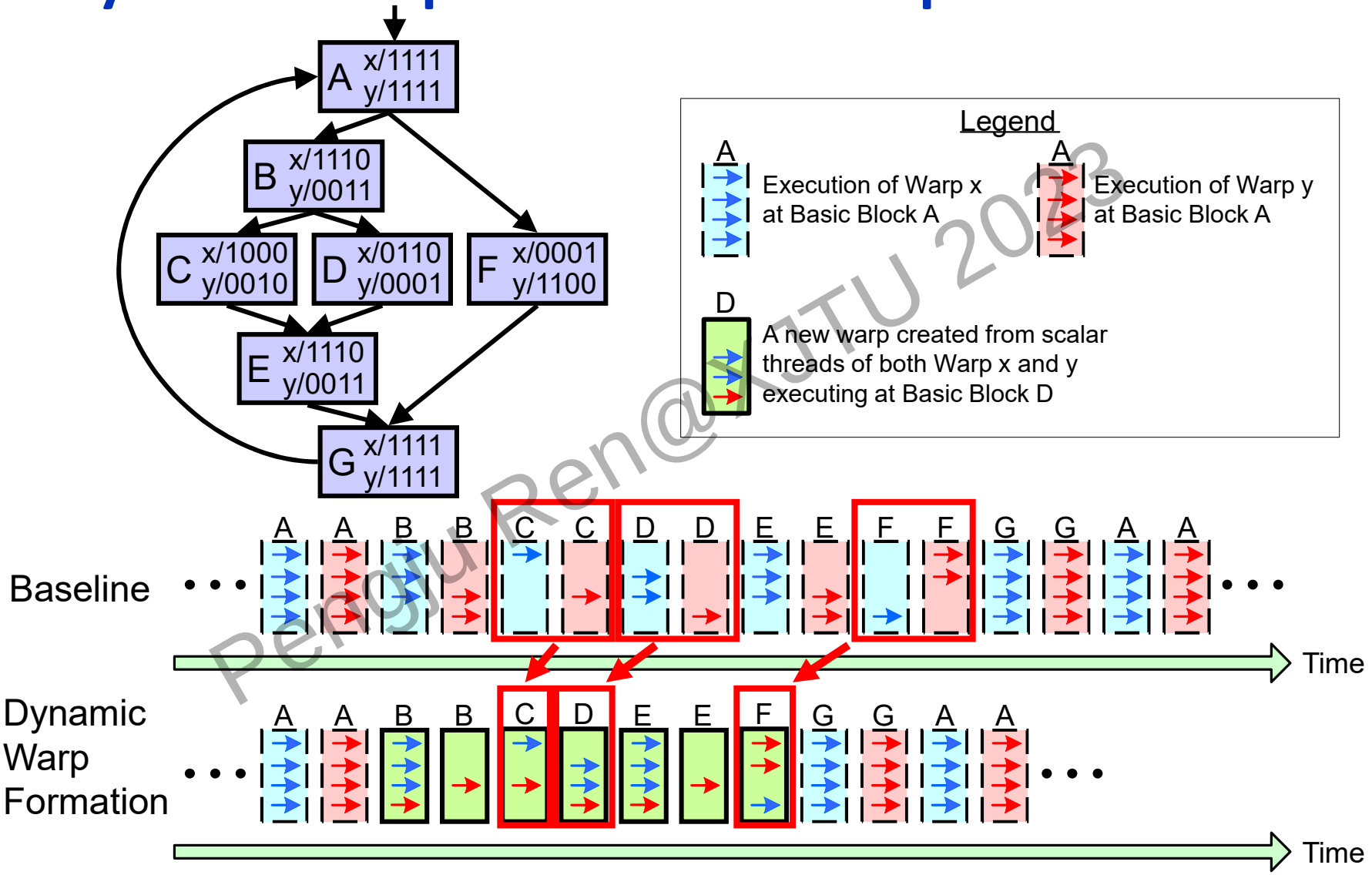
SIMT stack is used in current GPUs to serialize execution of threads following different paths within a given warp.

Dynamic Warp Formation/Merging

- **Idea: Dynamically merge threads executing the same instruction (after branch divergence)**
- **Form new warps from warps that are waiting**
  - **Enough threads branching to each path enables the creation of full new warps**



Dynamic Warp Formation Example

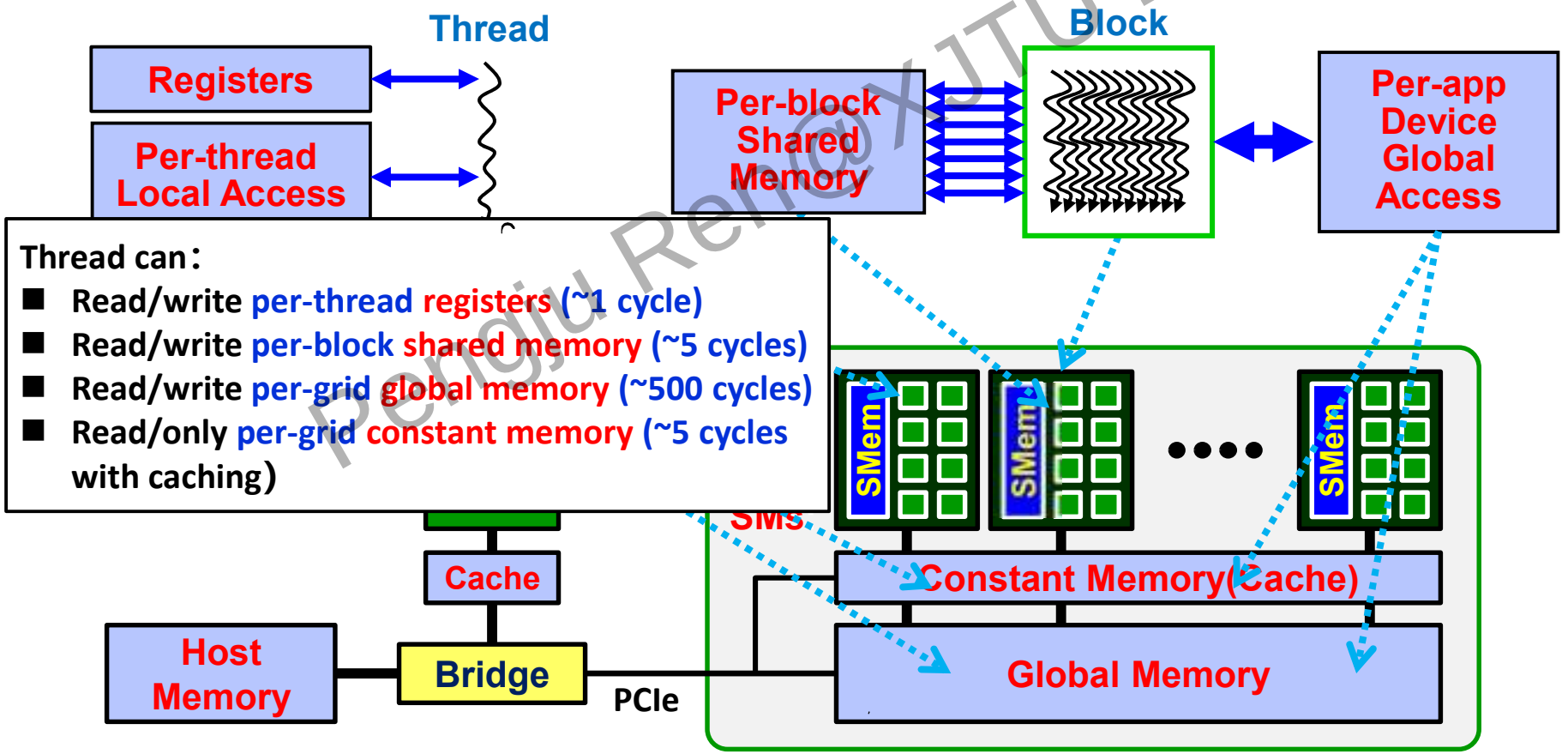


GPU Memory Hierarchy

local Memory (private memory)
– No cross-thread sharing
– Small, fixed size memory (can be used for constants)
– Multi-bank implementation (can be in global memory)

Shared memory (share data within a thread block)
– Small, fixed size memory (16K-64K/SM)
– Banked for high bandwidth
– Fed with address coalescing unit (ACU) + crossbar
• ACU can buffer/coalesce requests

global memory (Shared for the device)
– Large shared memory
– Will suffer also from memory divergence



Programmer View of CUDA Memories

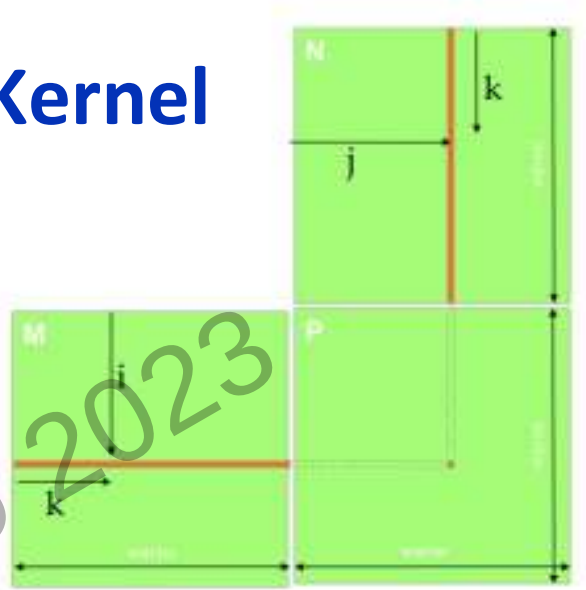
Variable declaration	Memory	Scope	Lifetime
`int LocalVar`	Register	Thread	Thread
`__device__ __shared__ int SharedVar`	Shared	block	block
`__device__ int GlobalVar`	Global	App.	App
`__device__ __constant__ int ConstantVar;`	Constant	App.	App

- Each thread can:
- – read/write per-thread **registers** (*~1 cycle*)
- – read/write per-grid **global memory** *(~500 cycles*)
- – read/write per-block **shared memory** *(~5 cycles*)
- – read/only per-grid **constant memory** *(~5 cycles with caching*)

  (will be introduced later in Example 5: Conv)

Example 4 : Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
 {
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width))  {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

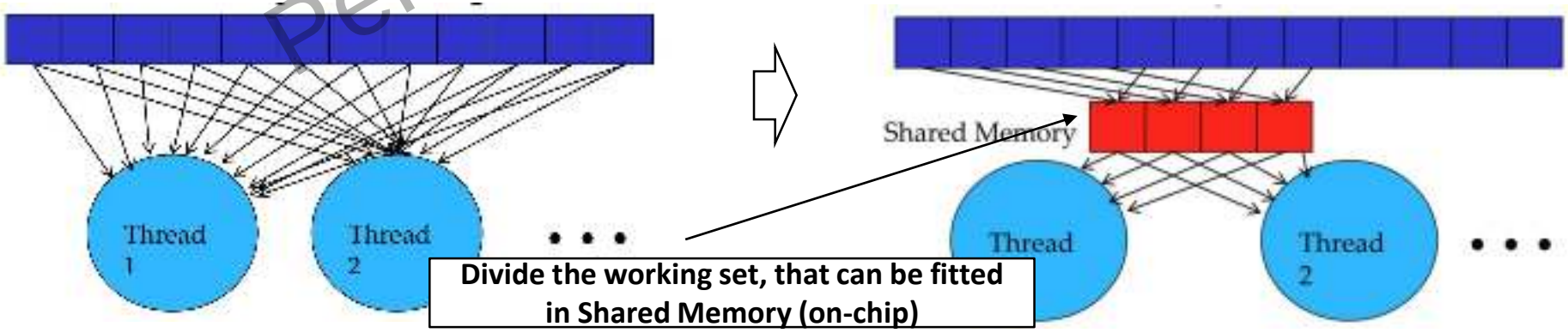


Global memory access

- Each threads access global memory and 2 FLOP for each pair of elements
  - –for elements of M and N
  - –4B each, or 8B per pair

Naïve Matrix Multiplication Kernel

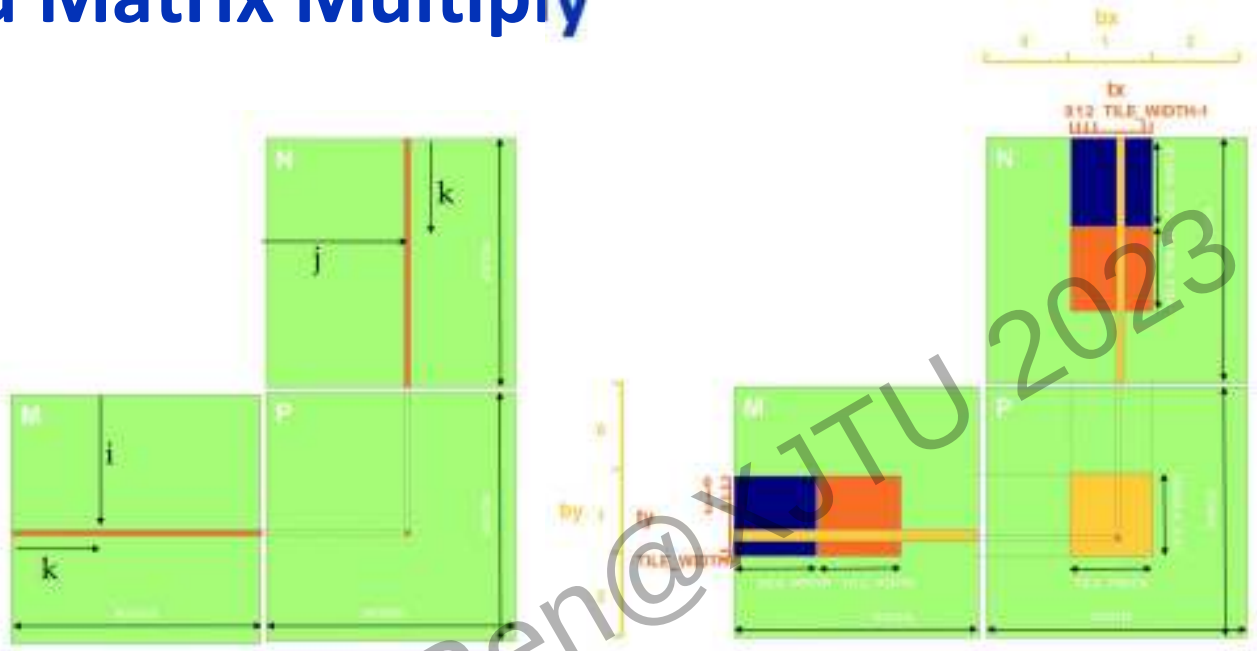
- In a GPU, only *threads in a block* can use **shared memory**
- To avoid Global Memory bottleneck, *tile the input data* to take advantage of Shared Memory:
  - Partition data into subsets (tiles) that fit into the (smaller but faster) **shared memory**
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, *using multiple threads to exploit memory-level parallelism*
    - Performing the computation on the subset from shared memory; each thread can efficiently access any data element
    - Copying results from shared memory to global memory



Tiled Matrix Multiply using Shared Memory

- In terms of distance from the Shared Memory, it is similar to L1 cache
- Like Cache, shared memory is on chip and use the same physical resources
- Unlike cache, shared memory does not necessarily hold a copy of data that is also in main memory (exclusive)
- Shared memory requires *explicit data transfer instructions into locations* in the shared mem, whereas cache doesn't. In another words, Programmer controls shared memory contents (therefore also term it scratchpad)

Tiled Matrix Multiply



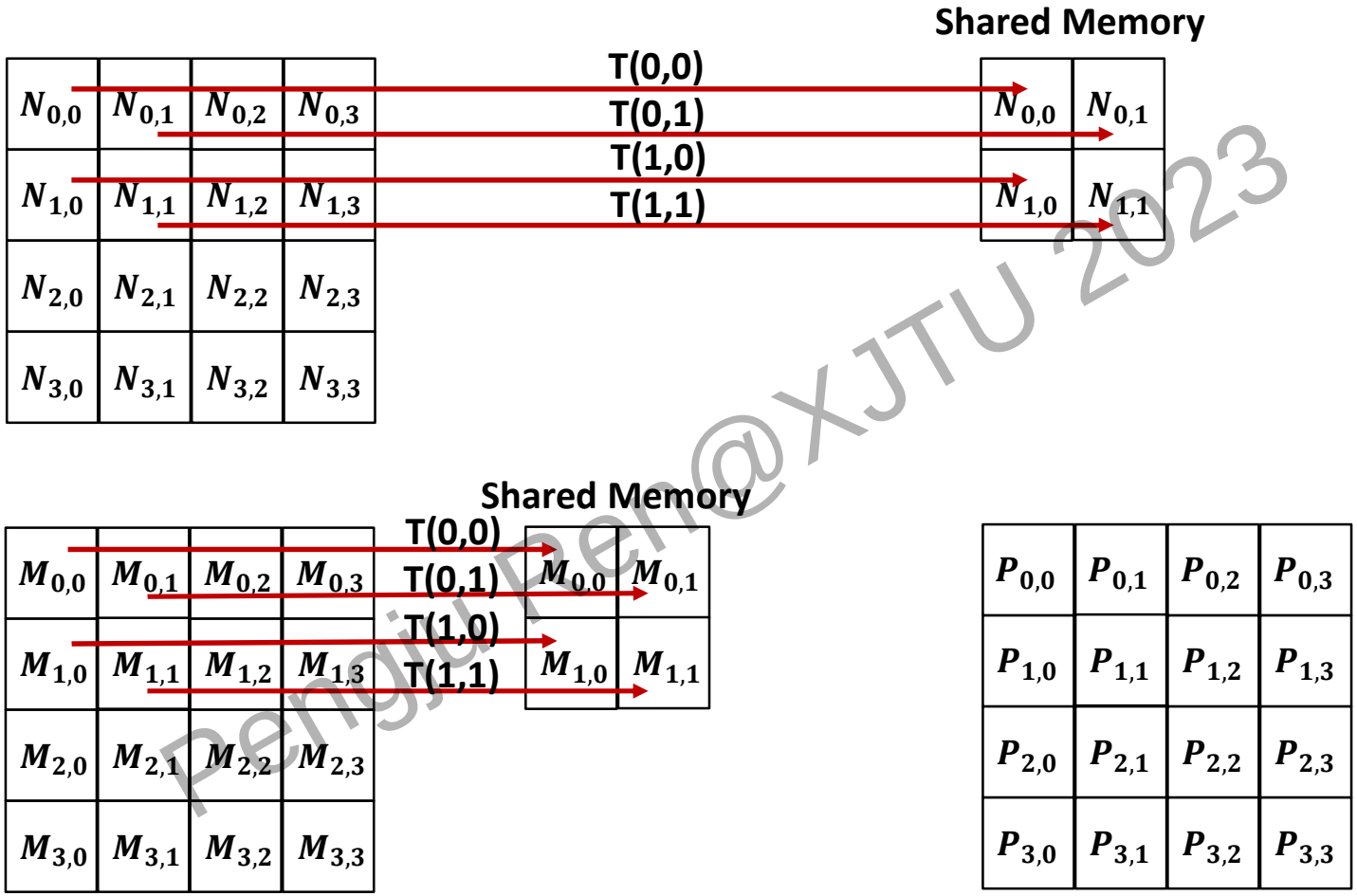
Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of M and N. For each tile:

– Phase 1: Load tiles of M & N into share memory

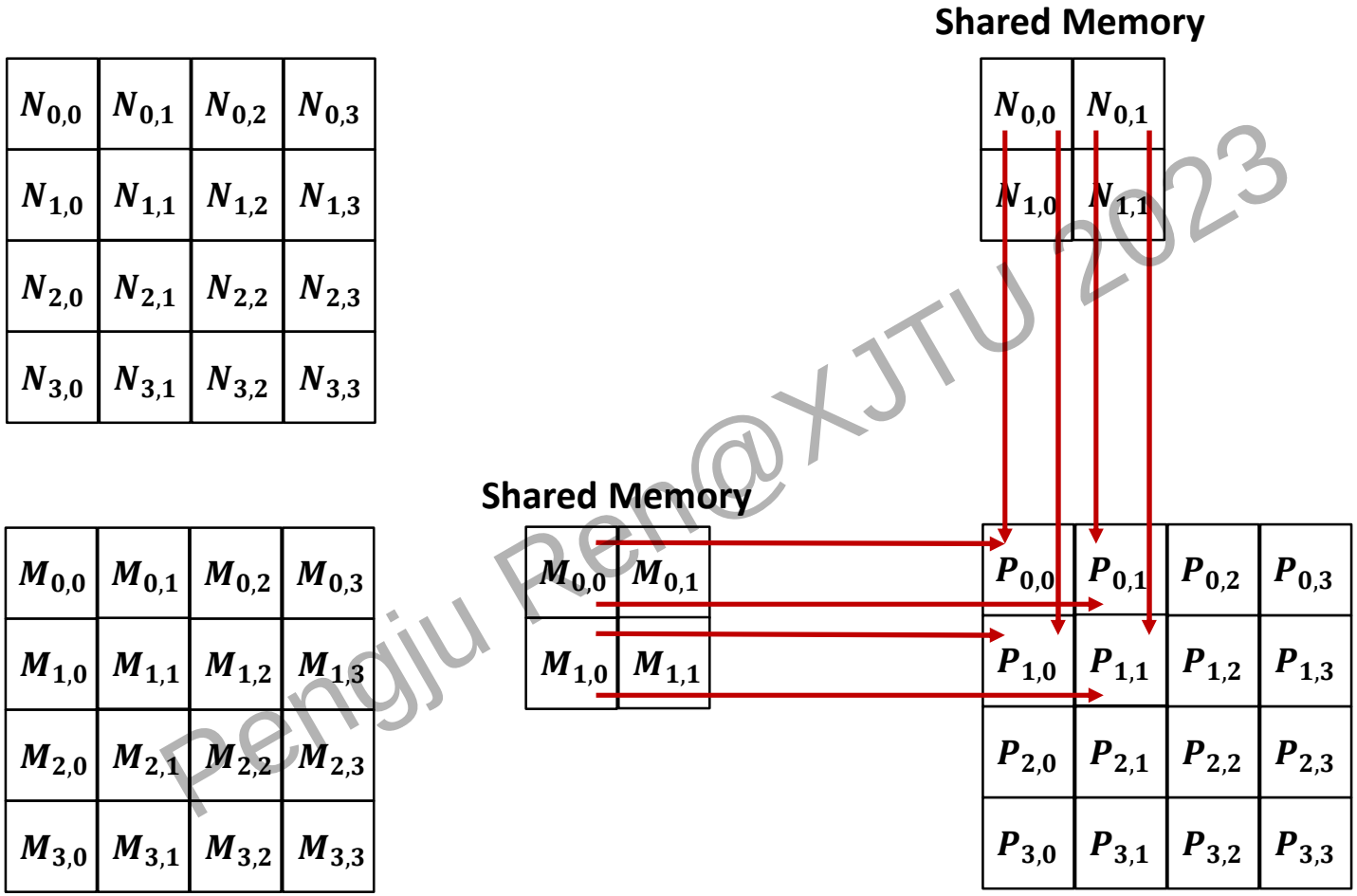
– Phase 2: Calculate partial dot product for tile of P

```
__global__
void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

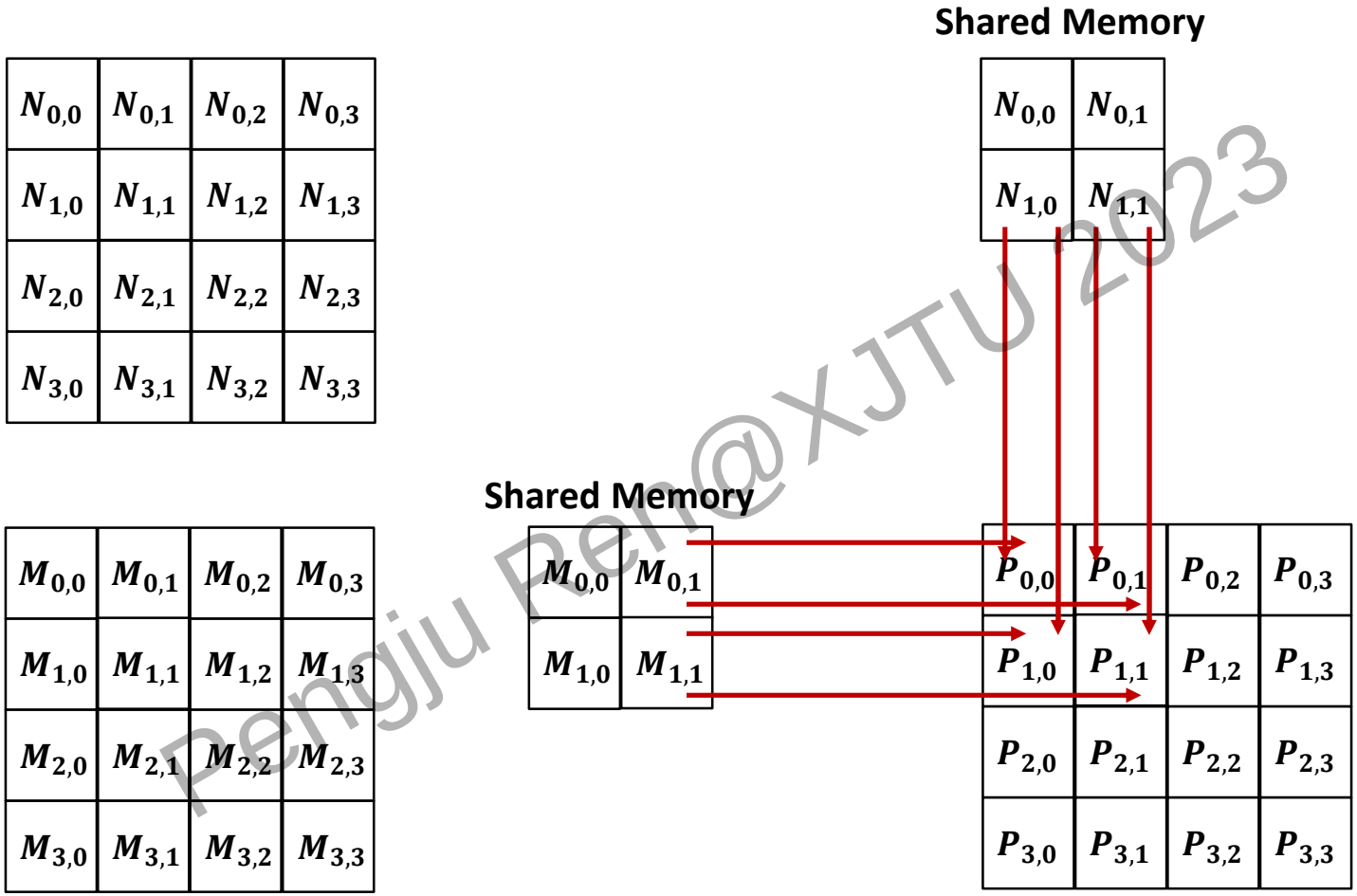
Loading a Tile



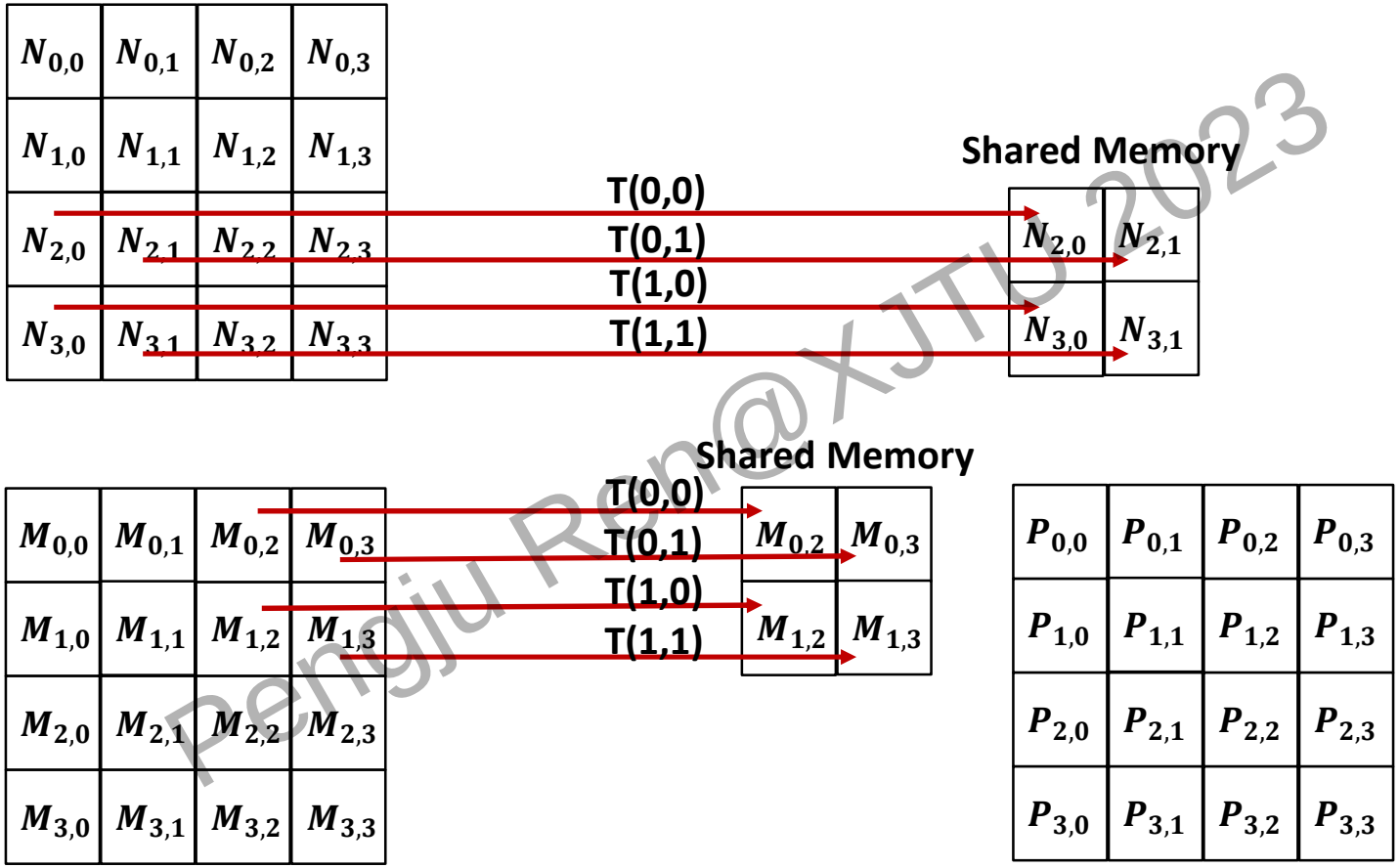
Loading a Tile (Step1)



Loading a Tile (Step2)



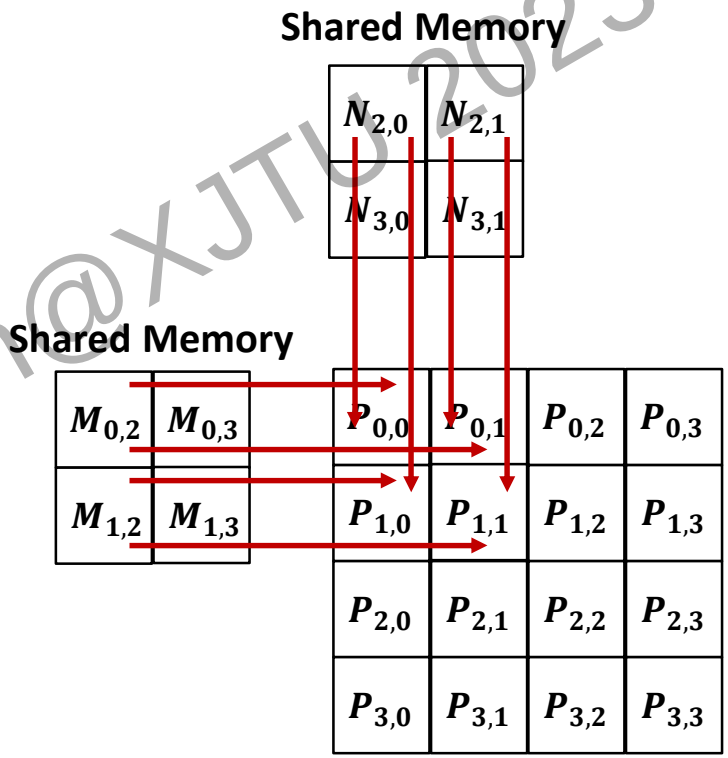
Loading a Tile (Step3)



Loading a Tile (Step4)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

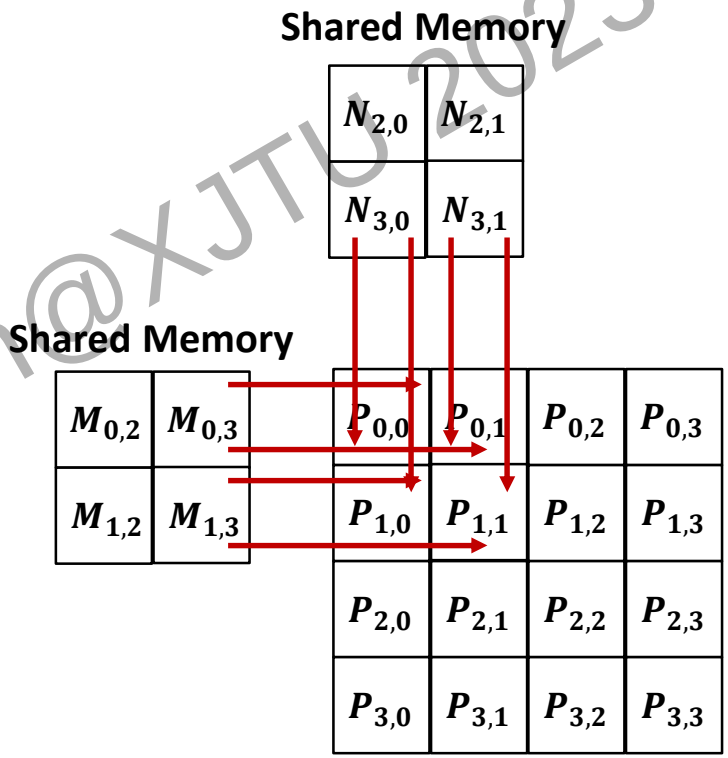
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Loading a Tile (Step5)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Phase 1: loading a Tile

- All threads in a block participate – Each thread loads one M element and one N element in basic tiling code

2D indexing for Tile **0**:

$$M[Row][tx]$$
$$N[ty][Col]$$

2D indexing for Tile **1**:

$$M[Row][\mathbf{1} * TILE\_WIDTH + tx]$$
$$N[\mathbf{1} * TILE\_WIDTH + ty][Col]$$

2D indexing for Tile $\boldsymbol{q}$:

$$M[Row][\boldsymbol{q} * TILE\_WIDTH + tx]$$
$$N[\boldsymbol{q} * TILE\_WIDTH + ty][Col]$$

$$M[Row * TILE\_WIDTH + \boldsymbol{q} * TILE\_WIDTH + tx]$$
$$N[(\boldsymbol{q} * TILE_{WIDTH} + ty) * TILE\_WIDTH + Col]$$

Phase2 : Compute partial product

To perform the k^{th} step of the product within the tile:

$subTileN[k][tx]$
$subTileM[ty][k]$

Then …

How can a thread know …

– That another thread has *finished its part of the tile*?

– Or that another thread has *finished using the previous tile*?

Synchronization is necessary !
Using a barrier to wait for threads to “Catch Up”

Example 5: Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the P element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  // Loop over the M and N tiles required to compute the P element
  // The code assumes that the Width is a multiple of TILE_W
  for (int q = 0; q < Width/TILE_WIDTH; ++q) {
      // Collaborative loading of M and N tiles into shared memory
      subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx];
      subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
      __syncthreads();
      for (int k = 0; k < TILE_WIDTH; ++k)
          Pvalue += subTileM[ty][k] * subTileN[k][tx];
      __syncthreads();
   }
  P[Row*Width+Col] = Pvalue;
}
```

Shared Memory holds subTile M&N

Global memory access

Barrier Synchronization

- To ensure all elements of a tile are loaded
- To ensure certain computation on elements is complete

Local memory access

Naïve v.s Tiled Matrix Multiplication Kernels

A GPU:

Computation Capabil

Naïve Matrix Multipli

- A thread performs
  therefore *4B/FLOP*
- So *150GB/s* can su

Tiled (16x16) Matrix

- Two *16x16* tiles us
  2x16x16x4-Byte/(1
- So *150GB/s* can su

Tiled (32x32) Matrix

- Two *32x32* tiles us
  2x32x32x4-Byte/(3
- So *150GB/s* can support ***1,200GFLOPs***!

- **Hardware has constrains, for example Maxwell GPGPU:**
  - **64kB Shared Memory per SM (Stream Processors)**
  - **Max. of 2048 threads/SM**
- **16x16 Tile (256 threads/block):**

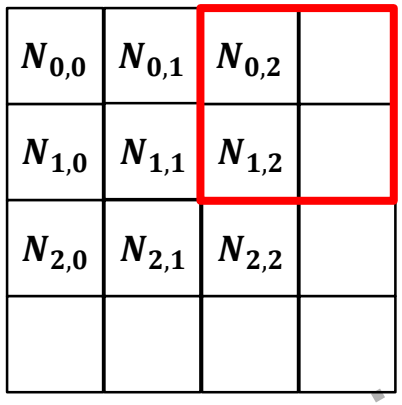
Each thread block uses 2x(16x16)x4B=2kB of shared memory, Size of the Shared Memory limits active blocks to 32 (64kB/2KB=32), and # of threads limits # blocks to 8 (2048/256=8).

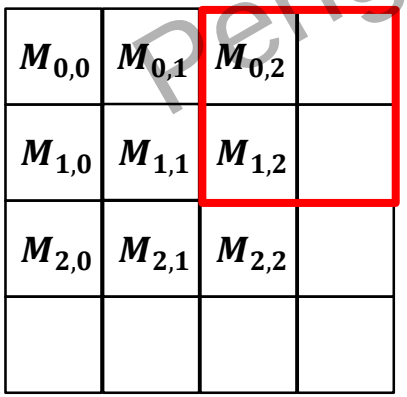
- **32x32 Tile (1024 threads/block):**

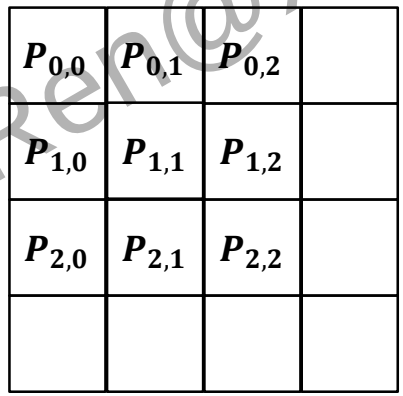
Each thread block uses 2x(32x32)x4B=8kB of shared memory, Size of the Shared Memory limits active blocks to 8 (64kB/8KB=8), and # of threads limits # blocks to 2 (2048/1024=2).

Handle boundary Conditions

Recap： address index of 2D-Array, we need handle boundary conditions with padding







The bound for q implicitly assumes that Width is a mult of TILE_WIDTH.
We need to round up.

Tests for threads outside of P

Tiled Matix Multiplication Kernel (Improved)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the P element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  // Loop over the M and N tiles required to compute the P element
  // The code assumes that the Width is a multiple of TILE_WIDTH!
    for (int q = 0; q < (Width-1)/TILE_WIDTH+1; ++q) {
     // Collaborative loading of M and N tiles into shared memory
    if (Row < Width && q*TILE_WIDTH+tx < Width) {
       subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx]; }
    else { subTileM[ty][tx] = 0; }
    if (q*TILE_WIDTH+ty < Width && Col < Width ) {
       subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col]; }
    else { subTileN[ty][tx] = 0; }
    __syncthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
          Pvalue += subTileM[ty][k] * subTileN[k][tx];
     __syncthreads();
   }
    if (Row < Width && Col < Width) {P[Row*Width+Col] = Pvalue; }
}
```

The bound for q implicitly assumes that Width is a mult of TILE_WIDTH. We need to round up.

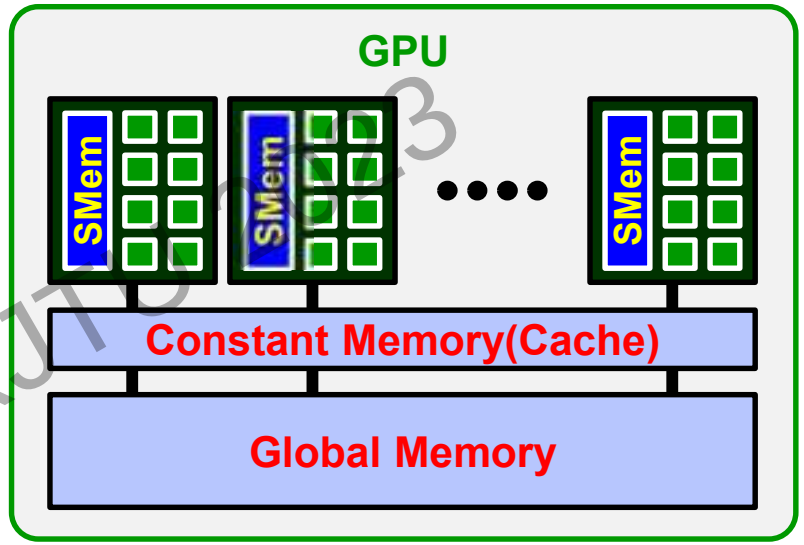
Tests for threads outside of P

Example 6 : Convolution

- **An important parallel computation pattern**
  - Widely used in signal, image and video processing
  - An array operation where each output data element is a weighted sum of a collection of neighboring input elements, The **weights** used in the weighted sum calculation are defined by an input mask array, commonly referred to as the **convolution filters** (kernel)
  - Critical component of Neural Networks and Deep Learning
- **Important GPU technique**
  - Taking advantage of cache memories **(Scratchpad)**

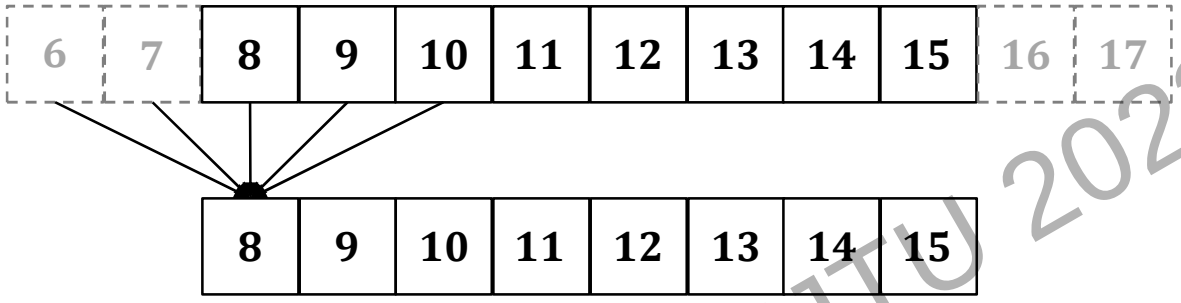
Constant Cache vs. Cache

- Weights are not changed during grid execution, can be stored in Constant Memory.
- Constant cache is a special cache for constant data that *will not be modified* during kernel execution by a grid during kernel execution.
- Constant cache can be accessed with higher throughput than L1 cache for some common patterns




```
// global variable, outside any kernel/function
__constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
// Initialize Mask float Mask[MASK_WIDTH][MASK_WIDTH]
for(unsigned int i = 0; i < MASK_WIDTH * MASK_WIDTH; i++) {
    Mask[i] = (rand() / (float)RAND_MAX); if(rand() % 2)
    Mask[i] = - Mask[i] }
cudaMemcpyToSymbol(Mc, Mask,MASK_WIDTH*MASK_WIDTH*sizeof(float));
```

1D Convolution Example



Tile_WIDTH is 8

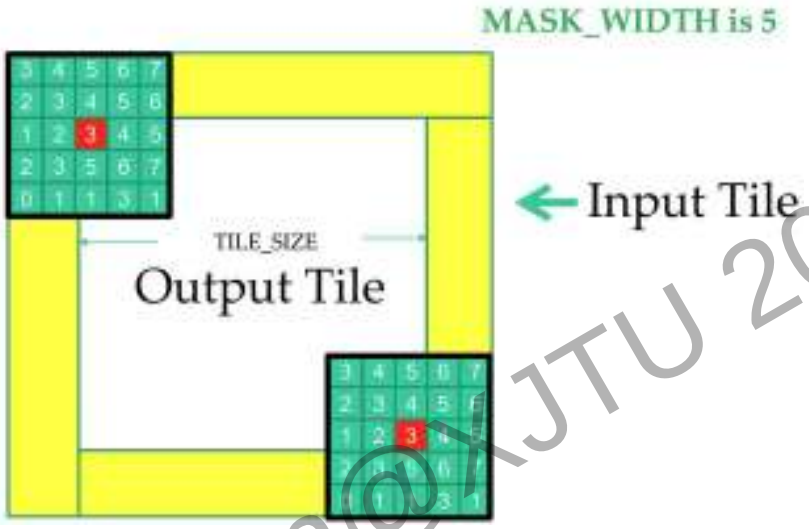
Filter_WIDTH is 5

A simple way to calculate tiling benefit 8+(5-1)=12 unique elements of input array, and 8x5 global memory accesses potentially replaced by shared memory accesses, which gives a bandwidth reduction of 40/12=3.3

In general, load (TILE_WIDTH + MASH_WIDTH-1) elements from global memory to shared memory, and replaced (TILE_WIDTH*MASK_WIDTH) global memory accesses, which gives a bandwidth reduction of:

(TILE_WIDTH *MASK_WIDTH)/(TILE_SIZE+ MASH_WIDTH-1)

2D Convolution Example



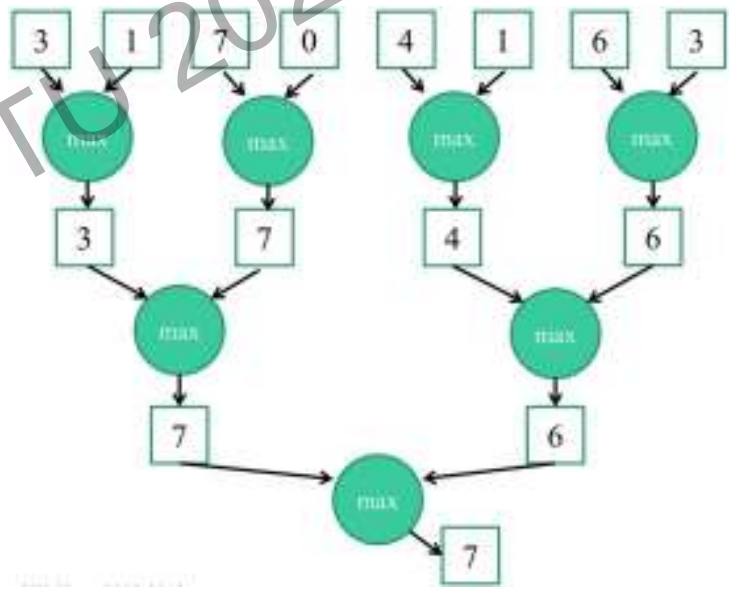
Loading input tile requires (8+5-1)^2=144 unique elements, and calculation of each output requires 5^2=25 input elements, One 8x8 output needs 8x8x25=1600 global memory accesses for computing output tile are converted to shared memory accesses. Bandwidth reduction of 1600/144=11.1x

In general, load (TILE_WIDTH + MASH_WIDTH-1)^2 elements from global memory to shared memory, and replaced (TILE_WIDTH*MASK_WIDTH)^2 global memory accesses, which gives a bandwidth reduction of:

(TILE_WIDTH *MASK_WIDTH)^2/(TILE_SIZE+ MASH_WIDTH-1)^2

Example 7：Reduction

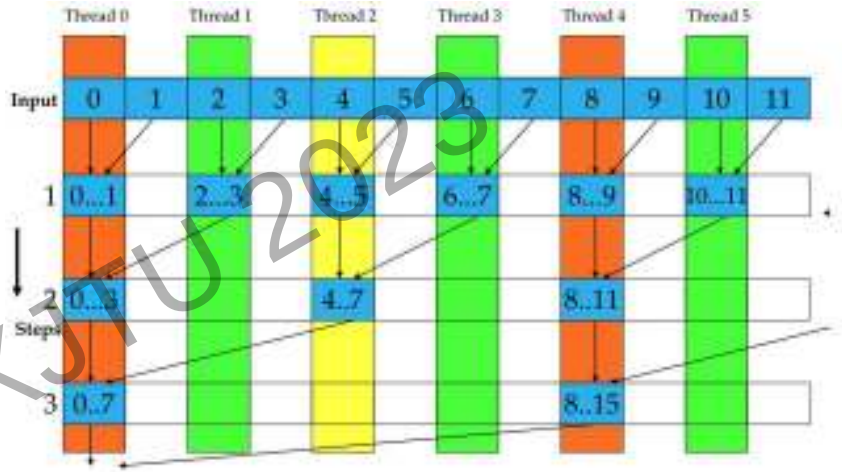
- Reduce a set of inputs to a single value using a binary operator, such *as sum, product, minimum, maximum*, or a user-defined reduction operation – must be associative and commutative
- Examples: Google and Hadoop MapReduce
- no required order for processing the values (operator is *associative and commutative*), so：
  - partition the data set into smaller chunks
  - have each thread to process a chunk, and
  - use a tree to compute the final answer



Parallel Reduction

Read block of 2M values into shared memory.

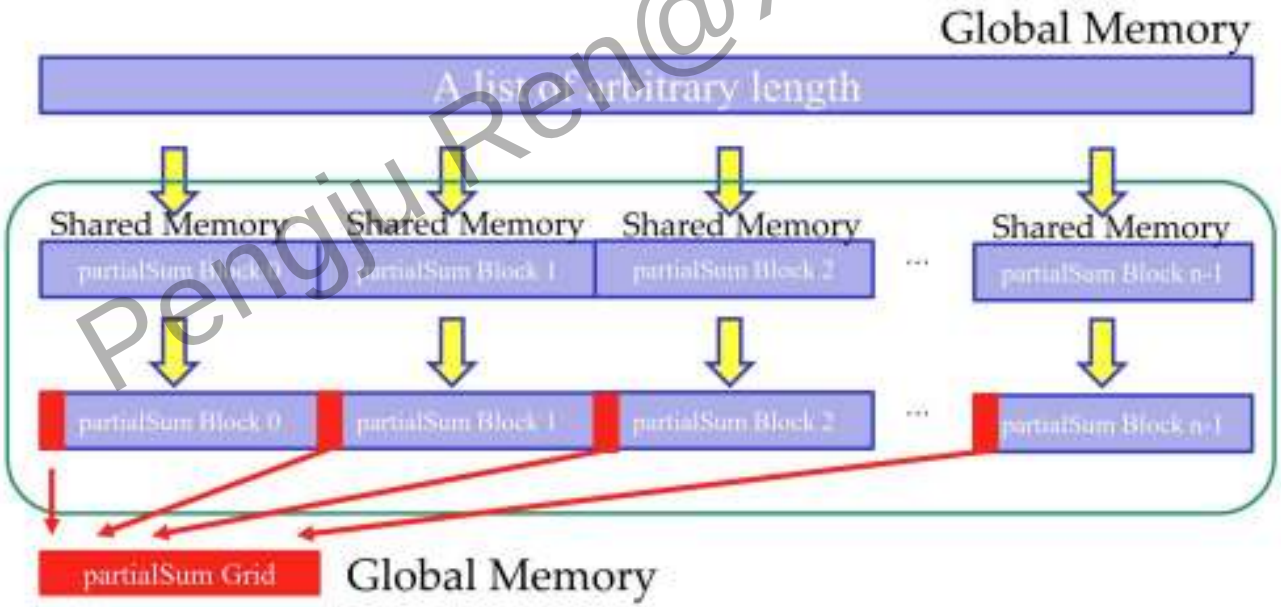
- **For each of log(2M) steps, combine two values per thread in each step, write result to shared memory, and halve the number of active threads.**
- **Write final result back to global memory**




```
// Stride is distance to the next value being
// accumulated into the threads mapped position
// in the partialSum[] array
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2)
 {
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t]+= partialSum[2*t+stride];
 }
```

Analysis of Execution Resources

- **All threads active in the first step. in all subsequent steps, two control flow paths:**
  - perform addition, or do nothing.
  - Doing nothing still consumes execution resources (Why?)
- **At most half of threads perform addition after first step**
  - all threads with odd indices disabled after first step
  - ***poor resource utilization***, but no divergence

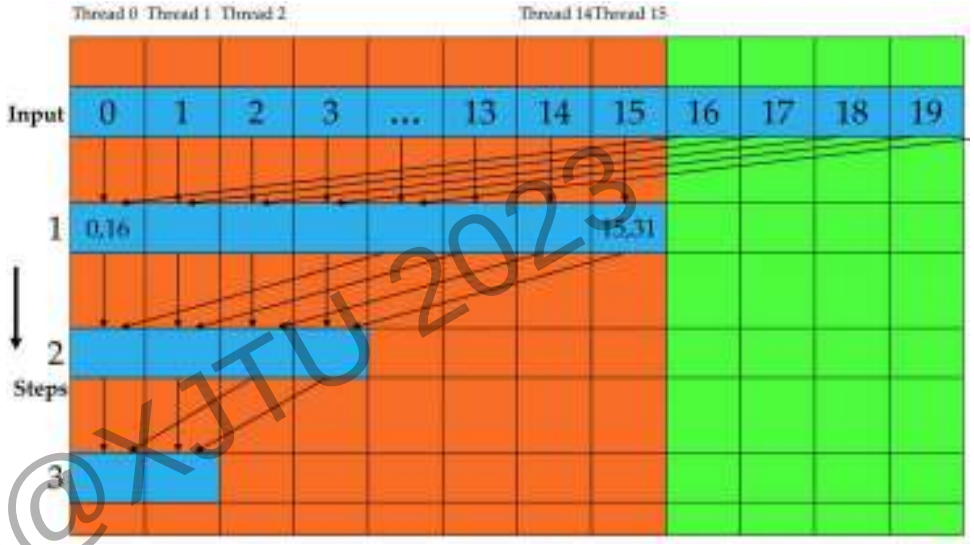


Copy back to host and host to finish the work.

Improved Parallel Reduction

Let's try this approach:

- **in each step, compact the partial sums into the first locations in the partialSum array**




```
for (unsigned int stride = blockDim.x;  stride >= 1; stride /= 2)
{
    __syncthreads();
    if (t < stride)  partialSum[t] += partialSum[t+stride];
}
```


Given 1024 threads, no branch divergence in the first six steps (32 threads/wrap):
– 1024, 512, 256, 128, 64, and 32
consecutive threads active; threads in each warp either all active or all inactive,
Last six steps have one active warp (branch divergence for last five steps).

Example 8: Histogramming (Atomic Operations)

- **Basic histogram algorithm using atomic operations**
- **To understand atomic operations**
  - **Read-modify-write** in parallel computation
  - A primitive form of **"critical regions"** in parallel programs
  - Use of atomic operations in CUDA
  - Why atomic operations reduce memory system throughput
  - How to avoid atomic operations in some parallel algorithms

A Common Arbitration Pattern

Example： Multiple customers booking air tickets

- **Each**
  - Brings up a flight seat map
  - Decides on a seat
  - Update the seat map, mark the seat as taken
- **A bad outcome**
  - Multiple passengers ended up booking the same seat

Why should this happen ? (Data Race)

Read-Modify-Write (Read-after-Write) Operations

If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?

The answer may vary due to data races.

To avoid data races, you should use atomic operations

Timing Scenario

t	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

t	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

Mem[x] = 2 after the sequence execution

t	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

t	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Mem[x] = 1 after the sequence execution

Atomic Operations are needed When Threads Write to the same Location

- When multiple threads may write to the same memory location, the program may need *atomic operations*
- Many ISAs offer *synchronization primitives*, and atomicity enforced by microarchitecture
- *Atomic Operations* prevent Interleaving, the section of code is executed atomically with respect to one another when used on the same address

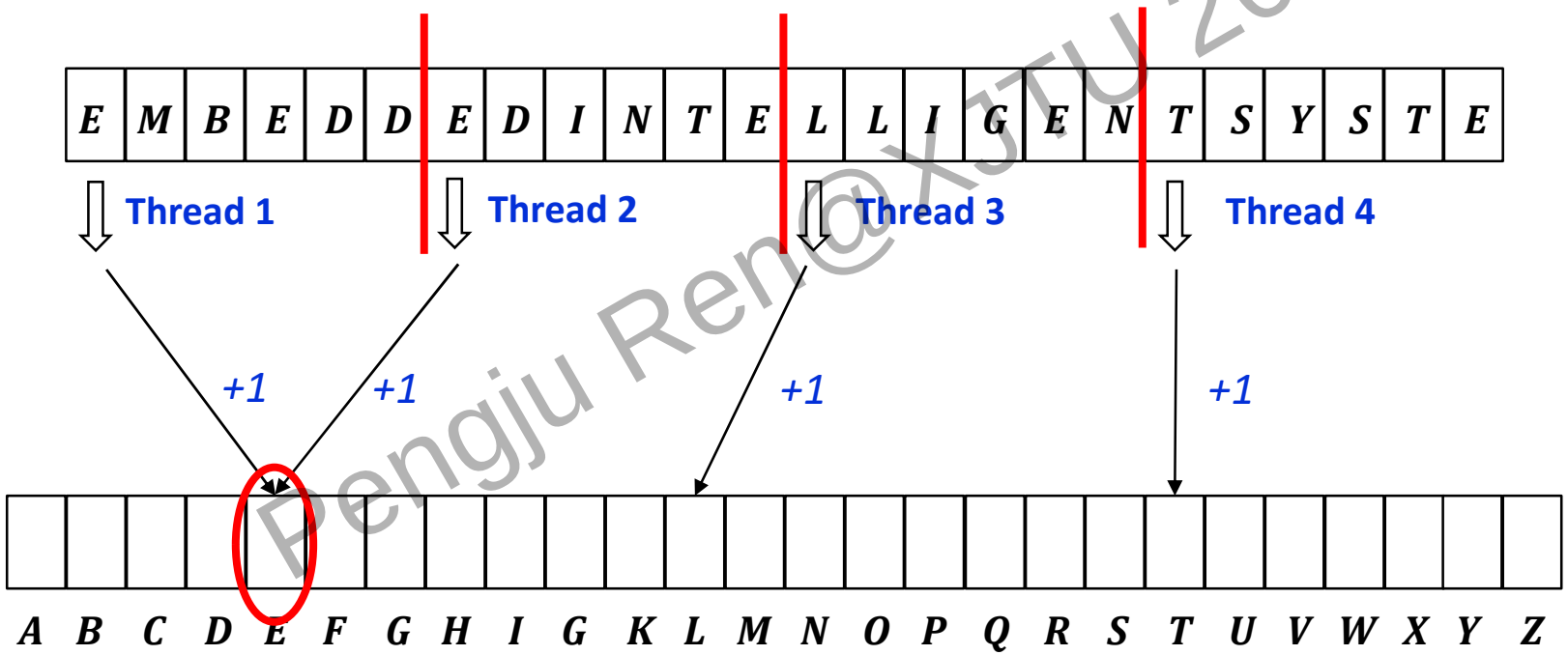
Function calls that are translated into single ISA instructions (a.k.a. *intrinsics*)

– Atomic *add, sub, inc, dec, min, max, exch* (exchange), *CAS* (compare and swap)

Example Atomic *add* : *int atomicAdd(int* **address**, int **val**);*

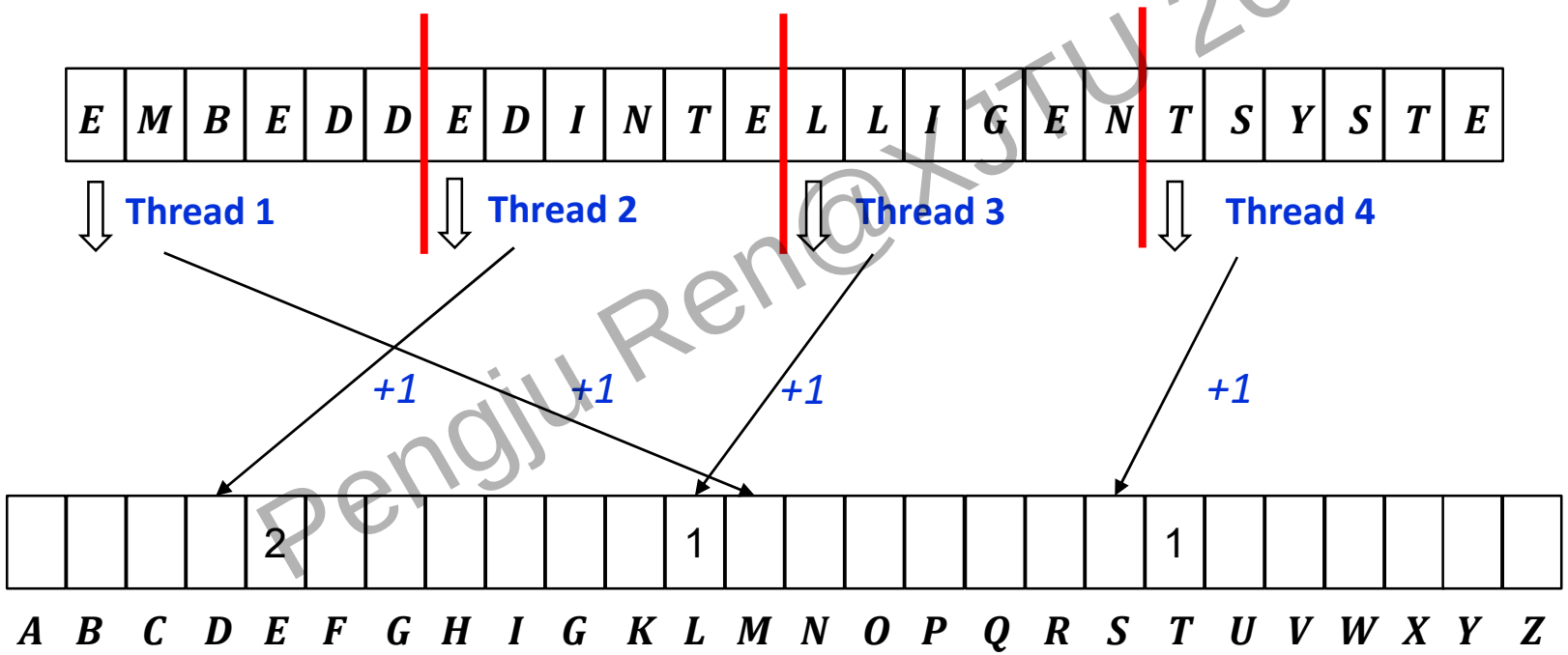
A Histogram Example (1)

In sentence "*Embedded Intelligent System*" build a histogram of frequencies of each letter



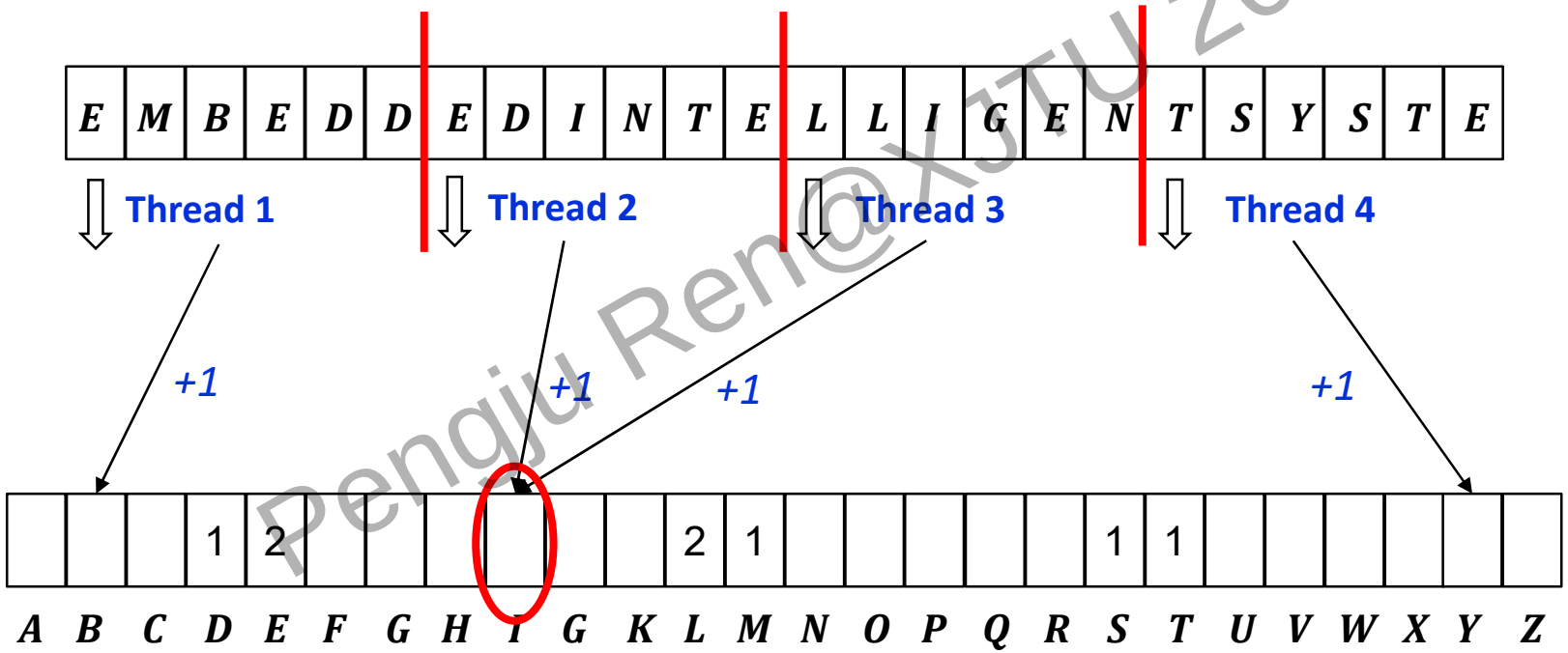
A Histogram Example (2)

In sentence "*Embedded Intelligent System*" build a histogram of frequencies of each letter



A Histogram Example (3)

In sentence "*Embedded Intelligent System*" build a histogram of frequencies of each letter

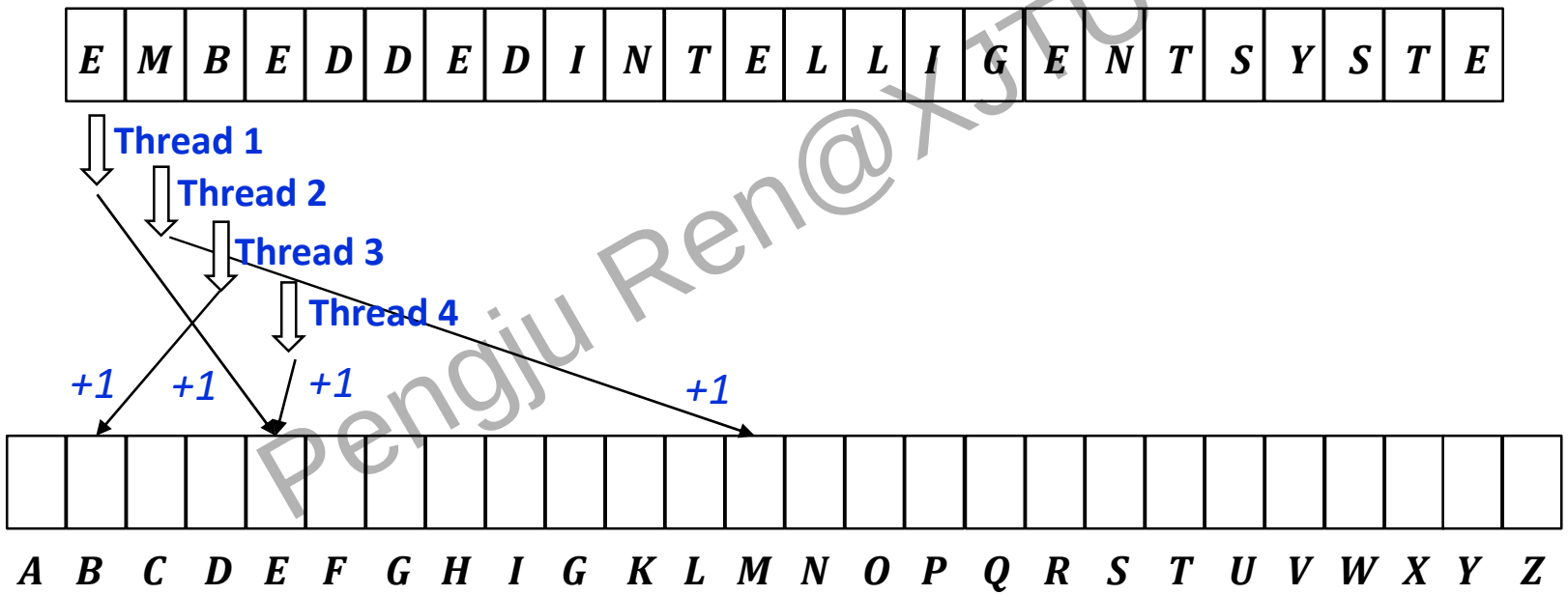


A Histogram Example (4) – A better Approach

Reads from the input array are not coalesced

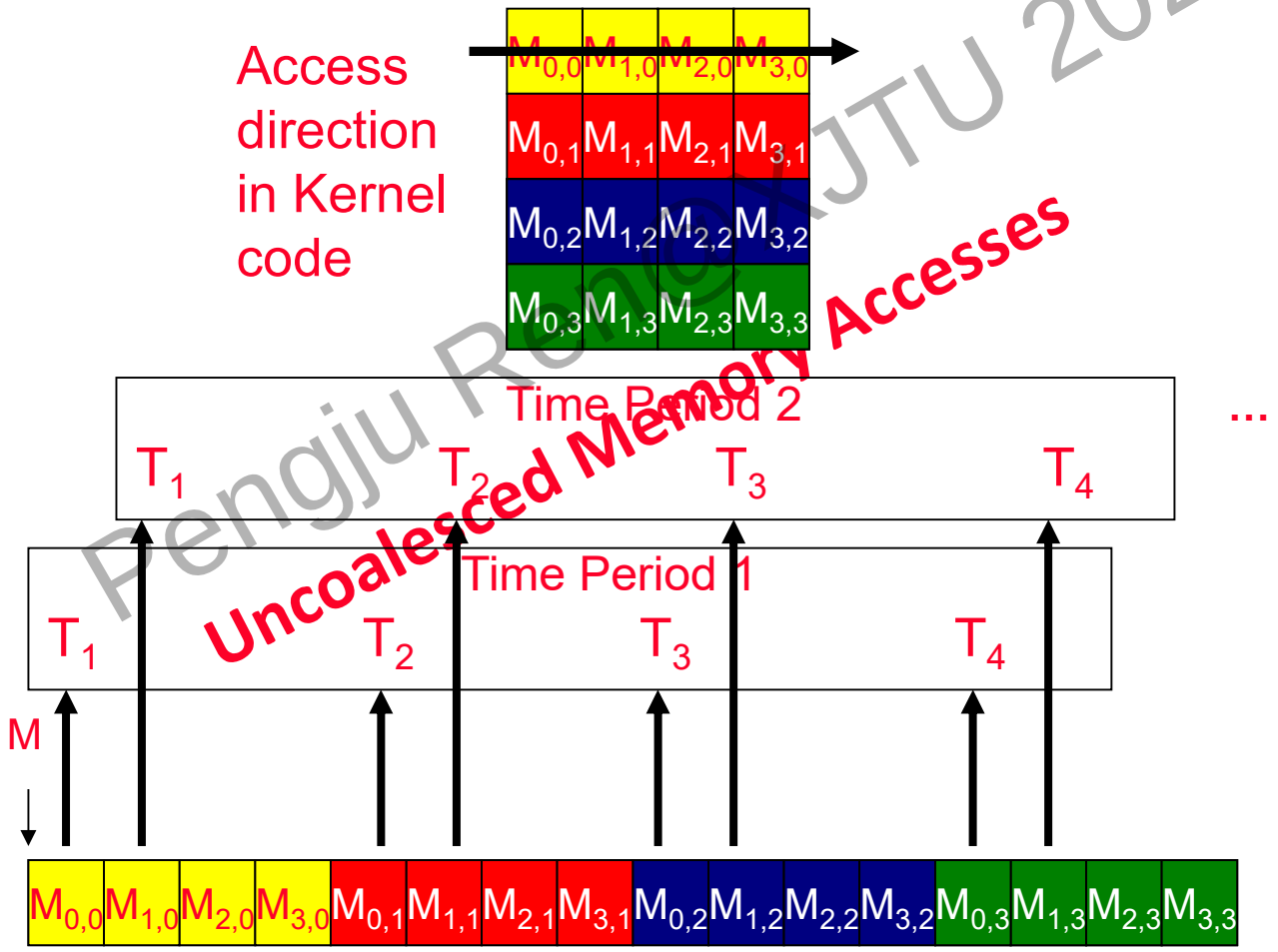
– Assign inputs to each thread in a strided pattern

– Adjacent threads process adjacent input letters



Address Coalescing (M matrix access)

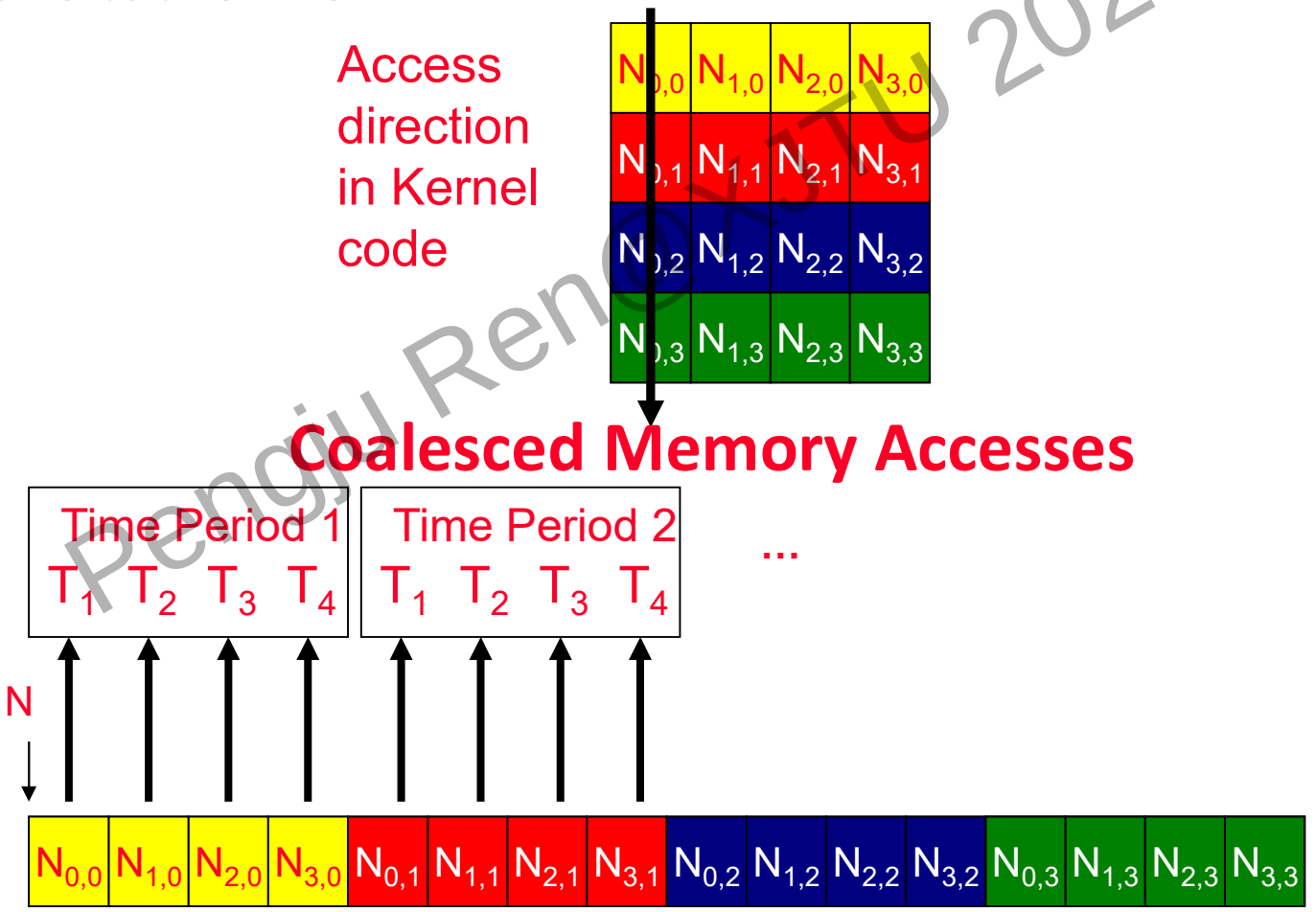
Merge many work-items requests into single Cache block request
When accessing global memory, we want to make sure that
concurrent threads access nearby memory locations



Address Coalescing (N matrix access)

Merge many work-items requests into single Cache block request

Peak bandwidth utilization occurs when all threads in a warp access one cache line



Return to Histogram Kernel

The kernel receives a pointer to the input buffer

Each thread process the input in a strided pattern

```
__global__
void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // All threads in the grid collectively handle
    // blockDim.x * gridDim.x consecutive elements
    while (i < size)
        { atomicAdd( &(histo[buffer[i]]), 1);
          i += stride;
     }
}
```

Atomics in Shared Memory Requires Privatization

Create private copies of the histo[] array for each thread block

```
__global__
void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    __shared__  unsigned int histo_private[256];
    // warning: this will not work correctly if there are fewer than 256 threads!
    if (threadIdx.x < 256)  histo_private[threadIdx.x] = 0;
    __syncthreads();
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        Add( &(histo_private[buffer[i]), 1);
        i += stride;
    {
    // wait for all other threads in the block to finish
    __syncthreads();
    if (threadIdx.x < 256)
        atomicAdd( &(histo[threadIdx.x]), private_histo[threadIdx.x] );
 }
```

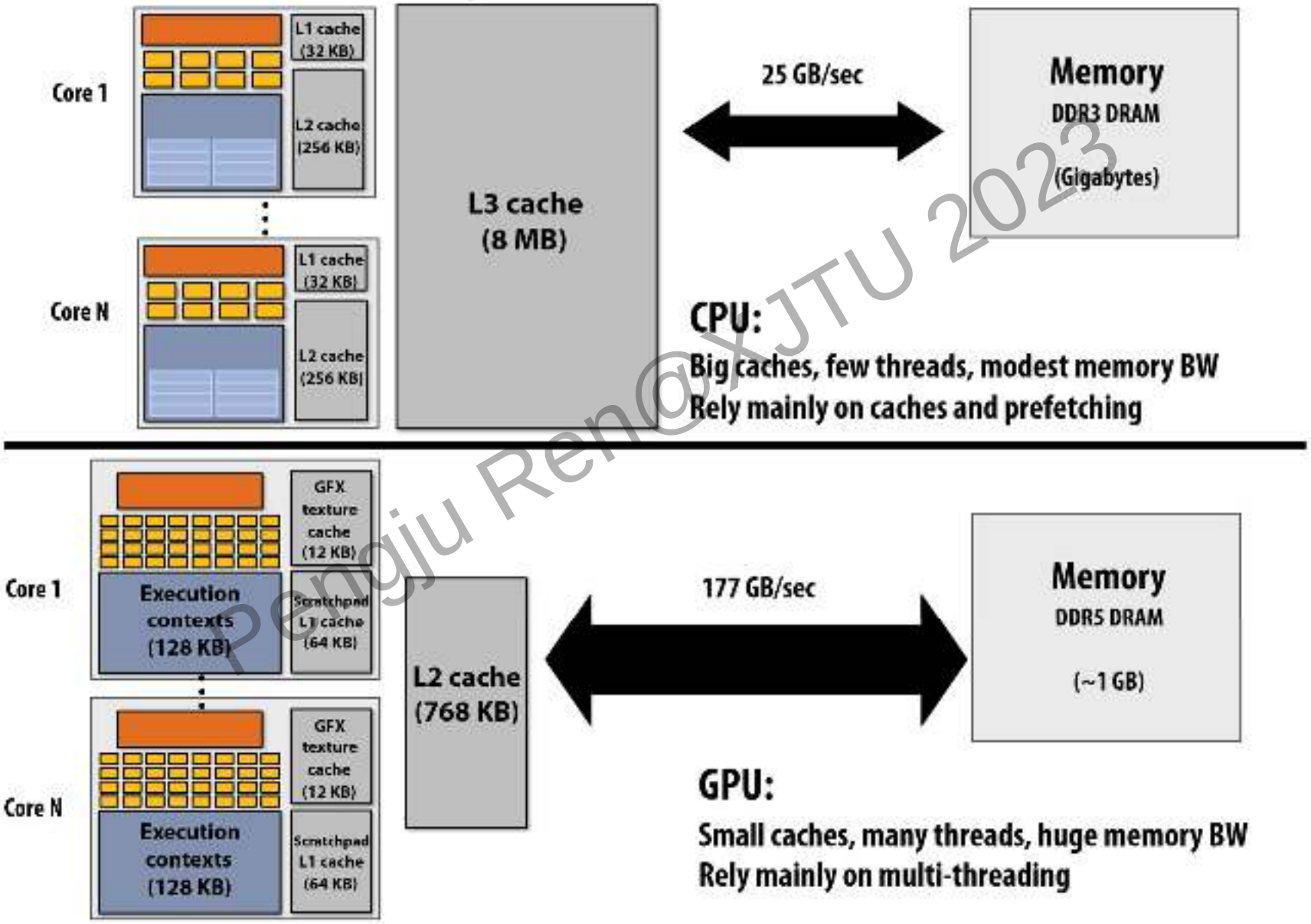
Update private copies of the histo[]

Reduction of Muliple histo[] to a single one

More on Privatization

- Privatization is a powerful and frequently used techniques for parallelizing applications
- The operation needs to be associative and commutative (*Max, Min,Add,Multiply, etc.)*
- The histogram size needs to be small to fits into shared memory

CPU v.s GPU memory hierarchies



Summary

- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are **bandwidth bound**
- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales

Next Lecture：Memory Hierarchy and Programming