https://hao-ai-lab.github.io/cse234-w25/

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

MCQ Time

What is the arithmetic intensity for the following function:

```
# A, B are 2-D matrices of shape [2,2]
func(matrix A, matrix B):
    Load A
    Load B
    C = matmul(A,B)
```

A. 0.334
B. 2
C. 1
D. 1.334

Important Notes

FLOPs of matmul: A x B

- A: (m, n)
- B: (n, p)
- Result: (m, p)
- Flops: 2mnp

Which of the following Tensor manipulations cannot benefit from strided representation

A. Broadcast_to

B. Slice

C. Reshape

D. Permute dimensions

E. Transpose

F. contiguous

G. indexing like t[:, 1:5]

If we have tensor of shape [2,9,1] stored contiguous in memory following row Major, what is its strides?

A. (9,1,1)

B. (2,9,1)

C. (1,9,2)

D. (9,9,9)

Which of the following is True for *Cache Tiling* in Matmul

A. It saves memory allocated in Cache

B. It reduce the memory movement between Cache to Register

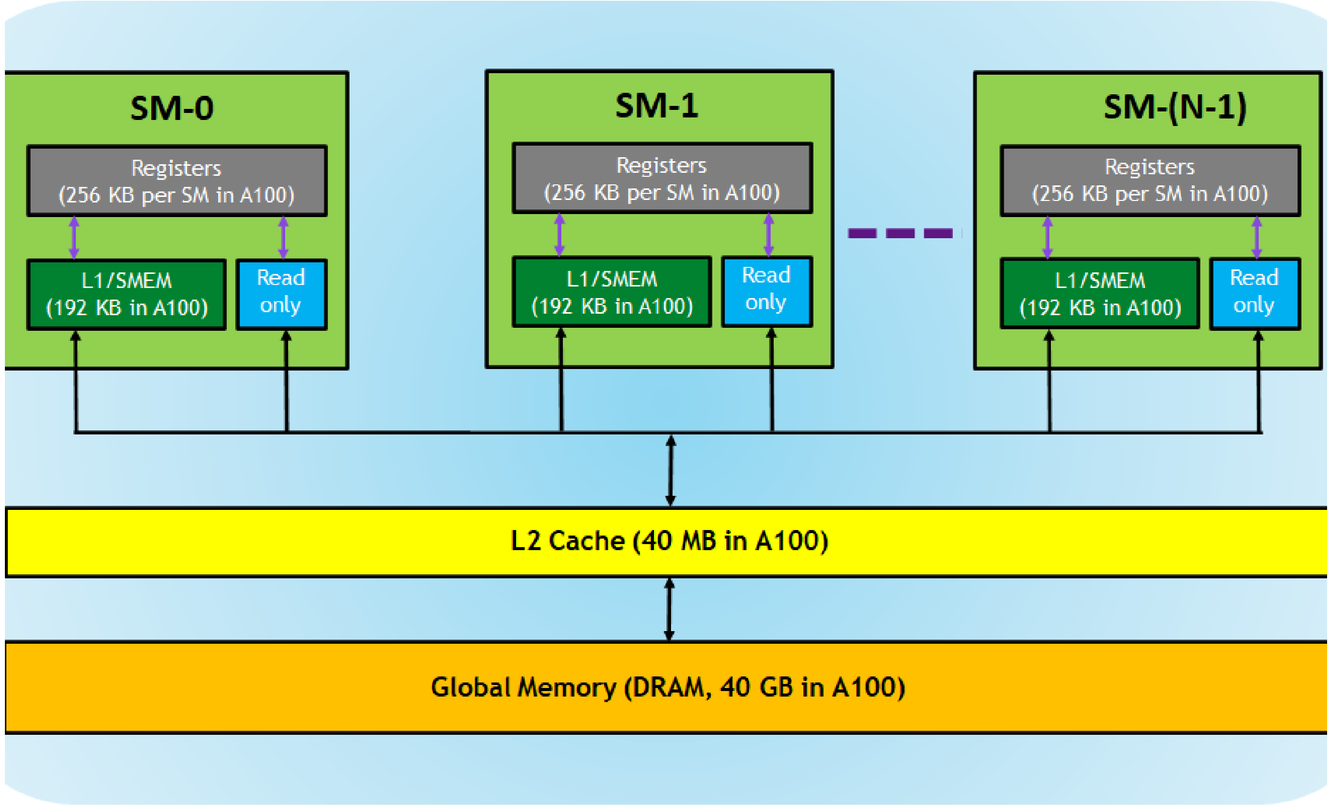
C. It reuses memory movement between Dram and Cache

D. It increases arithmetic intensity because it makes the computation faster

Today: GPU and CUDA

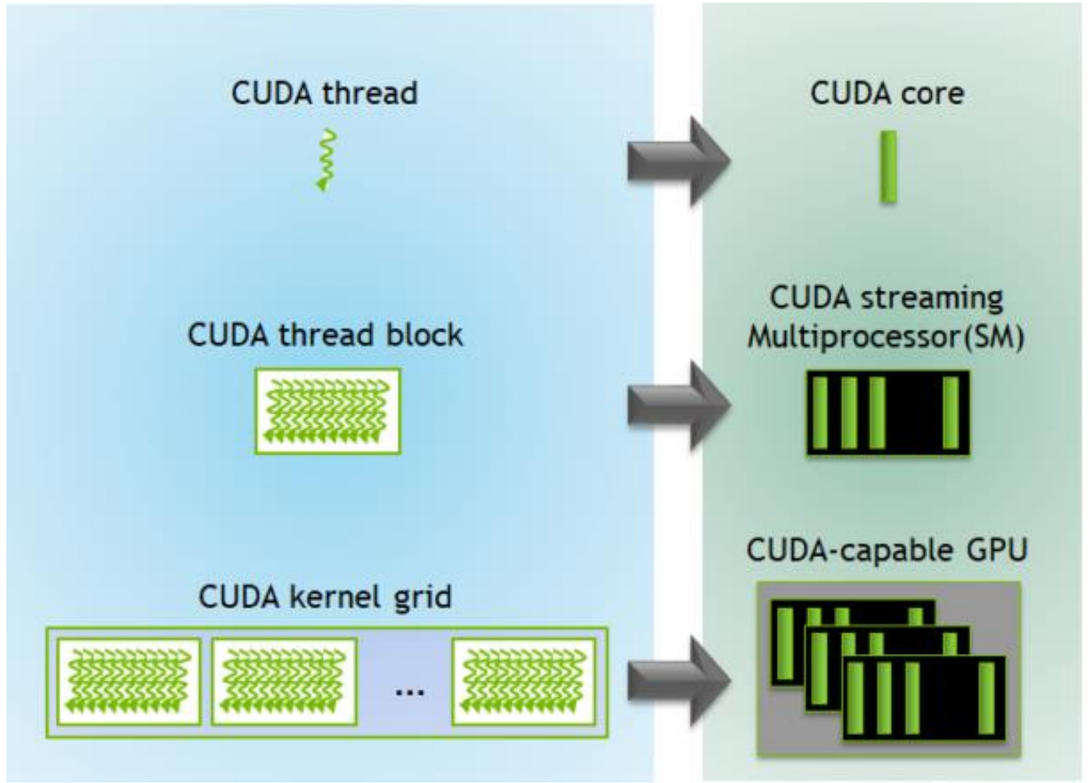
- Basic concepts in GPUs
  - Execution Model
  - Memory
- Programming abstraction
- Case study: Matmul

GPU Overview



Kernel, Threads, Blocks, Grids

- Threads: smallest units to process a chunk of data
- Blocks: A group of threads that share memory
- Grid: A collection of blocks that execute the same kernel
- Kernel: CUDA program executed by many CUDA cores in parallel

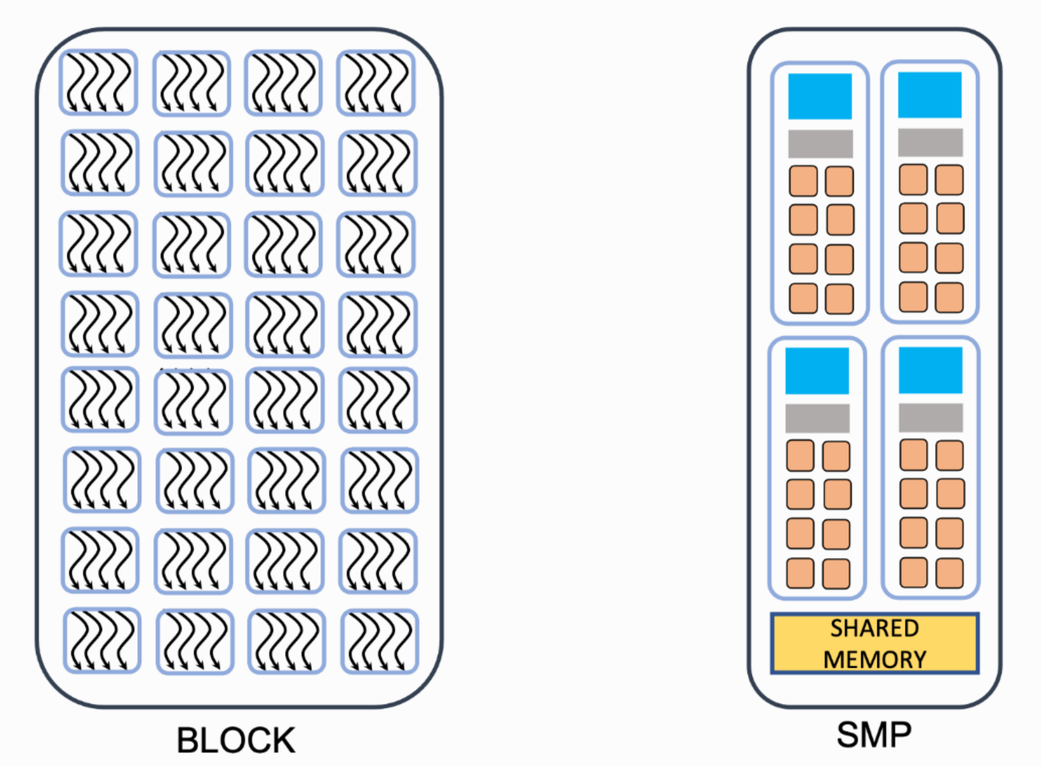


Threads

GPU/CUDA thread vs. OS thread?

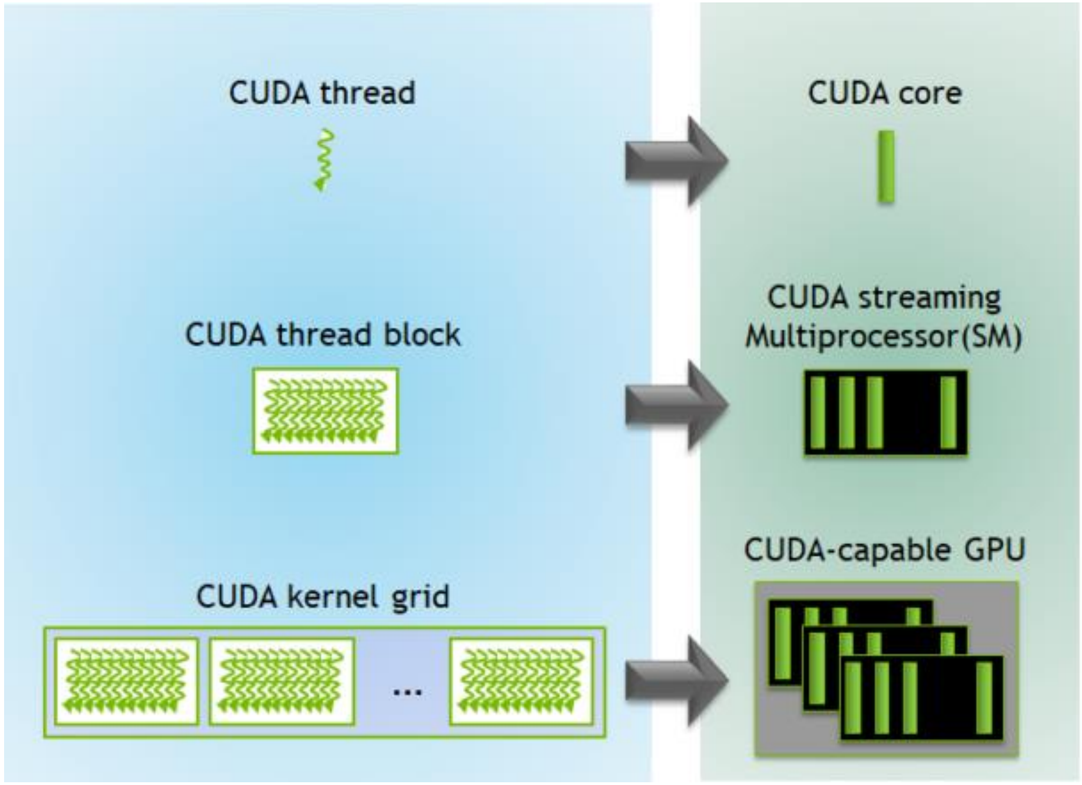
Thread Block

- A collection of many threads mapped to a streaming multiprocessor (SM/SMP)



Grid

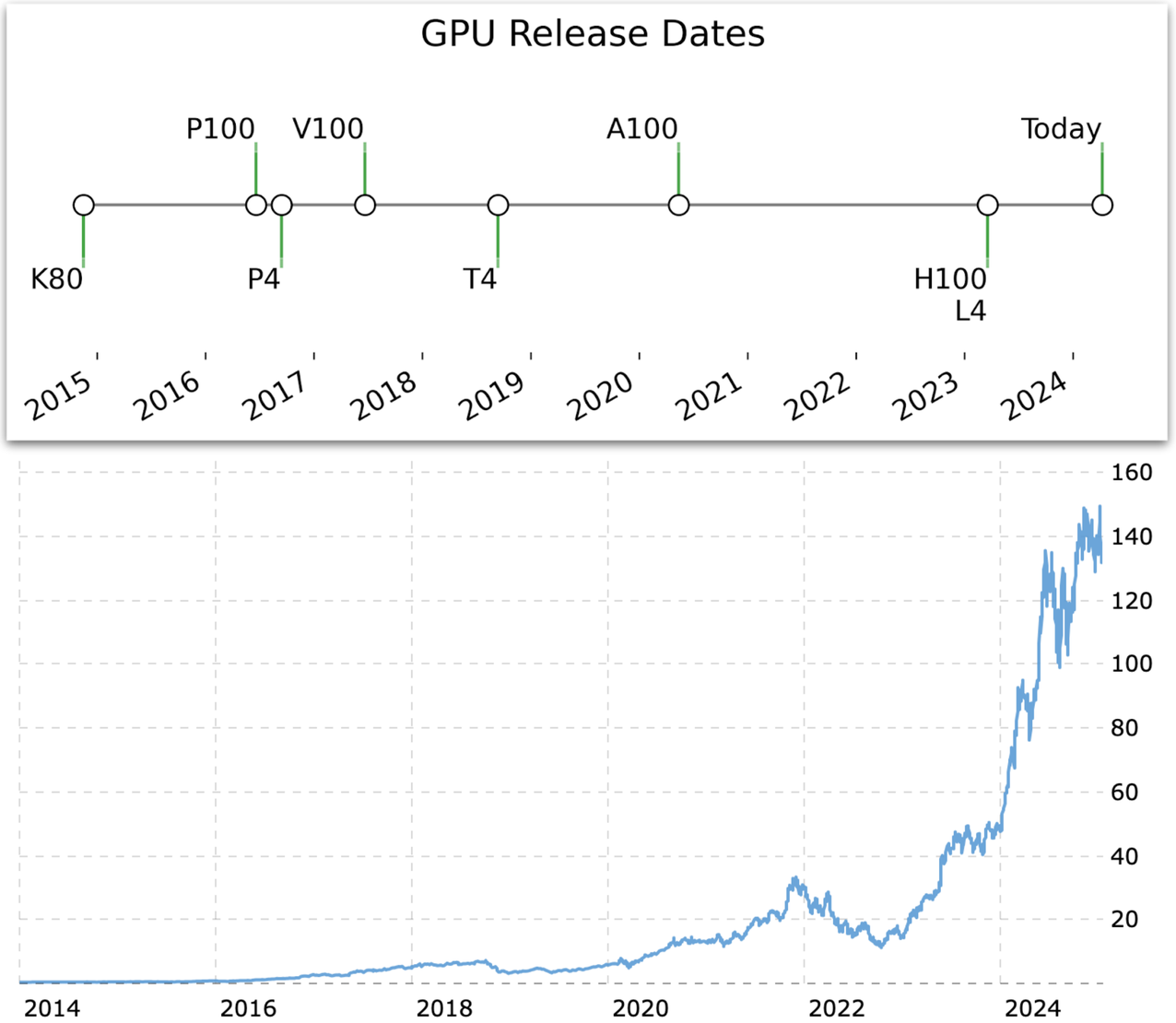
- A collection of blocks (SMs) that execute the same kernel



More power GPU generally means:

- More SMs
- More core/SM
- More powerful cores

Nvidia ML GPU Trajectory



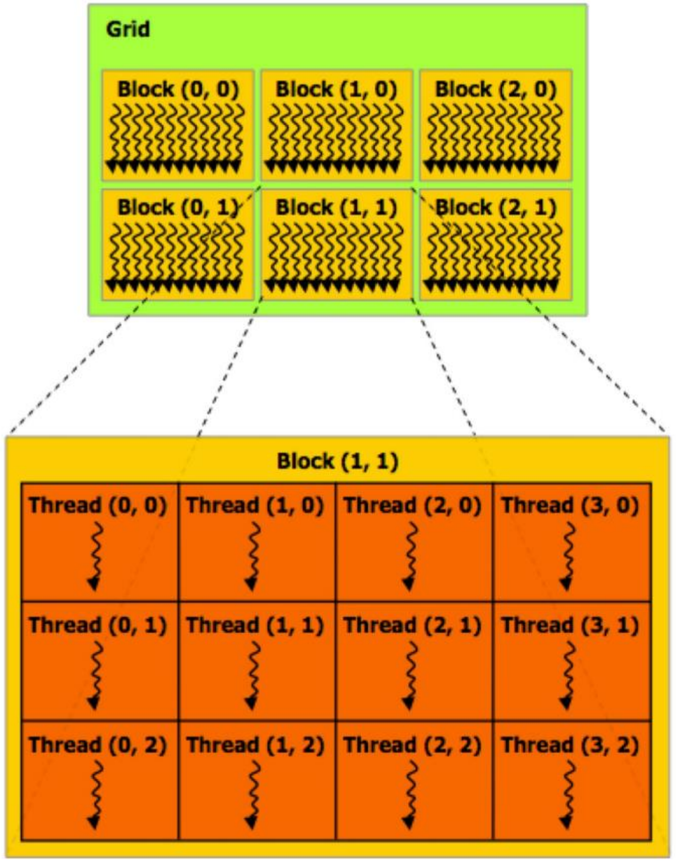
SMs/Threads on Nvidia's GPUs and AWS on-demand Price

- V100 (2018 - Now): 80 SMs, 2048 threads/SM,
  - ~$3/hour/GPU
- A100 (2020 - Now): 108 SMs, 2048 threads/SM,
  - ~$4/hour/GPU
- H100 (2022 - Now): 144 SMs, 2048 threads/SM
  - ~$12/hour/GPU
- B100 and B200 (2025 -): go surveying the number

CUDA

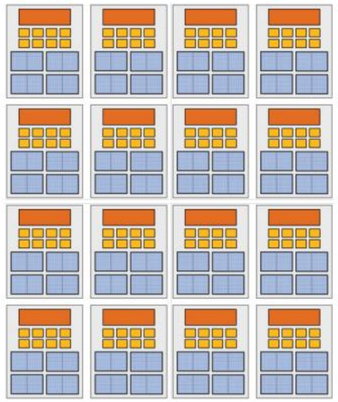
- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming GPUs
- CUDA's design matches the grid/block/thread concepts in GPUs

CUDA Programs contain A Hierarchy of Threads



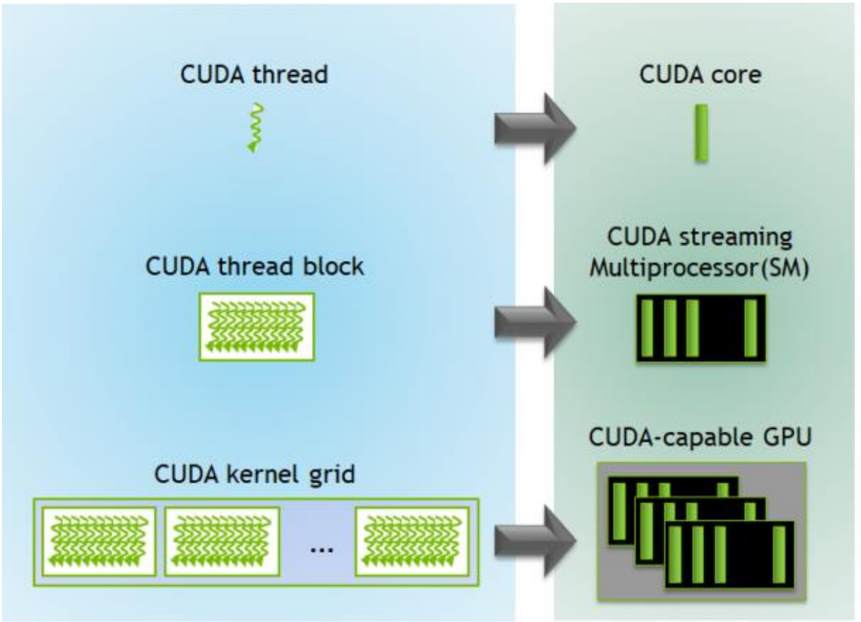

```
const int Nx = 12;
const int Ny = 6;

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);
```

Run on CPU



How Many threads/Blocks it runs on?

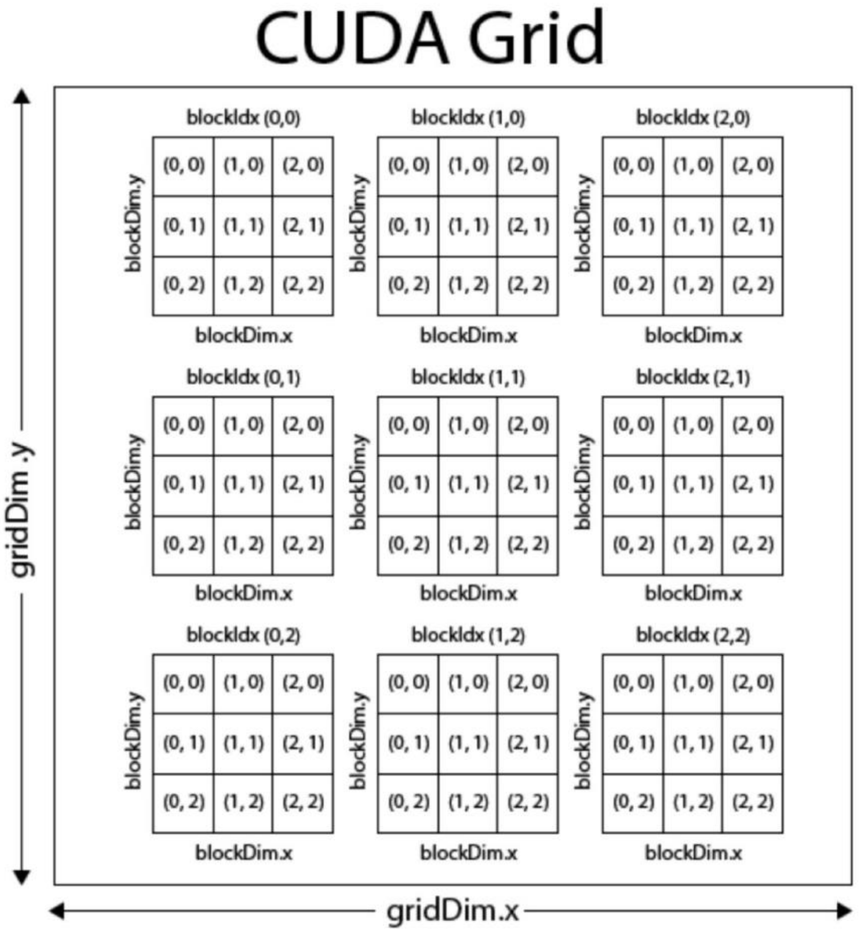
```
const int Nx = 12;
const int Ny = 6;

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);
```



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

- What About GridId?
- What about threadDim?



An Example CUDA Program: Matrix Add

```
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 cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

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

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(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] + doubleValue(B[j][i]);
}
```

- __global__ denotes a CUDA kernel function runs on GPU
- Each thread indexes its data using blockIdx, blockDim, threadIdx and execute the compute

Separation CPU and GPU Execution

```
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 cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

- Host code: serial execution on CPU

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(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] + doubleValue(B[j][i]);
}
```

- Device code: SIMD parallel execution on GPUs

Question

```
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 cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(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] + doubleValue(B[j][i]);
}
```

What happens post launching the kernel?

- Will the CPU program continue
- What if the function has return values?

#Threads is Explicit and Static in Programs

```
const int Nx = 11; // not a multiple of threadsPerBlk.x
const int Ny = 5; // not a multiple of threadsPerBlk.y

dim3 threadsPerBlk(4, 3, 1);
dim3 numBlocks(3, 2, 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, threadsPerBlk>>>(A, B, C);
```

```
// kernel definition
__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;
   // guard against out of bounds array access
   if (i < Nx && j < Ny)
      C[j][i] = A[j][i] + B[j][i];
}
```

Developers to:

- To provide CPU/GPU code separation
- Statically declare blockDim, shapes.
- Map data to blocks/threads

On potential factor many compiler (torch.compile) to require static shapes

Hence it is Important to:

- Check boundary conditions

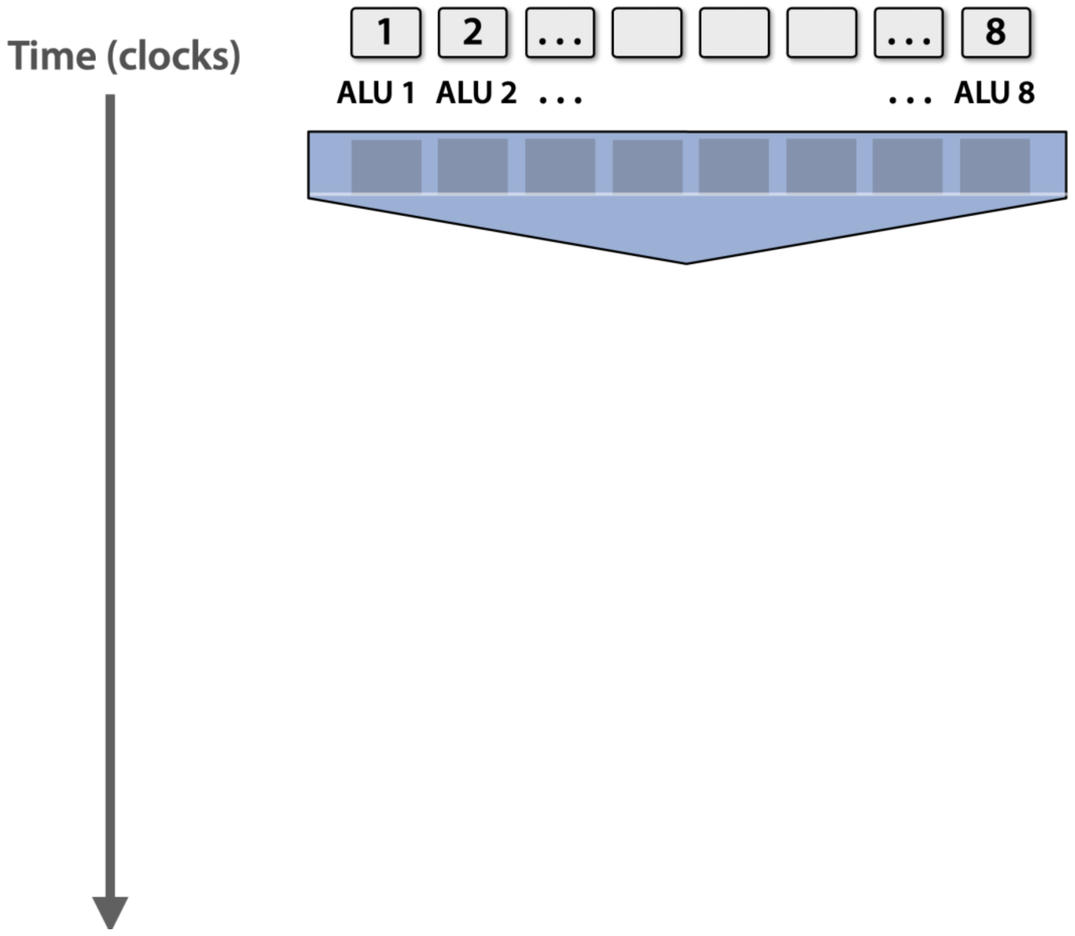
SIMD Constraints: how to handle control flow?

SIMD requires all ALUs/Core Must proceed in the same pace

- Why?
- Let's look at a control flow example

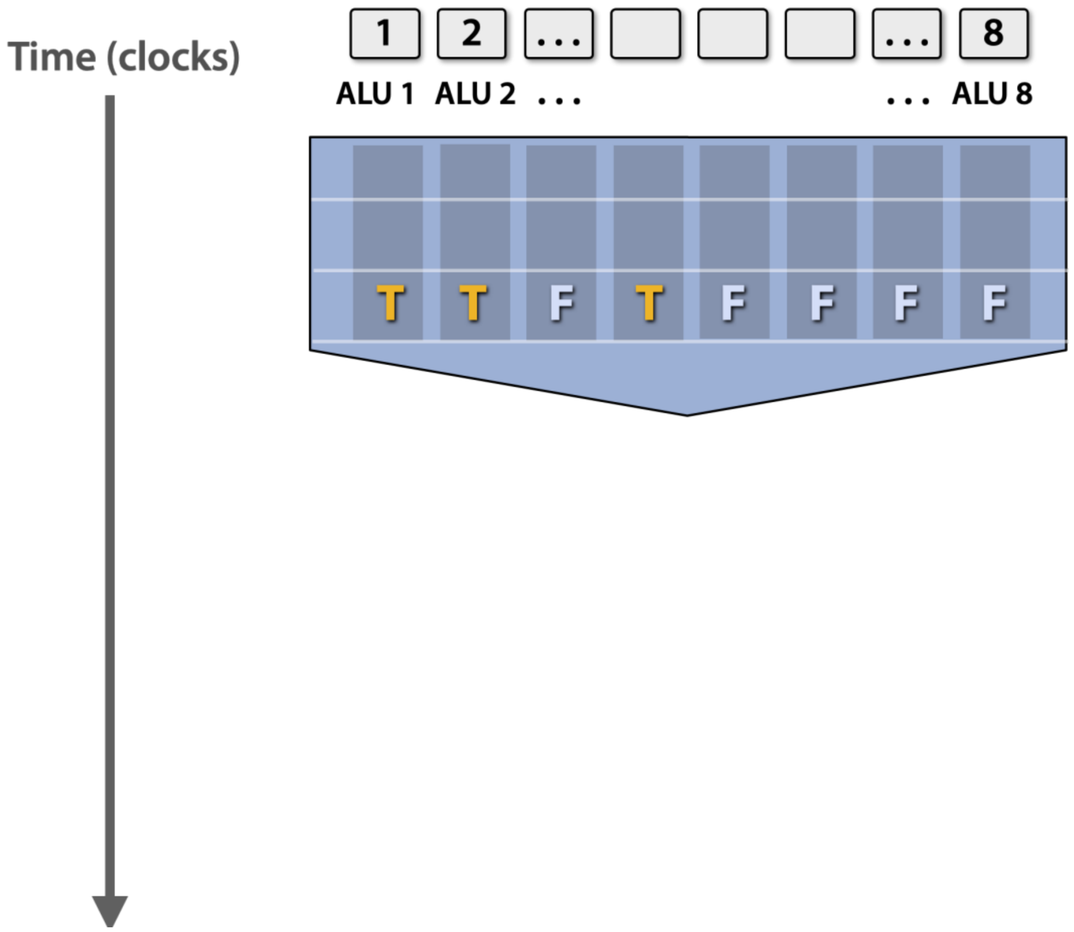
```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



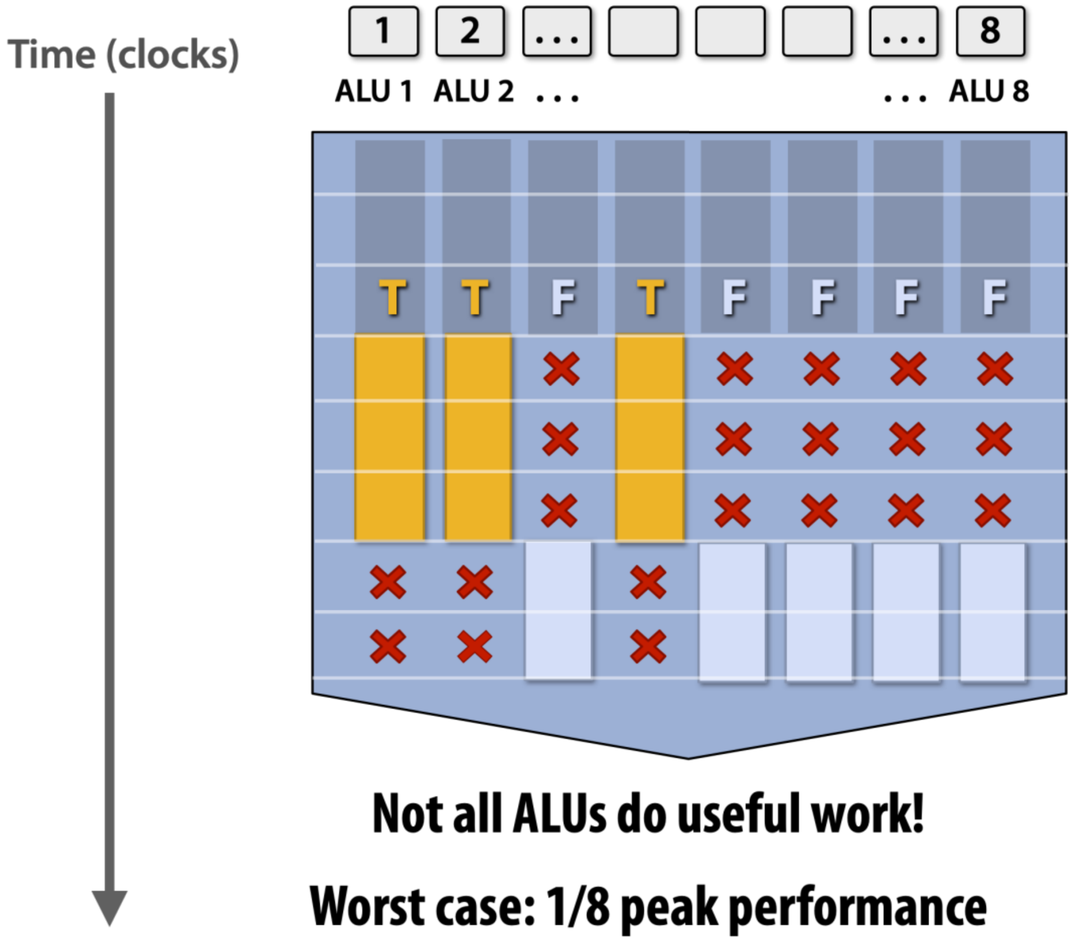

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



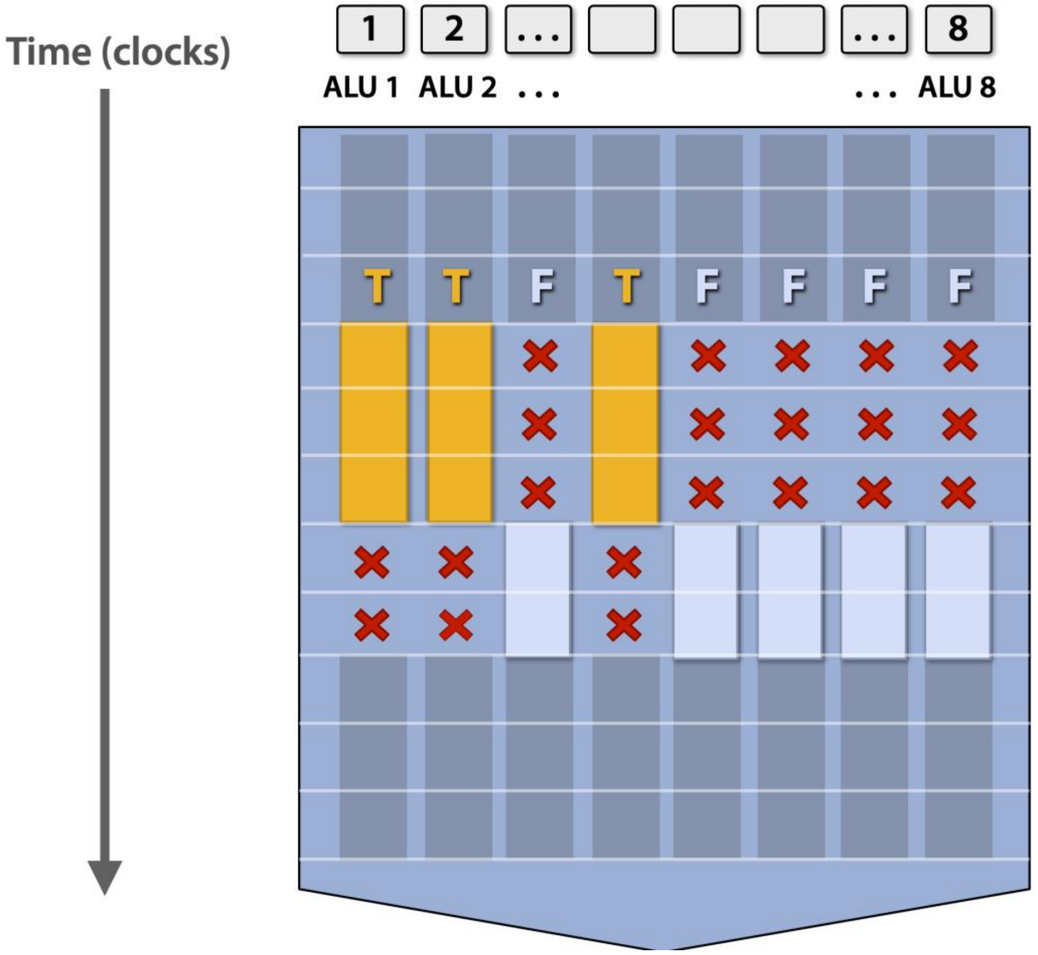

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow: Masking




```
// kernel definition
__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;
}
```

Handling Control Flow




```
// kernel definition
__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;
}
```

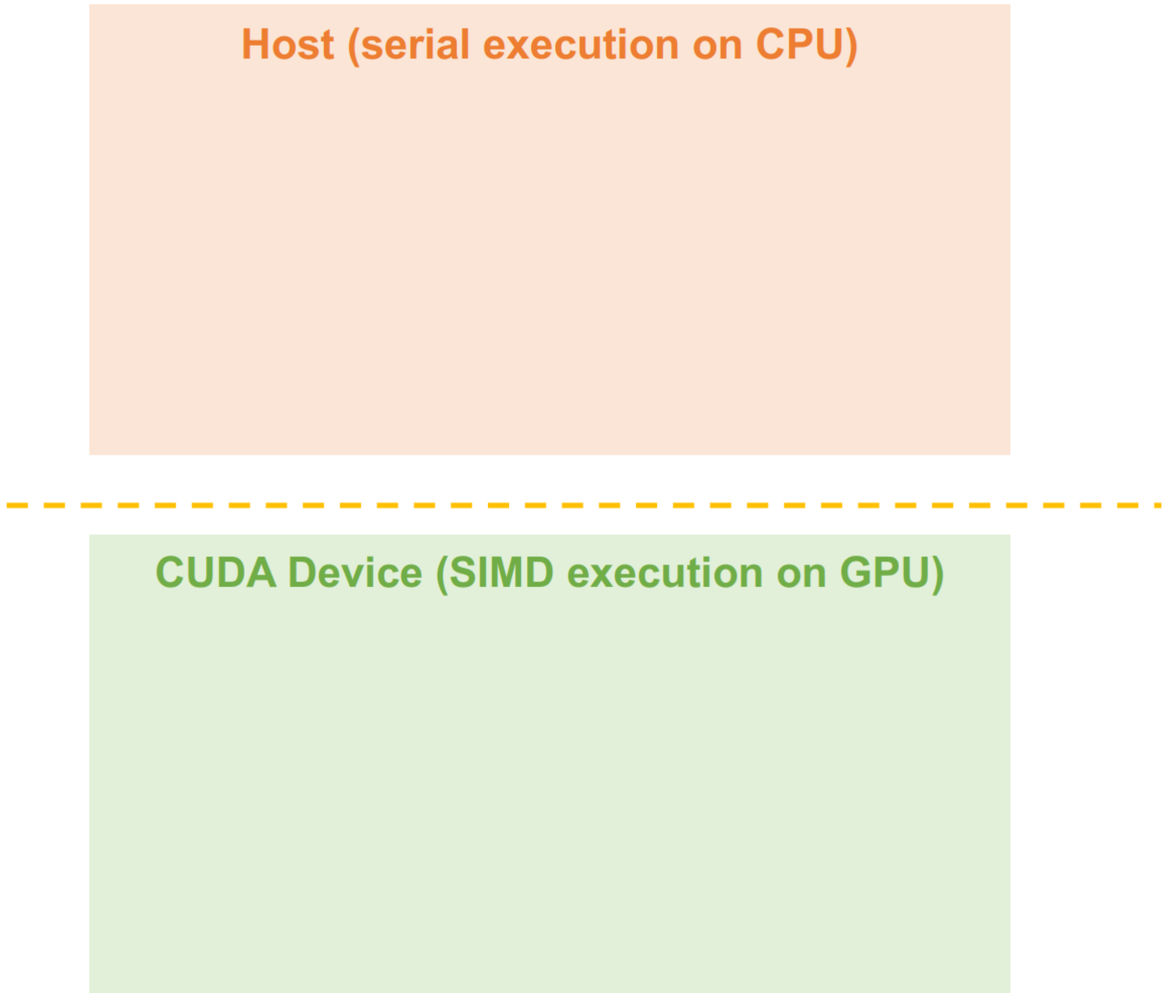
Coherent vs. Divergent

- Coherent execution:
  - Same instructions apply to all data
- Divergence Execution:
  - On the contrary of coherent
  - Should be minimized in CUDA programs
- A notable case
  - Language model masking, sliding window attention

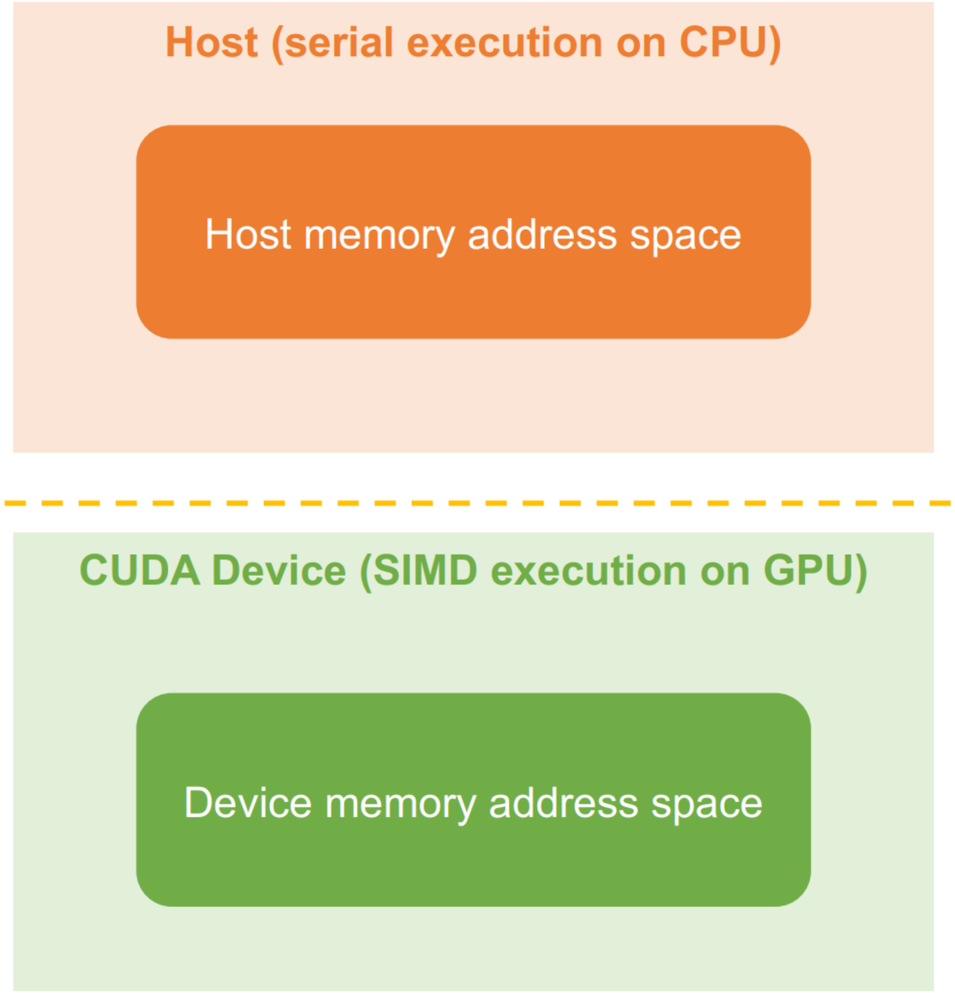
GPU and CUDA

- Basic concepts in GPUs
  - Execution Model
  - **Memory**
- Programming abstraction
- Case study: Matmul

CUDA Memory Model



CUDA Memory Model



Concepts:

- Host memory: RAM
- Device memory: GPU memory

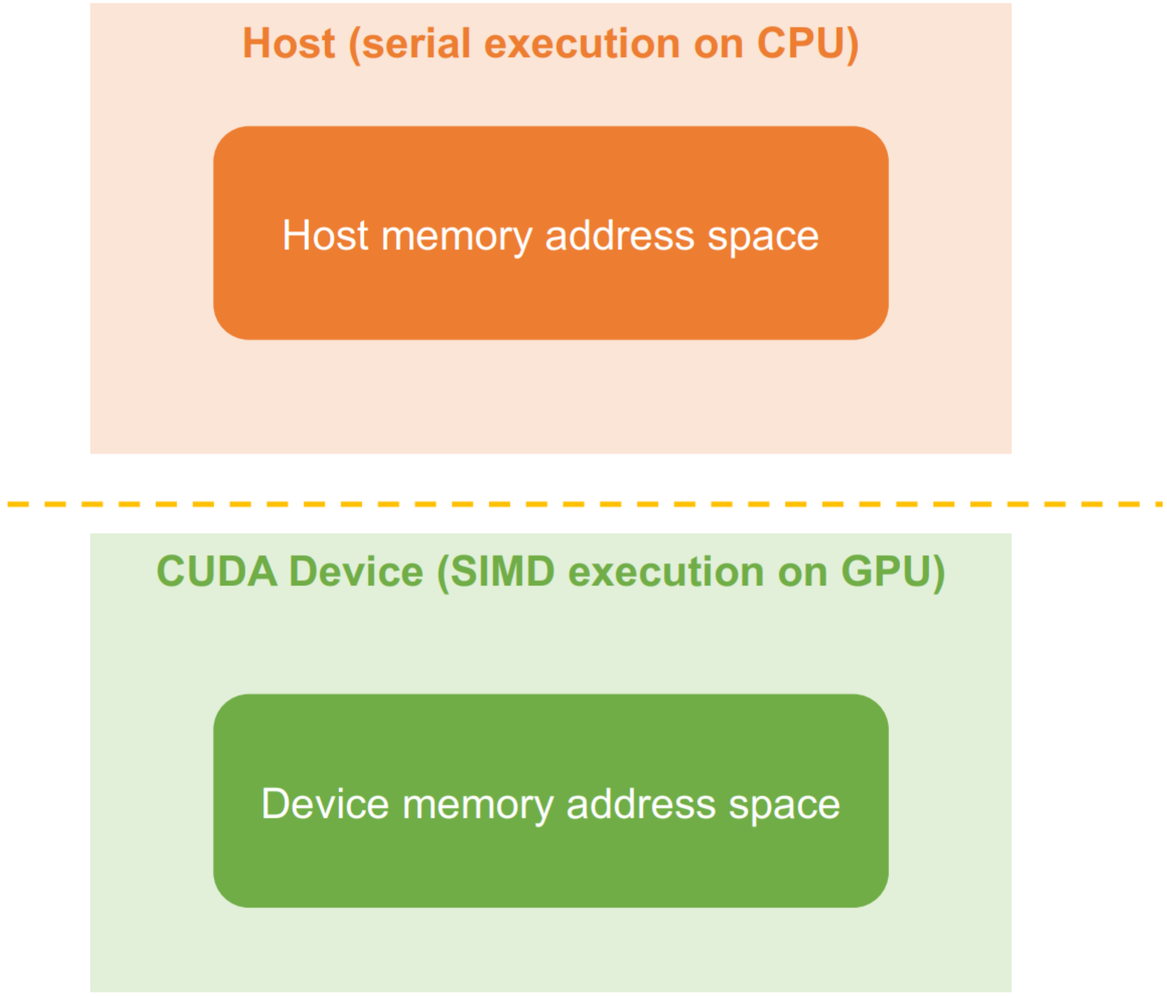
Recap:

- How is host memory managed in OS?

Distinct host and device address spaces:

- CPU code cannot access device memory
- GPU code cannot access host memory

cudaMemcpy



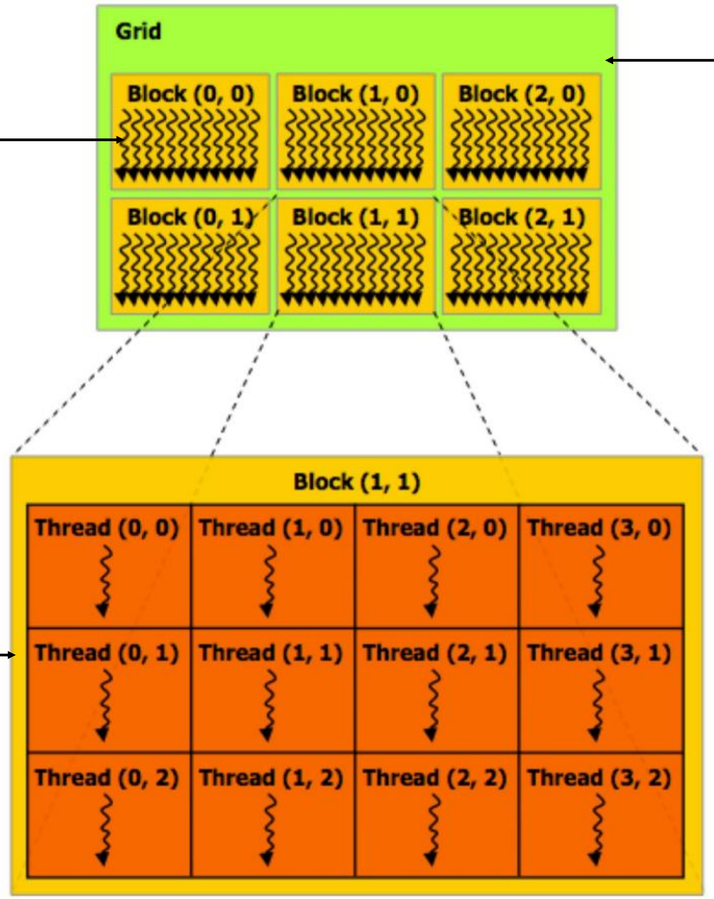

```
float* A = new float[N];

// populate host address space pointer A
for (int i=0 i<N; i++)
   A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;                  // allocate buffer in
cudaMalloc(&deviceA, bytes);     // device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

More concepts: Pinned memory

- A part of host memory
- Optimized for data transfer between CPU/GPU
- Not pagable by OS, a.k.a. locked
- Certain APIs only work on Pinned memory

Memory from a kernel's perspective

Per thread private memory
(r/w by that thread)



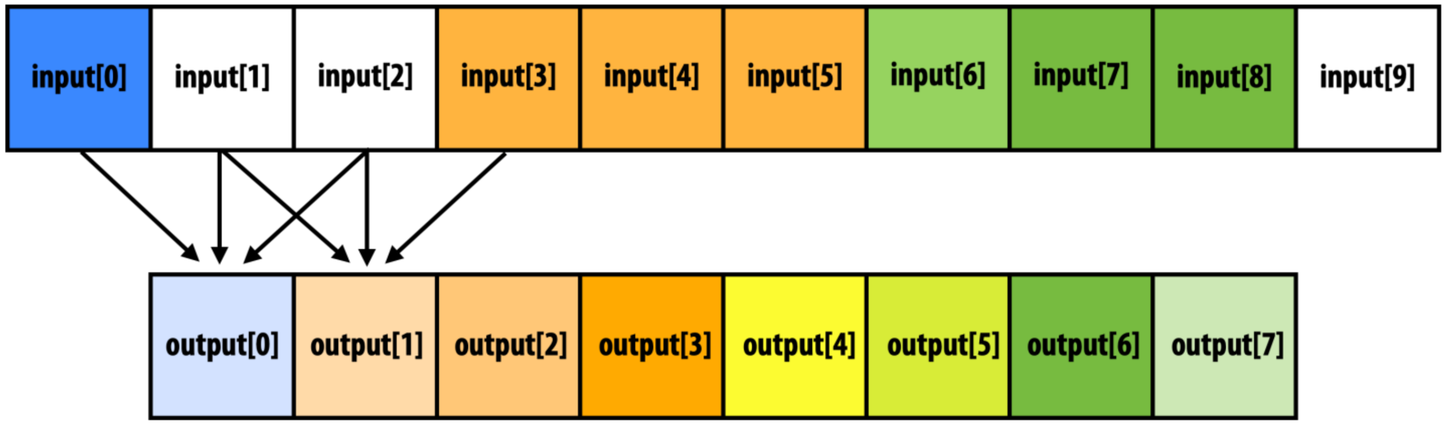
Global device memory
(r/w by all threads)

Per block shared memory:
(r/w by all threads of the block)

Why make it so complex:

- Balance between speed and shared-memory parallelism

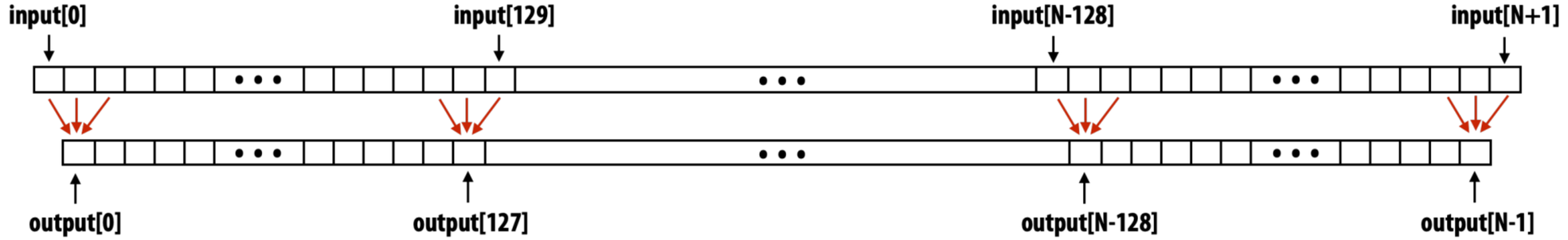
Example Program: Window Average




```
for i in range(len(input) - 2):
    output[i] = (input[i] + input[i+1] + input[i+2]) / 3.0
```

Q: what is the parallelizable part?

Window Average: GPU v1



- Pattern: every 3 adjacent input elements are reduced as an output.
- Parallelization: Every 3-element tuple reduction is independent
- Idea: map each reduction computation to a CUDA core

GPU v1

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += input[index + i];

   output[index] = result / 3.f;
}
```

each thread computes result for one element

- How many threads In total?
- How many blocks?

GPU v1

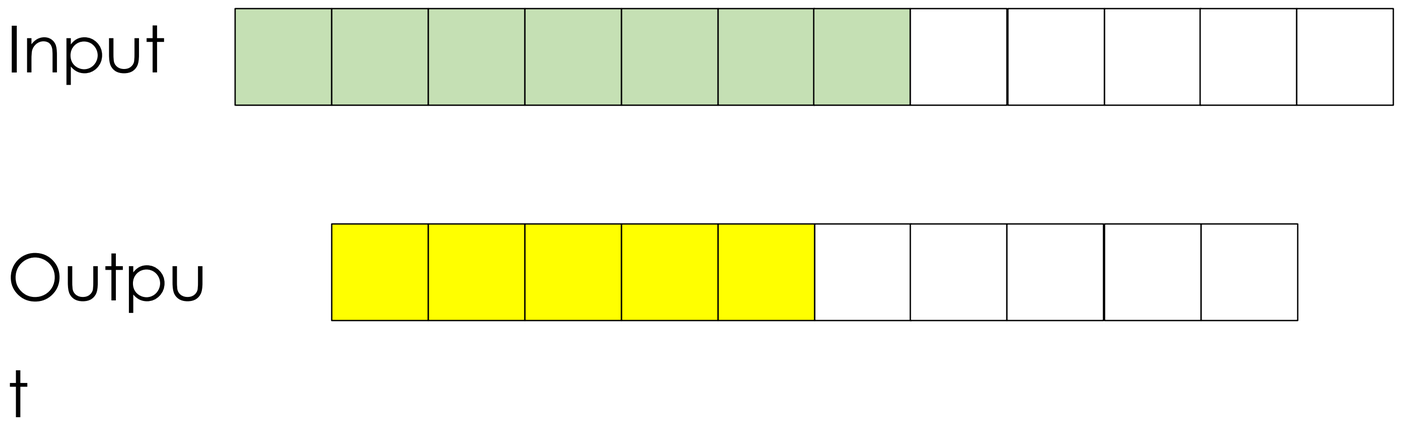
```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += input[index + i];

   output[index] = result / 3.f;
}
```

each thread computes result for one element

Identify a Problem of the above implementation?

High-level Idea to Improve



GPU v2

Q: how many reads we save per block?
Previous: 3 * 128
Now: 130

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   __shared__ float support[THREADS_PER_BLK+2];        // per-block allocation
   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Parallel read by all threads

barrier

Read from the allocated array `support`

Synchronization Primitives

- __syncthreads(): wait for all threads in a block to arrive at this point

- cudasycnhronize(): sync between host and device

```
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 cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

What happens post launching the kernel?

- Will the CPU program continue
- What if the function has return values?

CUDA kernel code needs to be compiled (like C/CPP)

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );
cudaMalloc(&devOutput, sizeof(float) * N);

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Launch 8K thread blocks

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;

   __shared__ float support[THREADS_PER_BLK+2];
   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

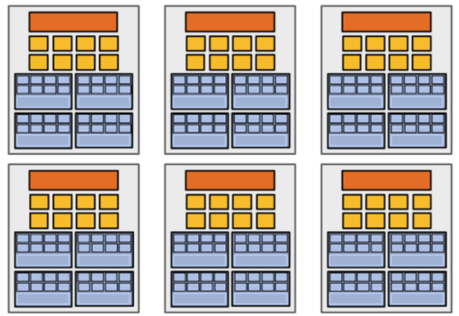
A compiled CUDA device binary includes:

Program text (instructions)
Information about required resources:

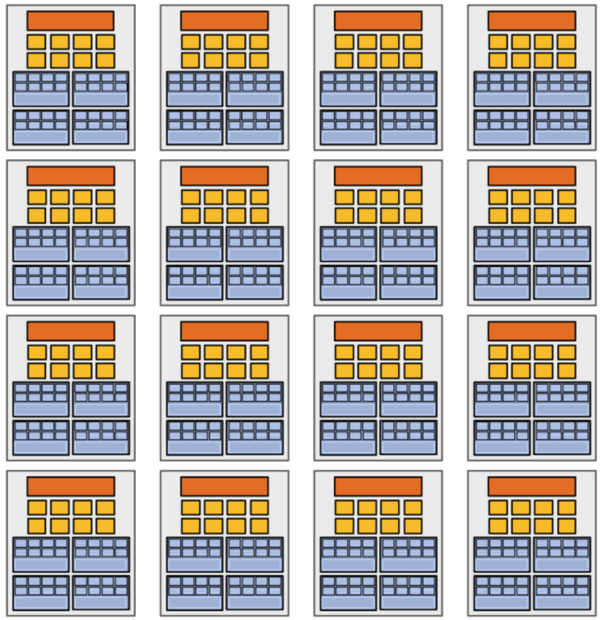
- 128 threads per block
- 8 bytes of local data per thread
- 130 floats (520 bytes) of shared space per thread block

Problem: different GPUs have different SMs

- The user asks for a static (large) number of blocks
- GPUs has varying (limited) number of blocks



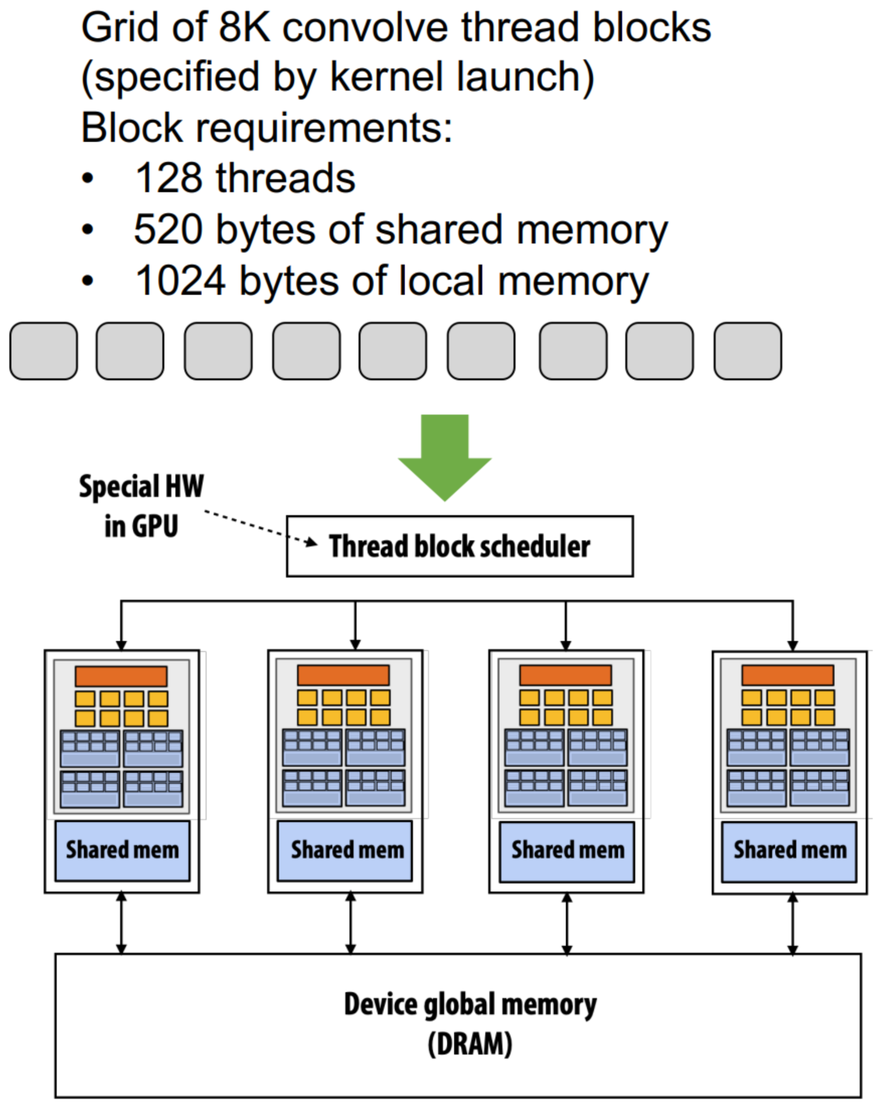
Mid-range GPU (6 cores)



High-end GPU (16 cores)

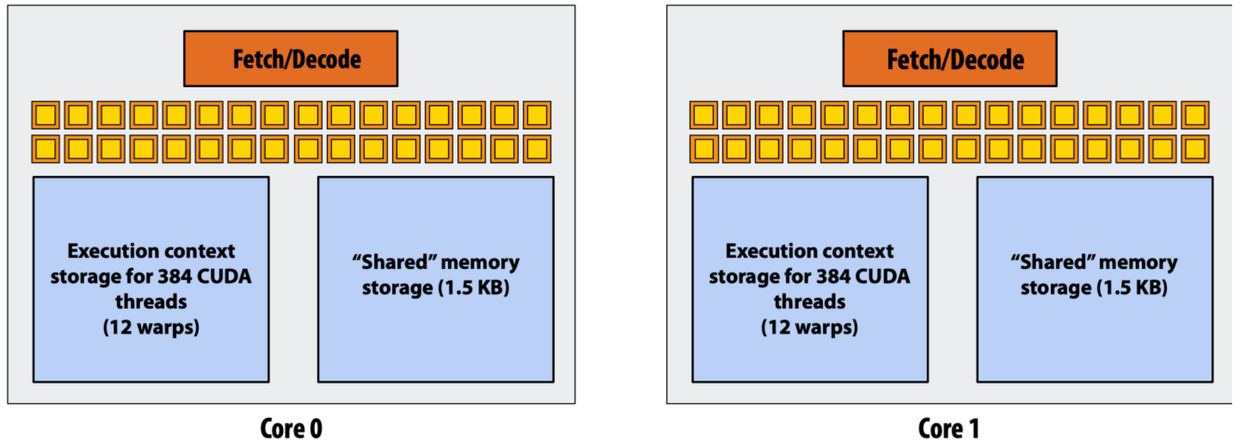
Scheduling on CUDA

- Core assumption: threadblocks can be executed in any order (no dependencies between threadblocks)
- GPUs maps threadblocks to cores using a dynamic scheduling policy threat respects resource requirements



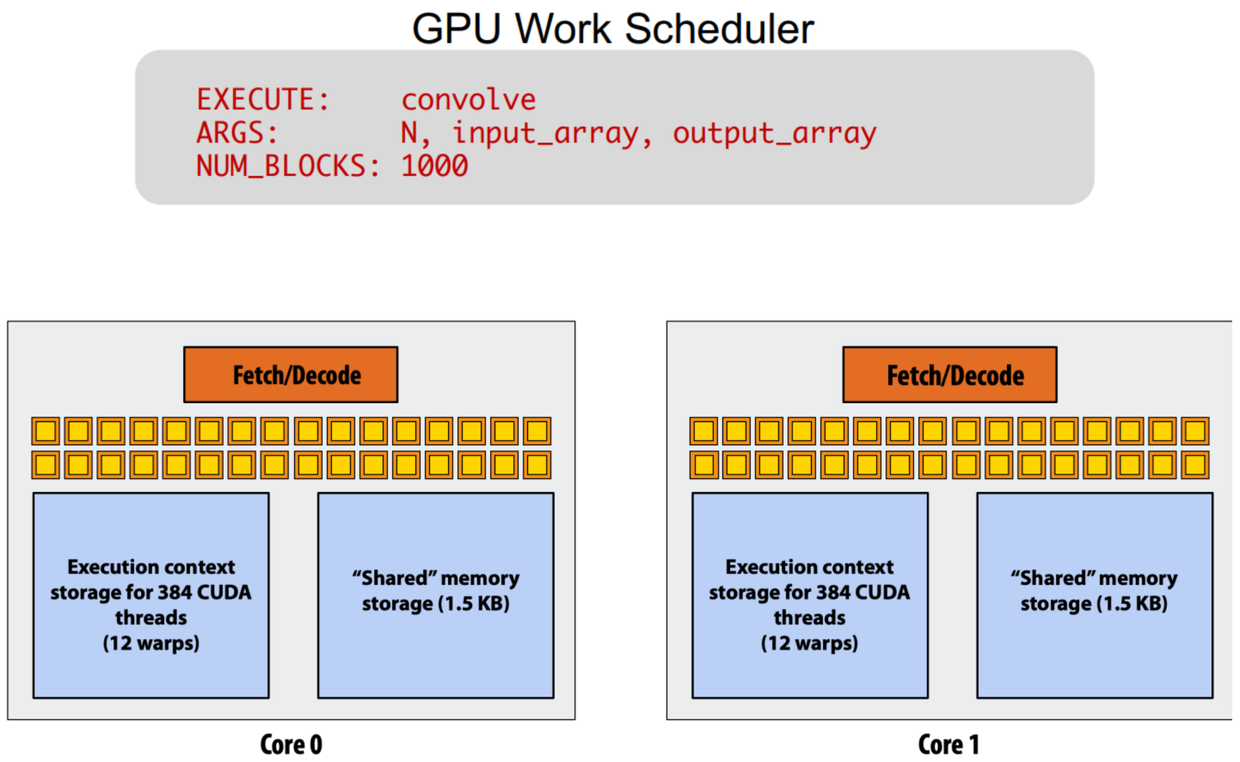
Deep Dive into CUDA scheduling

- Conv1d spec on 1024 x 1024
  - 128 CUDA threads / threadblock
  - 1024 blocks
  - Each threadblock asks for 130 * 4 = 520 bytes of shared memory
- Given: a GPU with two SMs, specs below
- How is the scheduling looking like?



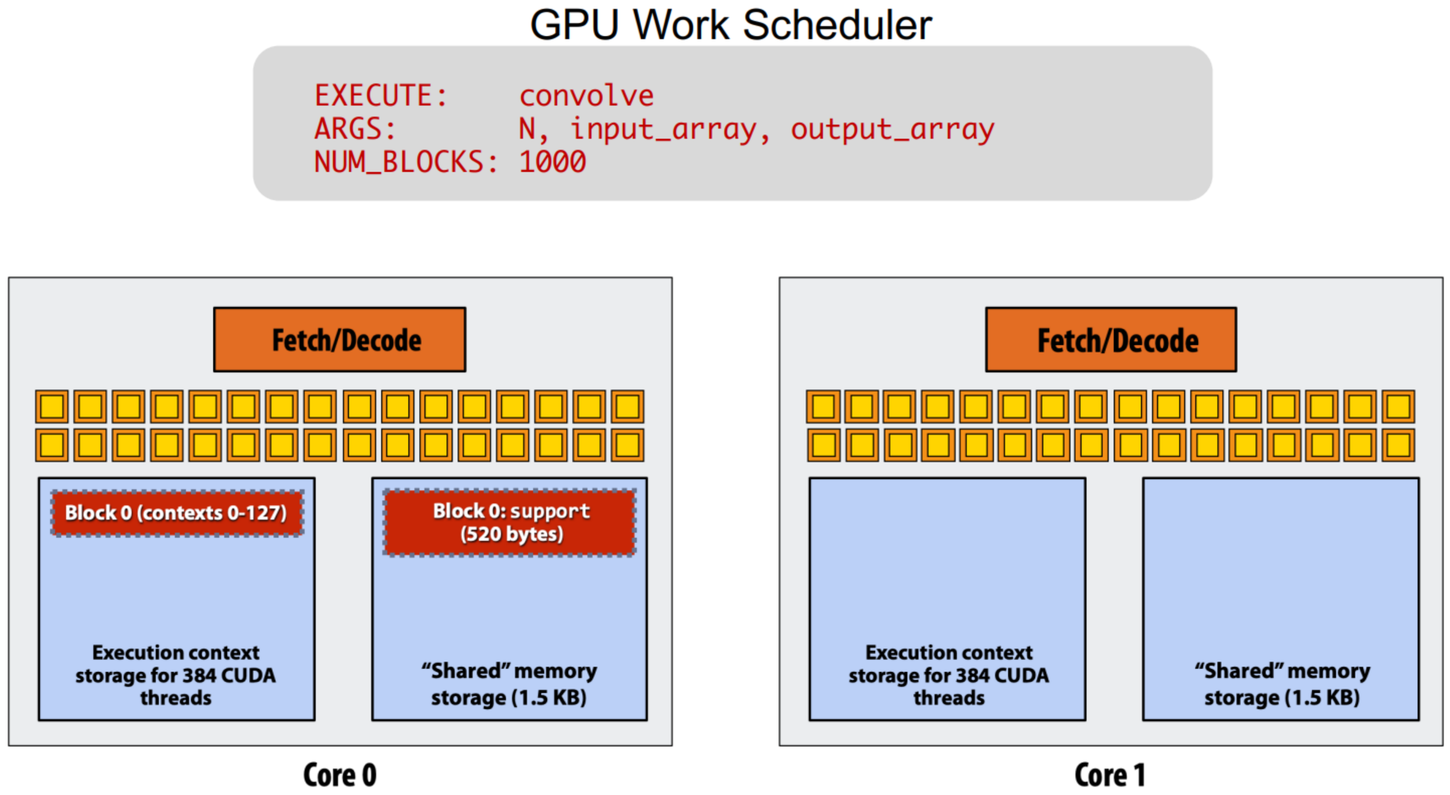
Deep Dive into CUDA scheduling

- Step 1: host sends CUDA kernel instructions to GPU device



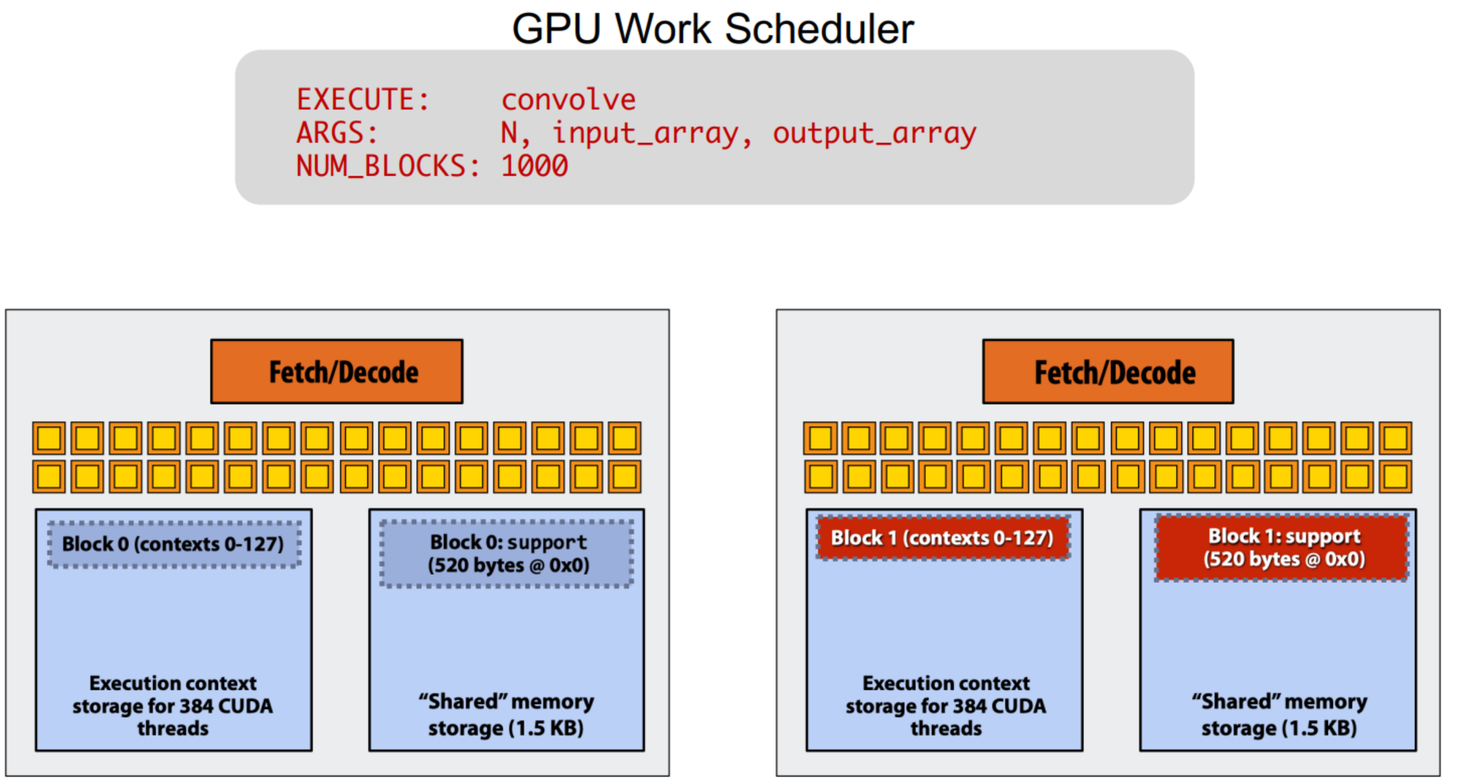
Deep Dive into CUDA scheduling

- Step 2: scheduler maps block 0 to SM 0 (reserves execution contexts for 128 threads and 520 bytes of shared memory)



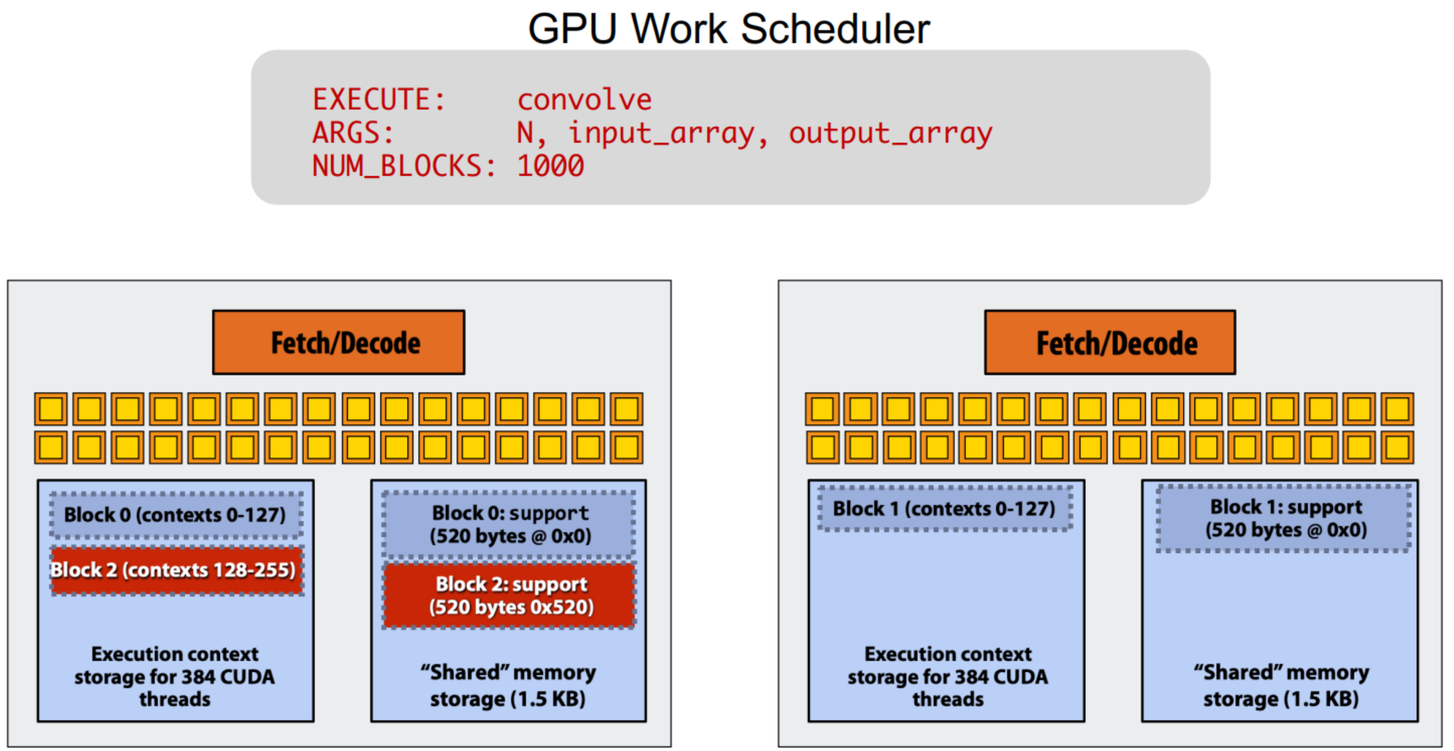
Deep Dive into CUDA scheduling

- Step 3: scheduler continues to map blocks to execution contexts



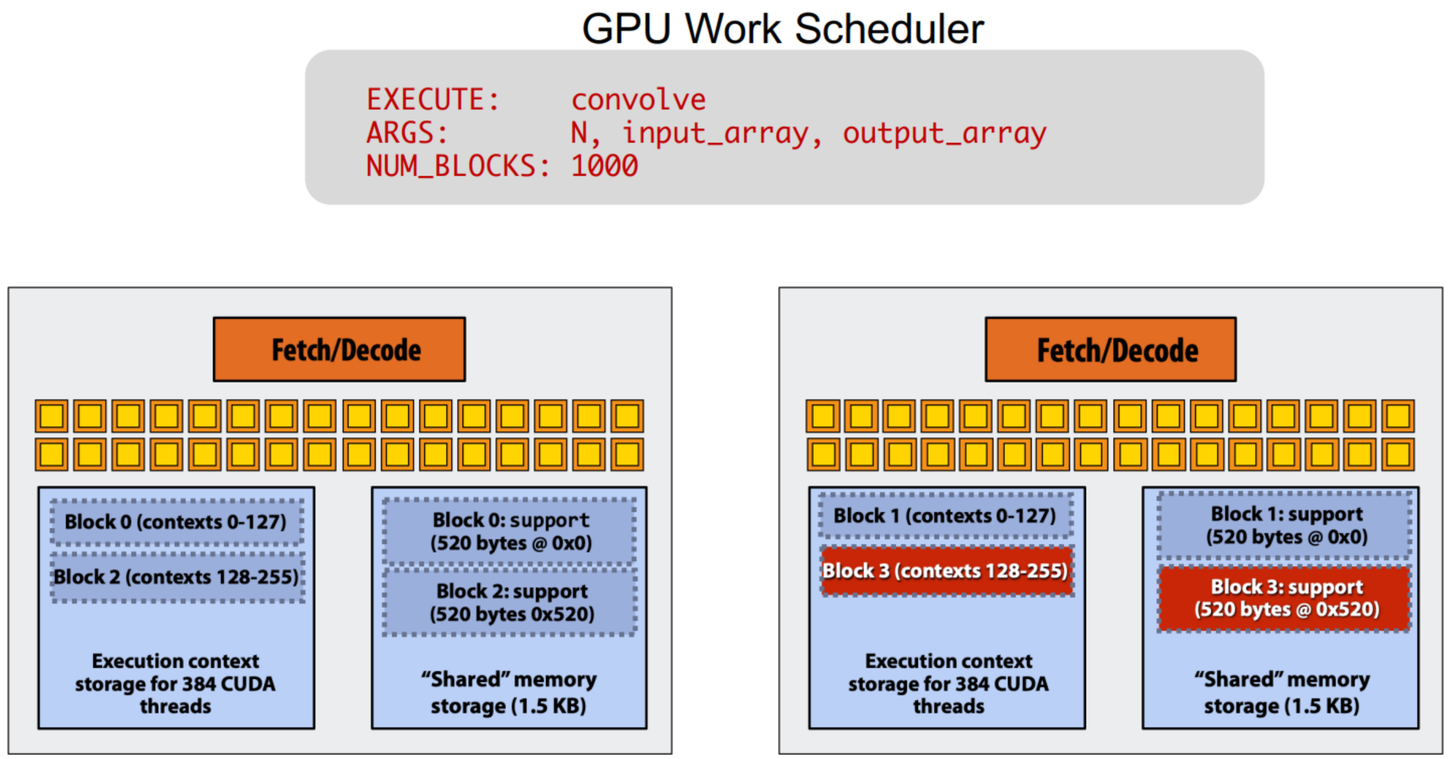
Deep Dive into CUDA scheduling

- Step 3: scheduler continues to map blocks to execution contexts



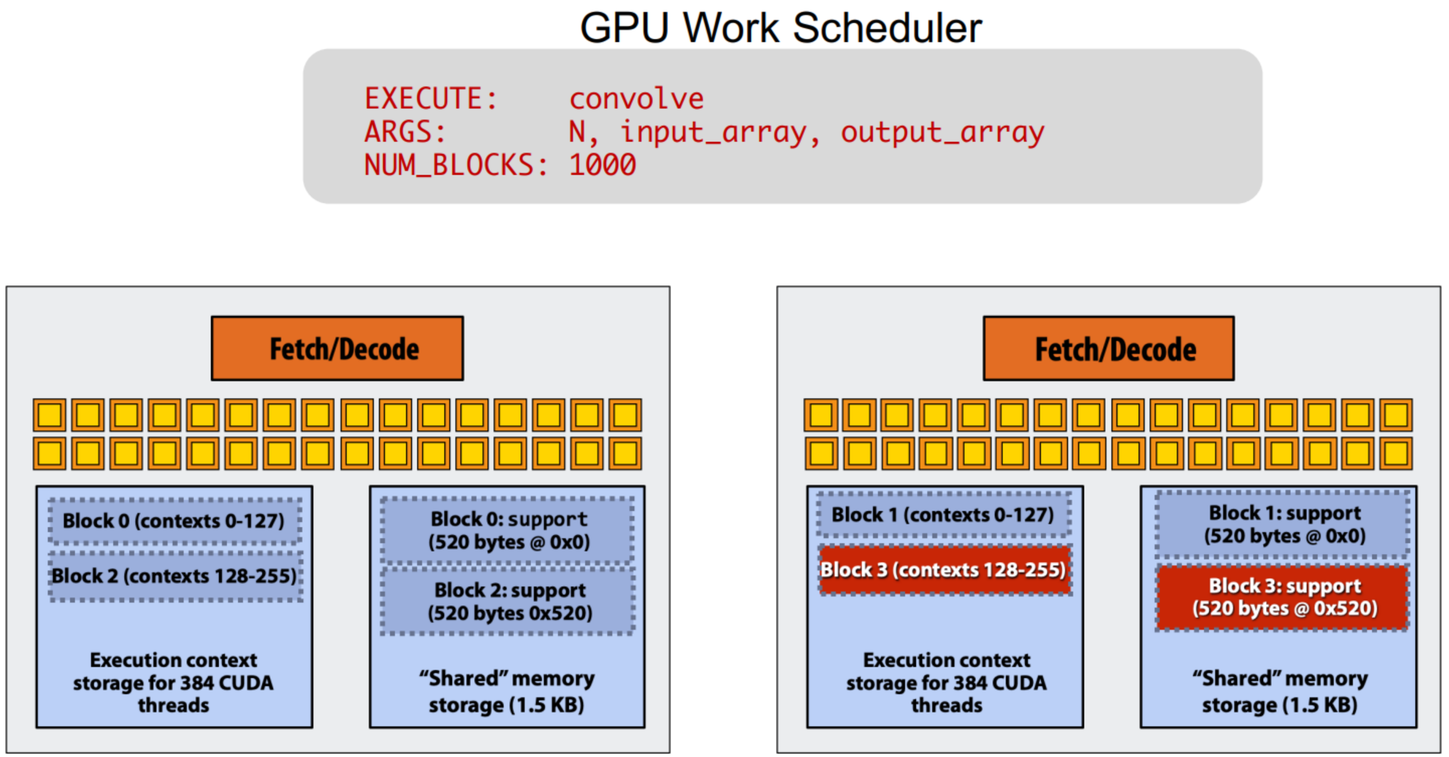
Deep Dive into CUDA scheduling

- Step 3: scheduler continues to map blocks to execution contexts



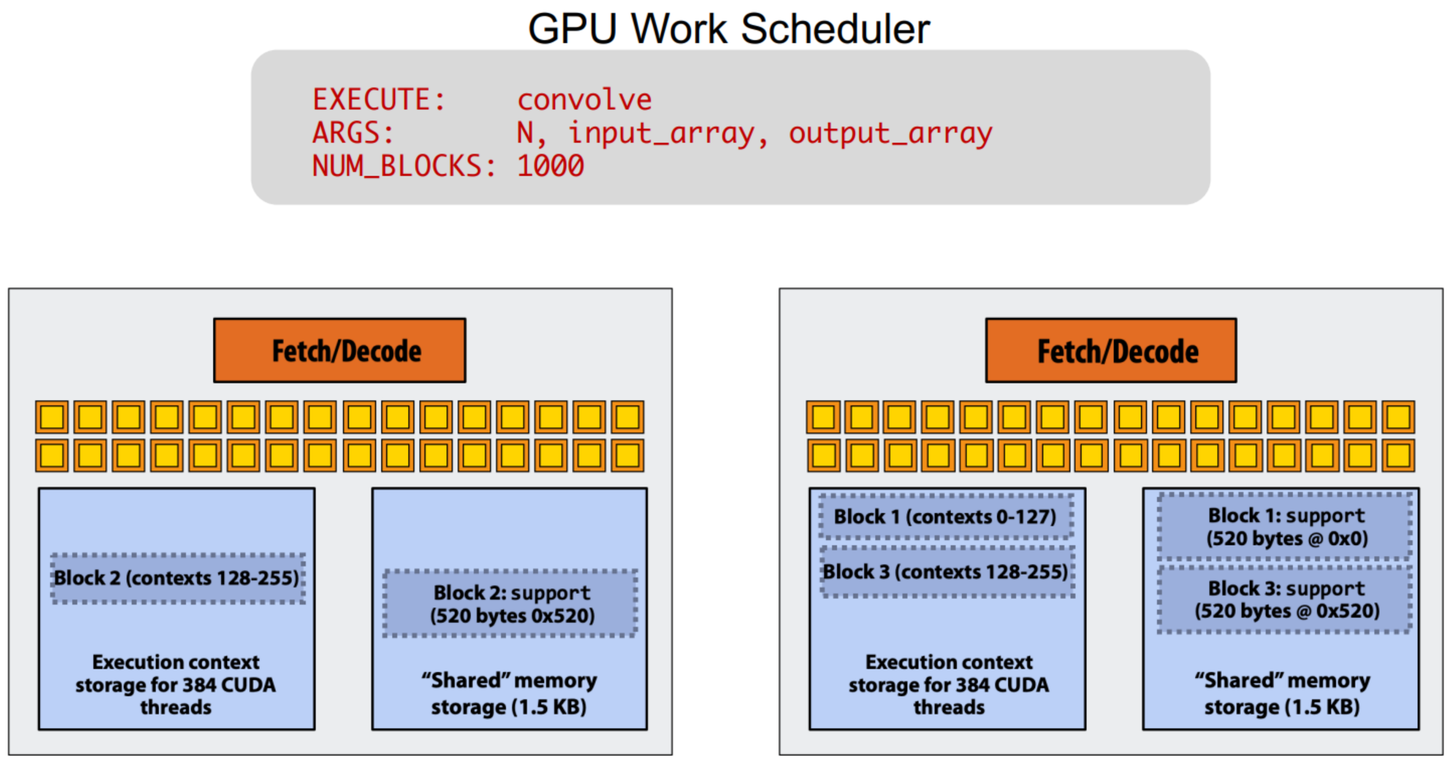
Deep Dive into CUDA scheduling

- Step 3: scheduler continues to map blocks to execution contexts
- But: cannot schedule the 4th block on SM 0 or SM 1. Why?



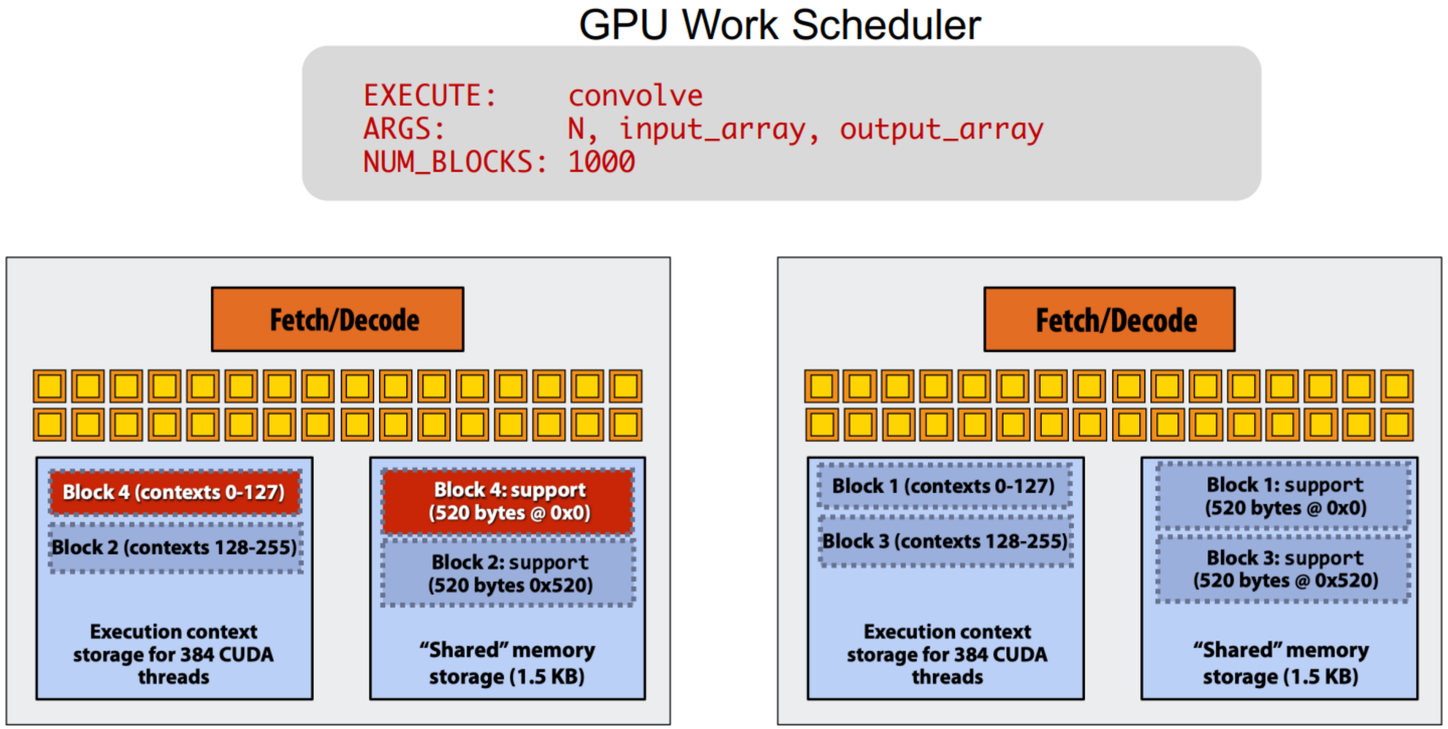
Deep Dive into CUDA scheduling

- Step 4: thread block 0 completes on SM 0



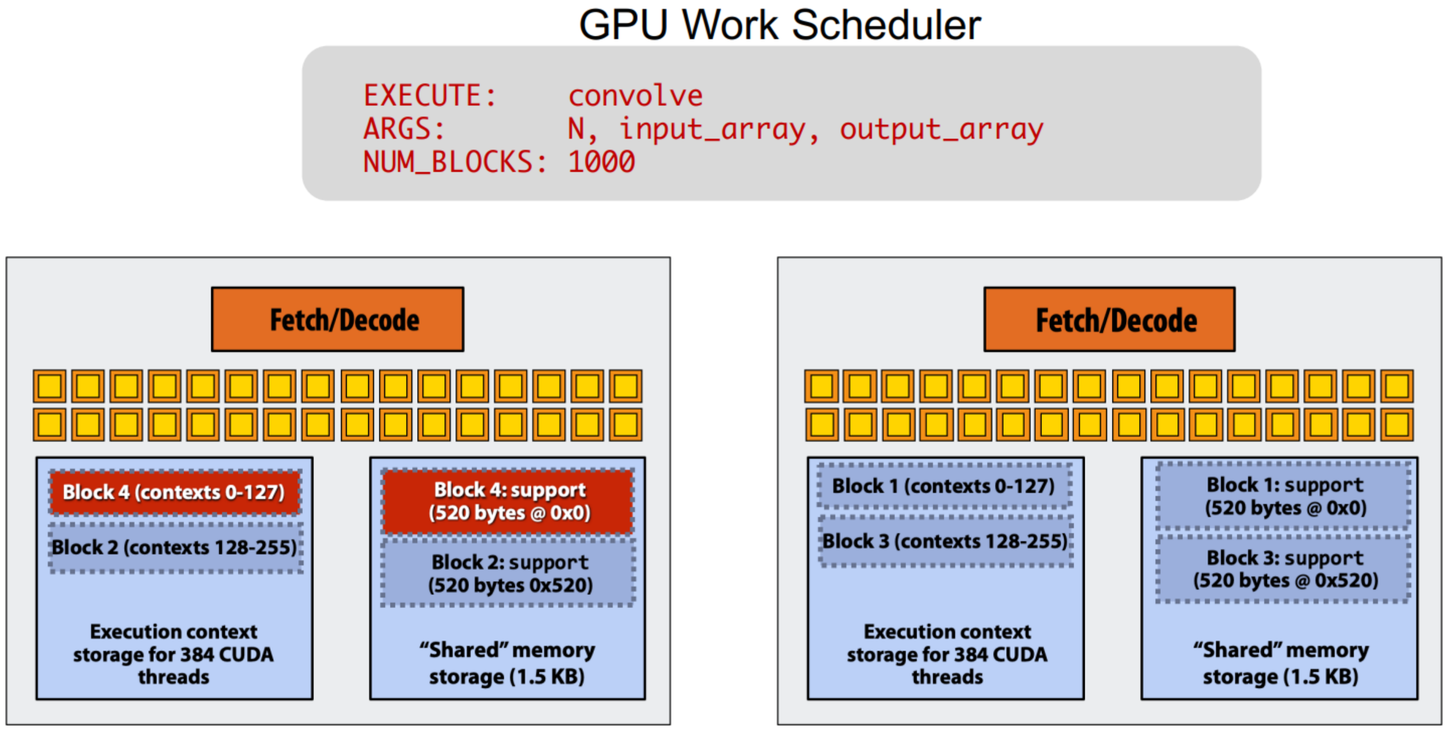
Deep Dive into CUDA scheduling

- Step 5: thread block 4 is scheduled on SM 0 (mapped to execution contexts 0-127)



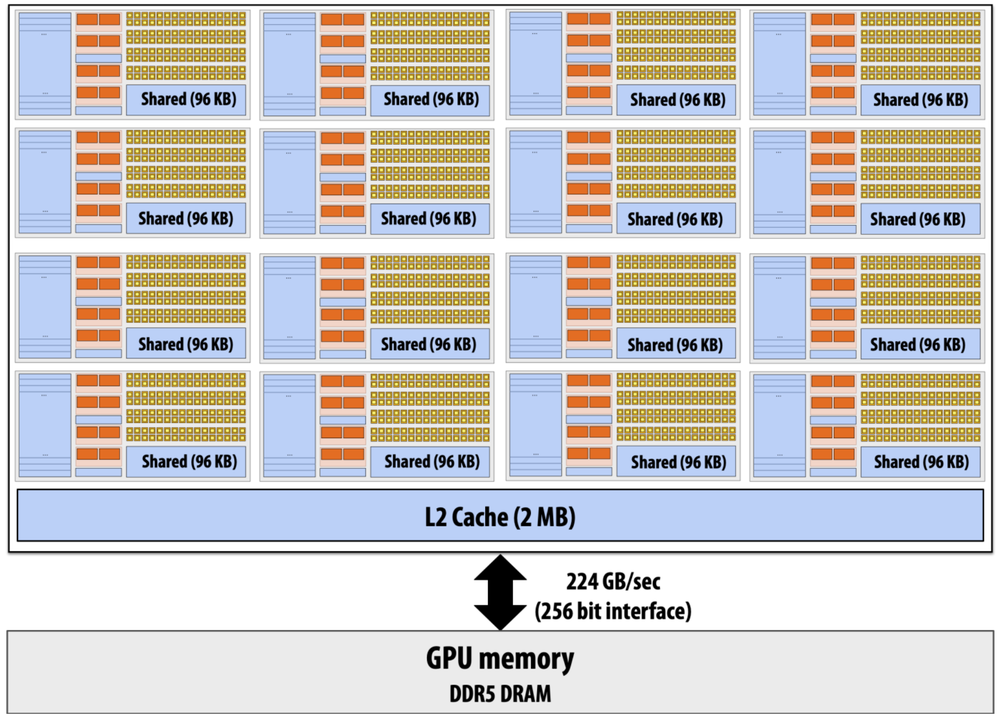
Deep Dive into CUDA scheduling

- Step 5: thread block 4 is scheduled on SM 0 (mapped to execution contexts 0-127)



Recall: An SM on a NVIDIA GTX 980 (2014)

- SM resource:
  - 96KB of shared memory
  - 16 SMs
  - 2048 threads / SM
  - 128 CUDA cores / SM
  - # CUDA cores != # threads, why?



GTX 980 (2014) -> H100 (2022)

- SMs remain the same
  - Threads per block: 2048 -> 2048
  - CUDA cores: 128 -> 128
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)
- #SMs: 16 SMMs -> 132 SMMs
- Flops: 4.6 TFLOPs -> 1000 TFLOPs (mainly because of tensor core)
  - Q: what is tensorcore – how does it work?

If you still remember Groq

GroqCard™

Card Specifications

Form Factor
Dual width, full height, ¾ length PCI Express Gen4 x16 adapter

Performance
Up to 750 TOPs, 188 TFLOPs (INT8, FP16 @900 MHz)

Memory
230 MB SRAM per chip
Up to 80 TB/s on-die memory bandwidth

Chip Scaling
Up to 9 RealScale™ chip-to-chip connectors

Numerics
INT8, INT16, INT32 & TruePoint™ technology
MXM: FP32
VXM: FP16, FP32

Power
Max: 375W; TDP: 275 ; Typical: 240W

Data Center GPU	NVIDIA Tesla V100	NVIDIA A100	NVIDIA H100
GPU Architecture	NVIDIA Volta	NVIDIA Ampere	NVIDIA Hopper
Compute Capability	7.0	8.0	9.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Thread Blocks / Thread Block Clusters	NA	NA	16
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block (CTA)	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (# of threads)	1024	1024	1024
FP32 Cores / SM	64	64	128
Ratio of SM Registers to FP32 Cores	1024	1024	512
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

After Class Survey

- How about B100?
- How does Tensorcore works?
- Why #cores != #active threads in an SM?

Today's summary

- Basic concepts in GPUs
- Execution Model
  - Launch kernel code to grids with many threadblocks
- Memory hierarchy
  - Shared memory – SRAM
- Two example code: matrix-add and conv1d
- Next: matmul grinding