https://hao-ai-lab.github.io/dsc291-s24/

DSC 291: ML Systems
Spring 2024

LLMs

Parallelization

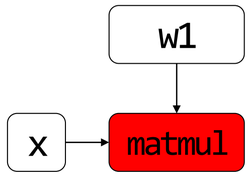
Single-device Optimization

Basics

Logistics

- Next Quiz: Thursday (4/18)
- HW1 will be released by Friday
  - 3 weeks to finish
- Readings of week 3:
  - Nvidia documentation

Today



Dataflow Graph

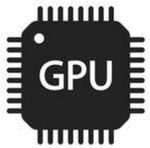
Autodiff

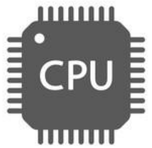
Graph Optimization

Parallelization

Runtime: schedule / memory

Operator optimization/compilation

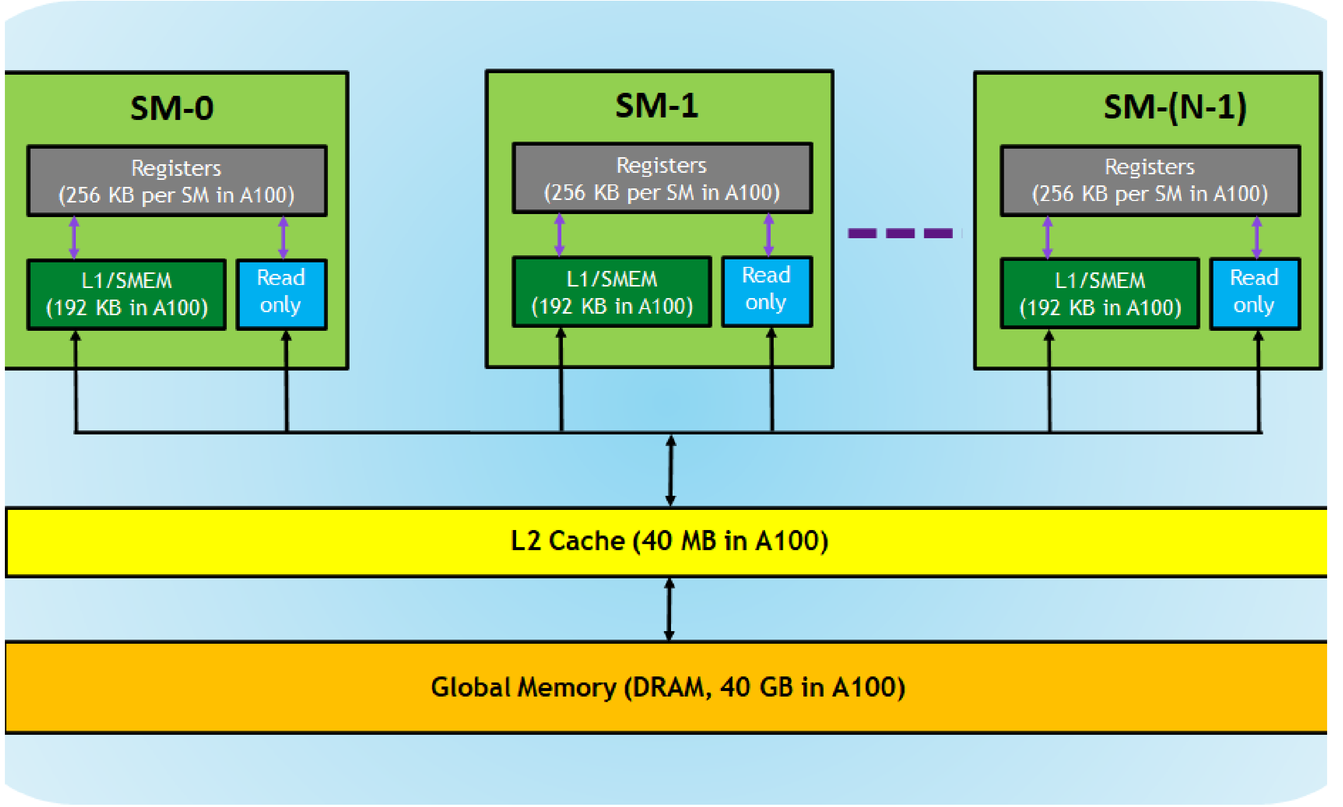




GPU and CUDA

- Basic concepts and Architecture
  - Concepts
  - Execution Model
  - Memory
- Programming abstraction
- Case study: Matmul
- Case study: parallel reduction

GPU HW Architecture Overview



Threads, Blocks, Grids

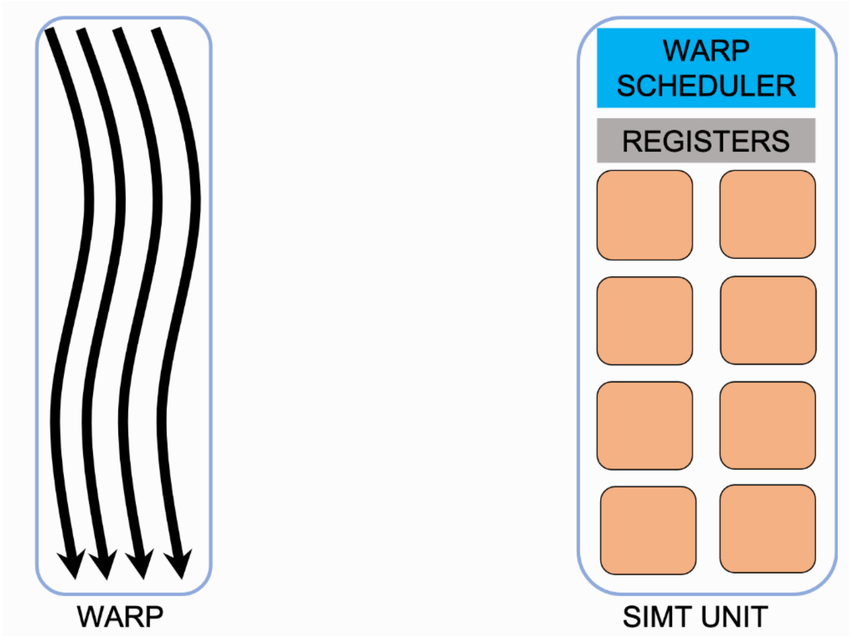
- **Kernel:** CUDA program executed by many CUDA cores in parallel
- **Threads:** smallest units to process a chunk of data
- (**Warp:** a group of threads without communication)
- **Blocks:** A collection of threads that share memory
- **Grid:** A collection of blocks that execute the same kernel

Threads

- Recall Threads vs Process. Naming 2 primary differences?

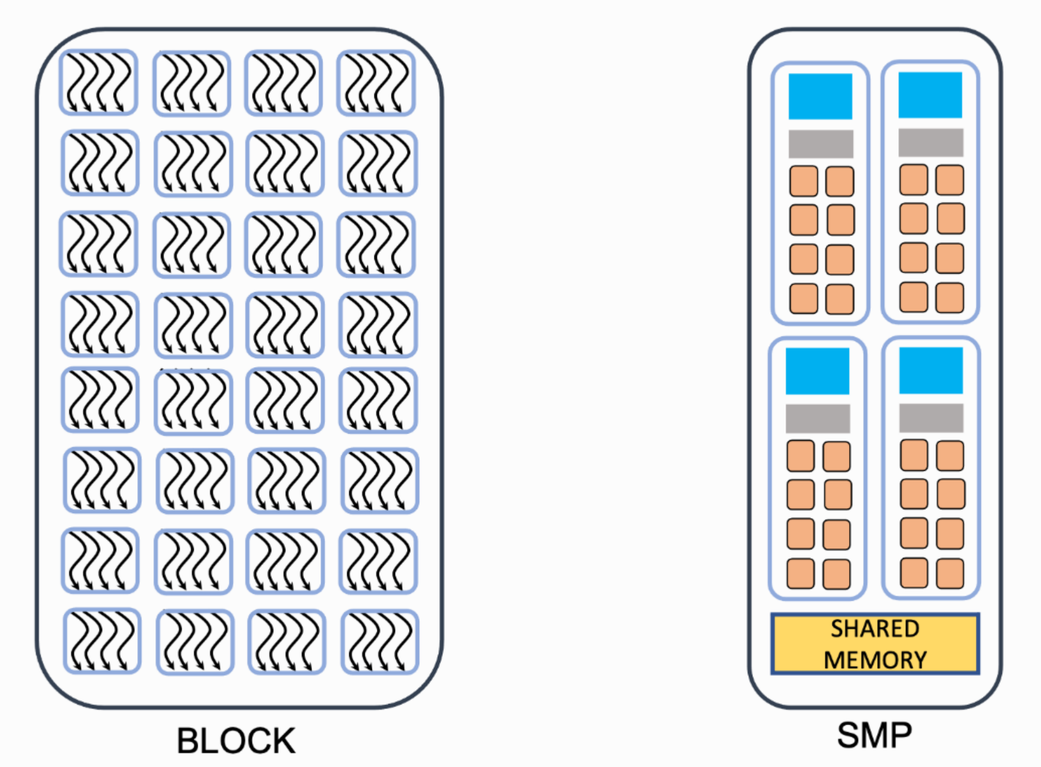
Warp (between threads and blocks)

- Warp: a group of threads executing the same instructions
  - Cannot share data
  - Lock-step execution
  - Finest scheduling granularity by GPUs



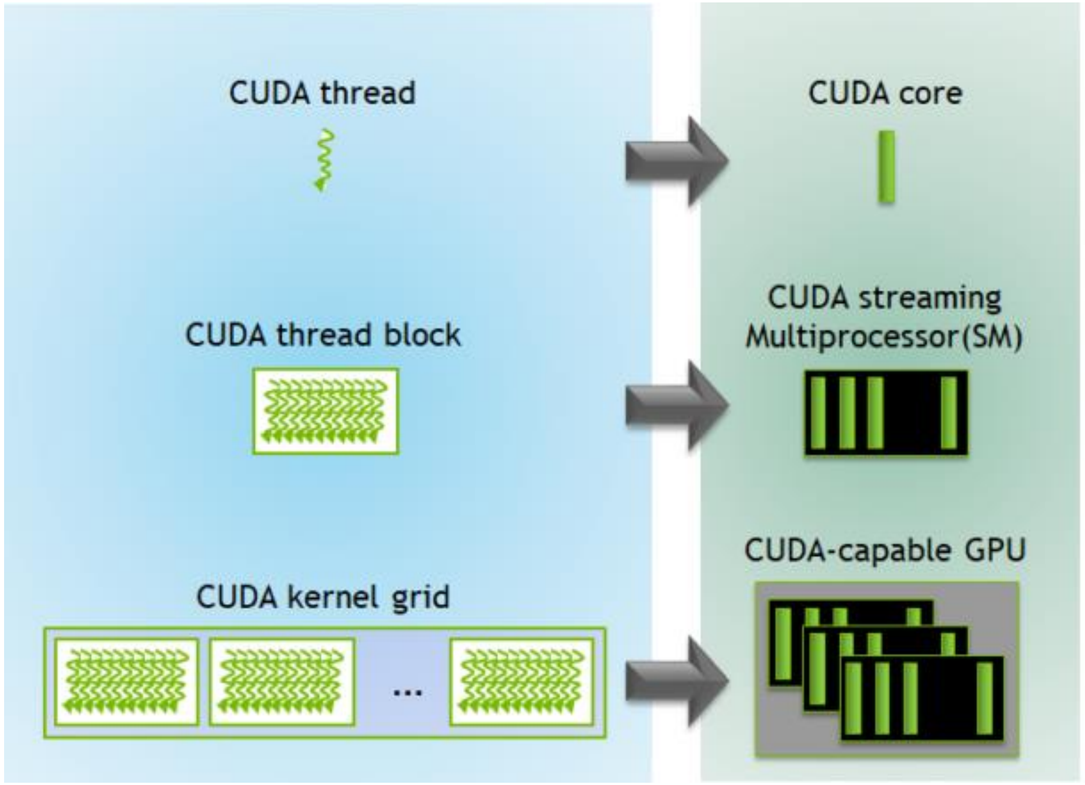
Thread Block

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



Grid

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



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

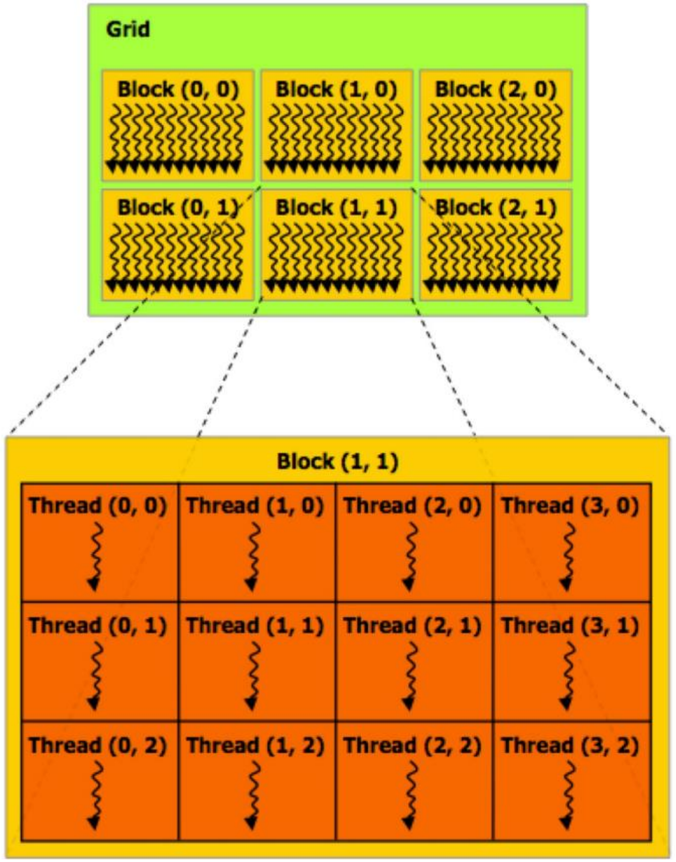
How many SMs/Threads we have?

- 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

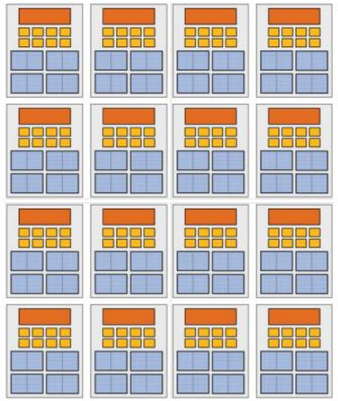
- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming GPUs
- CUDA's design matches the grid/block/thread concepts in GPUs

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


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

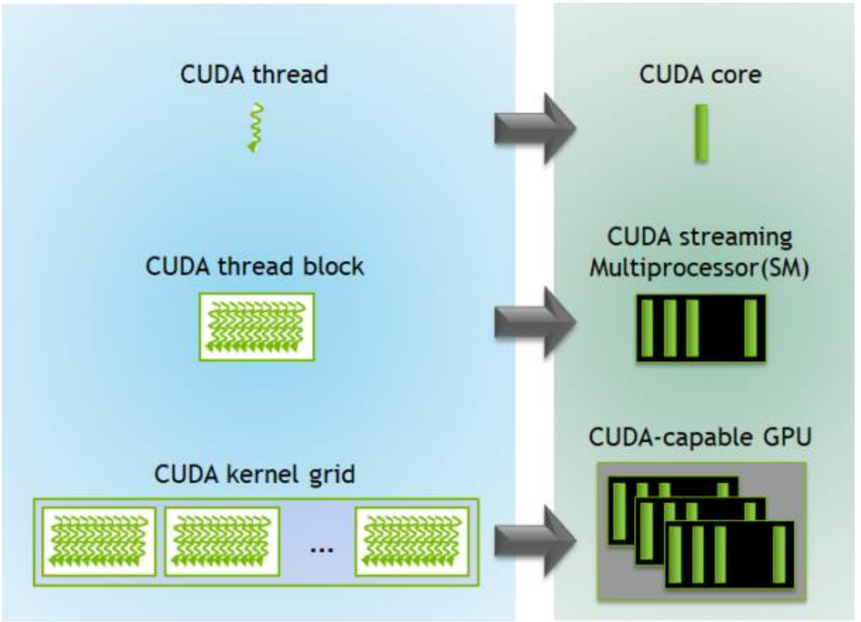
Run on CPU



How Many threads/Blocks it runs on?

- How many blocks it runs on?
- How many threads it runs on?

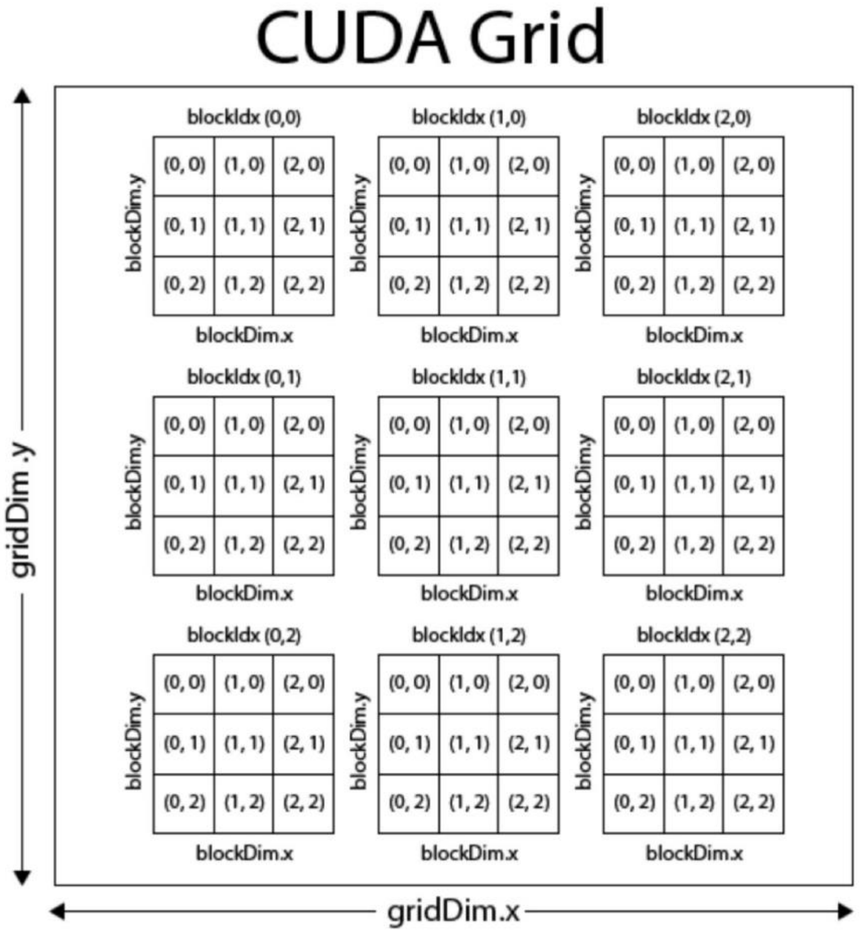
```
const int Nx = 12;
const int Ny = 6;

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


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

```
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 computation

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

Hence it is Important to:

- Check boundary conditions

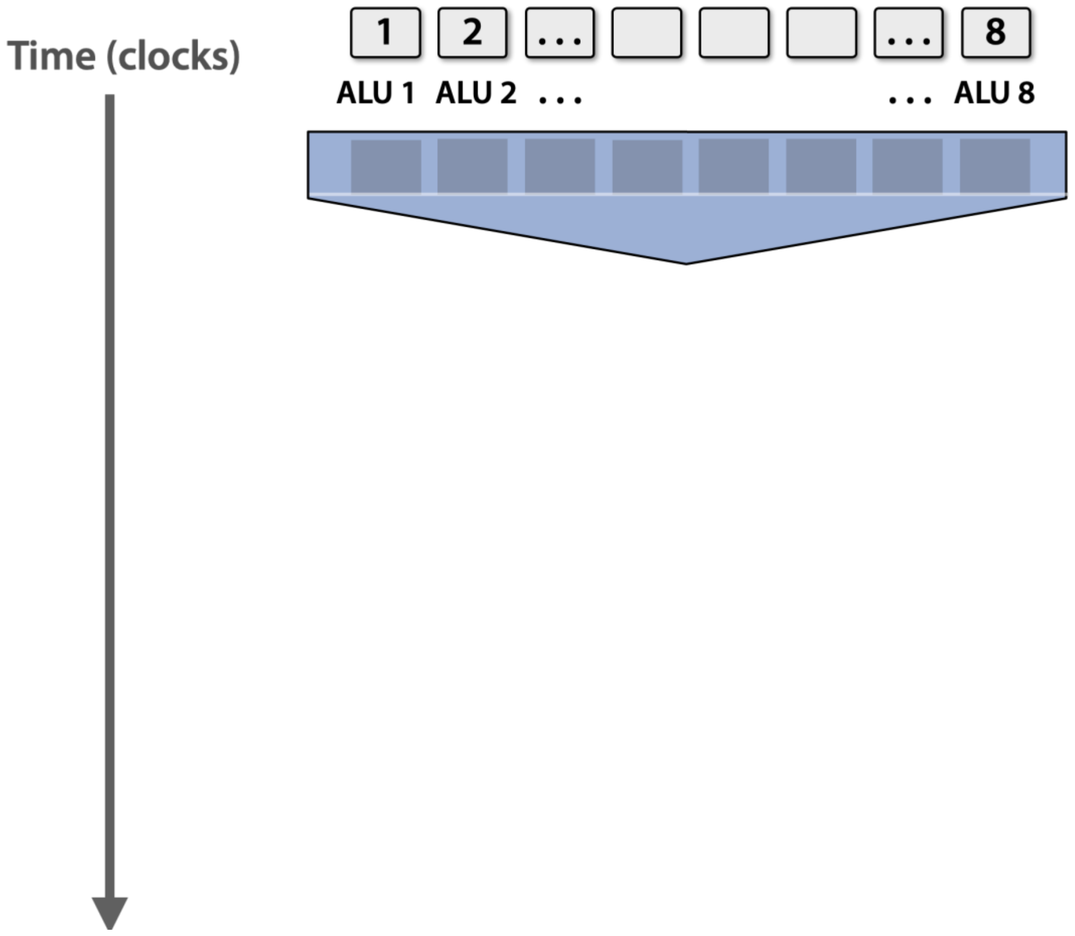
SIMD Constraints: how to handle control flow?

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

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

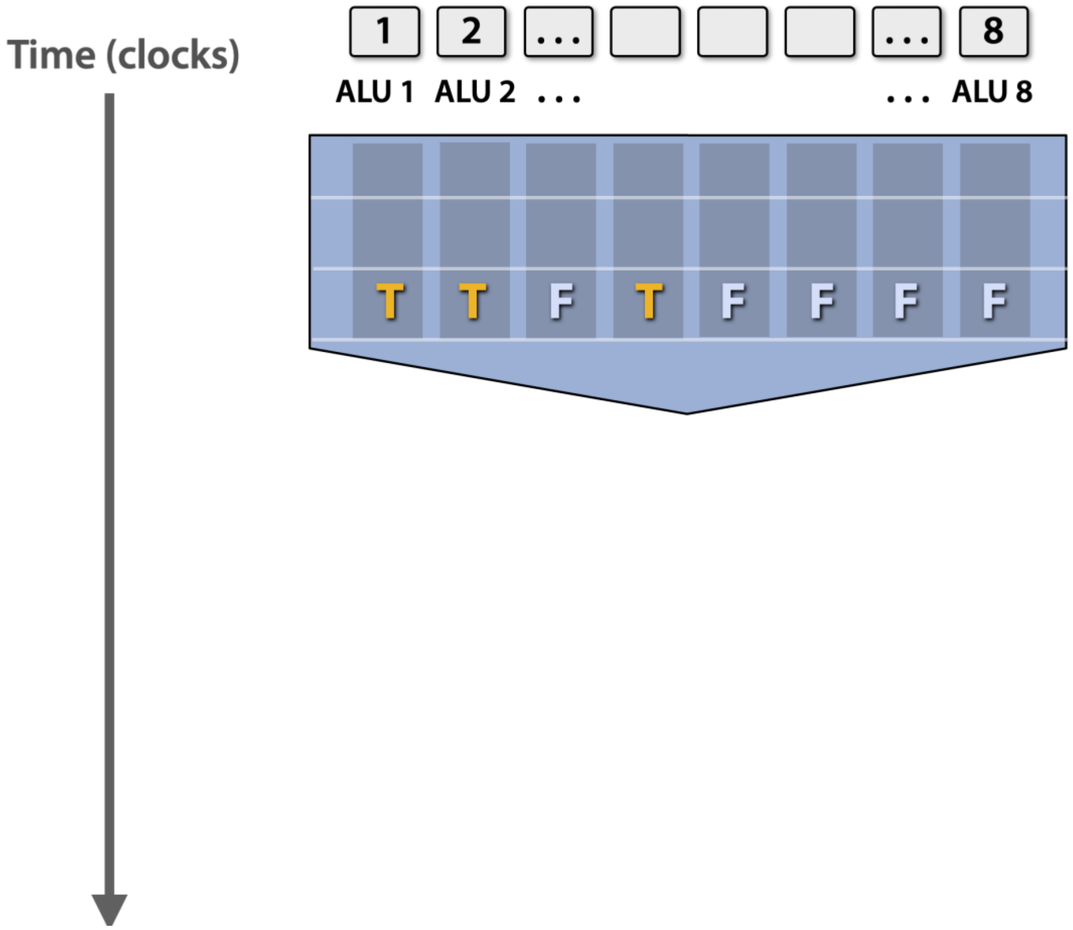
```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



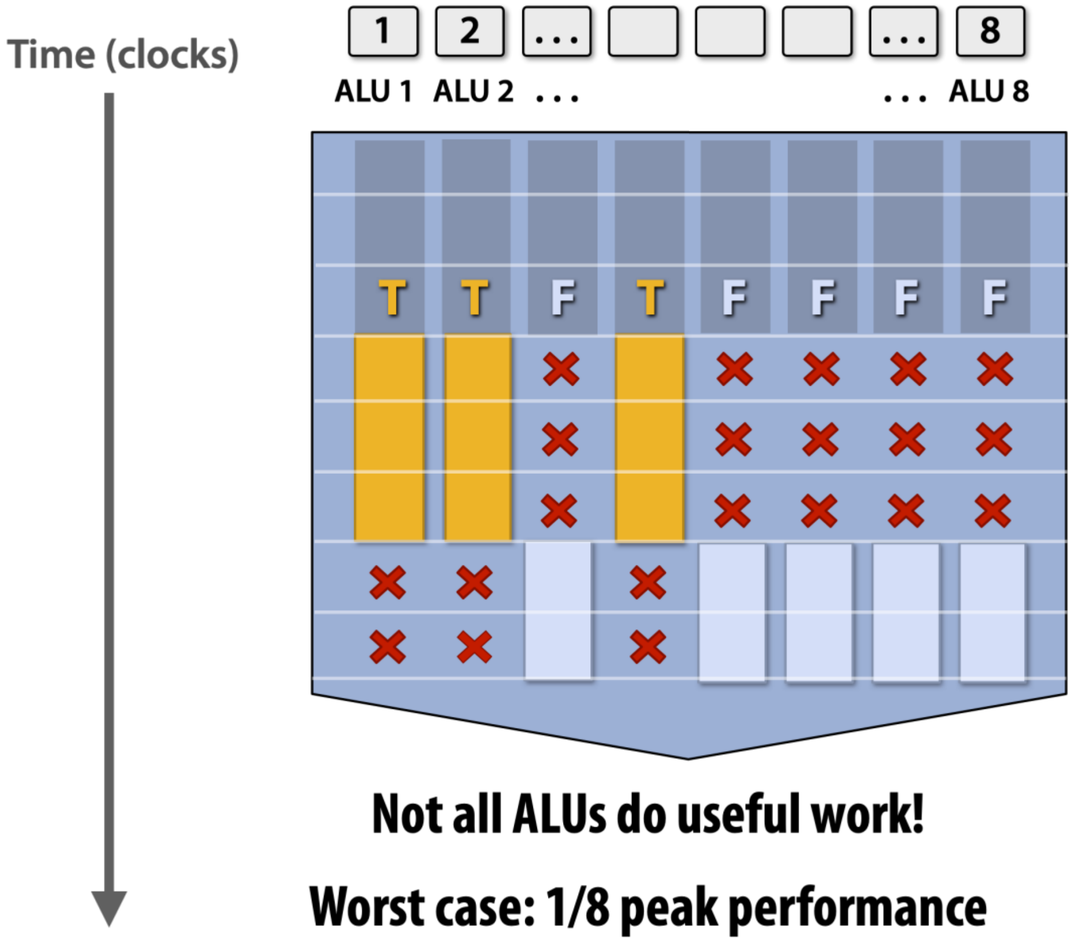

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



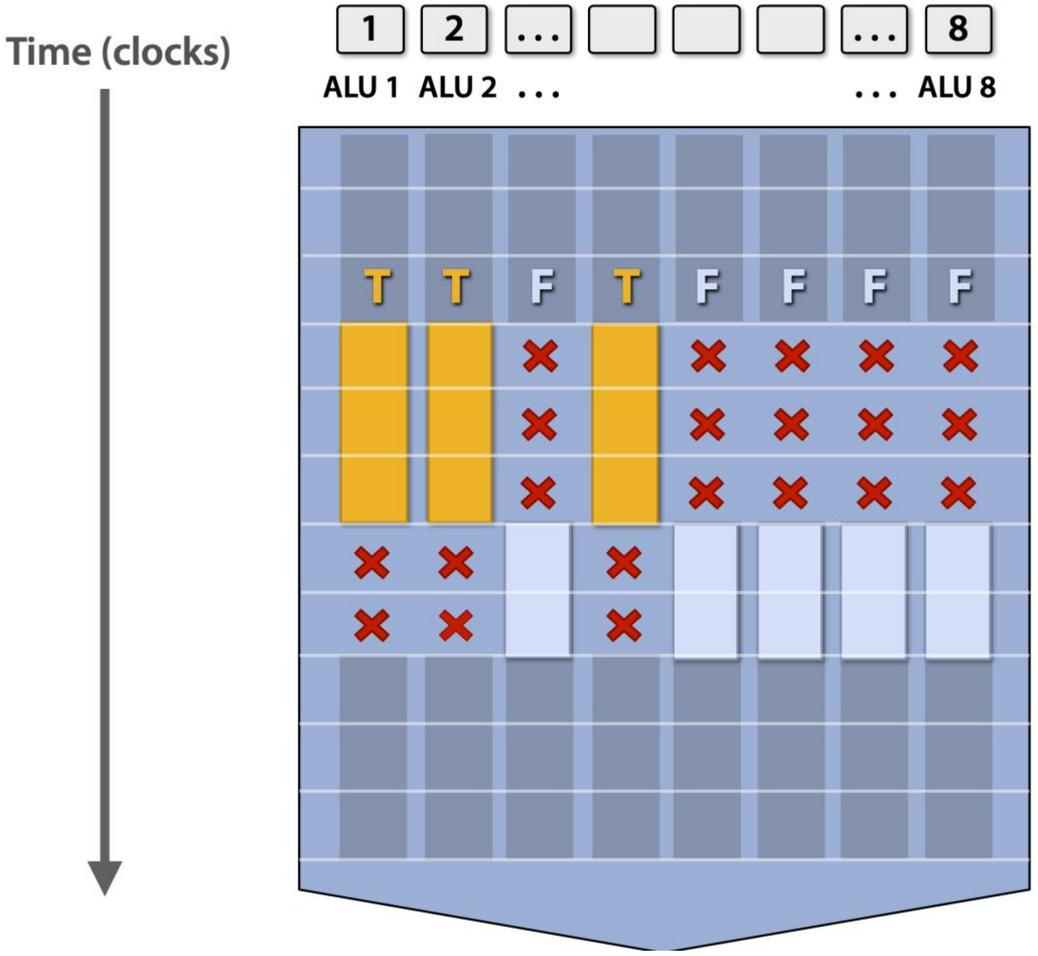

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

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;
}
```

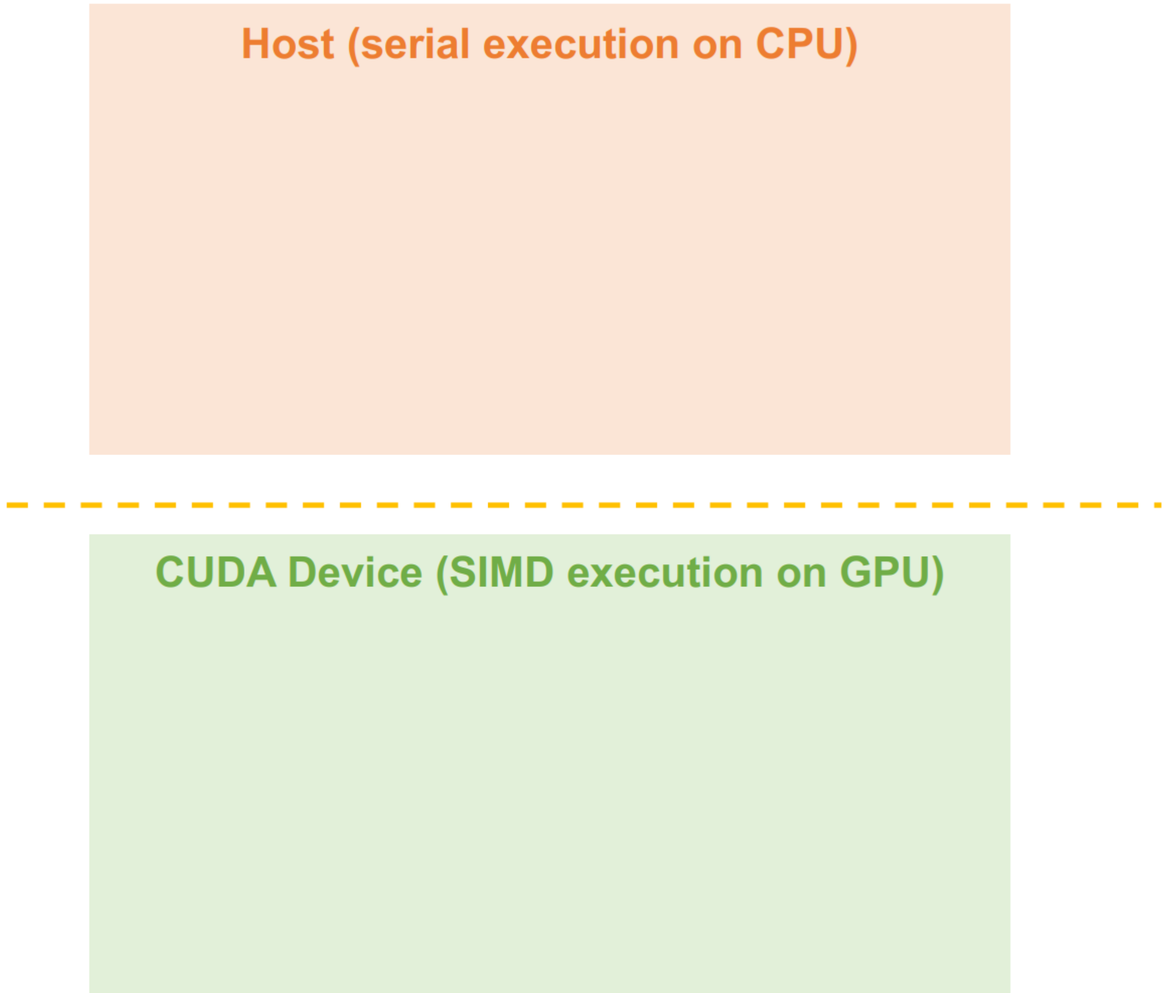
Coherent vs. Divergent

- Coherent execution:
  - Same instructions apply to all data
- Divergence Execution:
  - On the contrary of coherent
  - Should be minimized in CUDA programs
  - Is this the case for CPU cores?

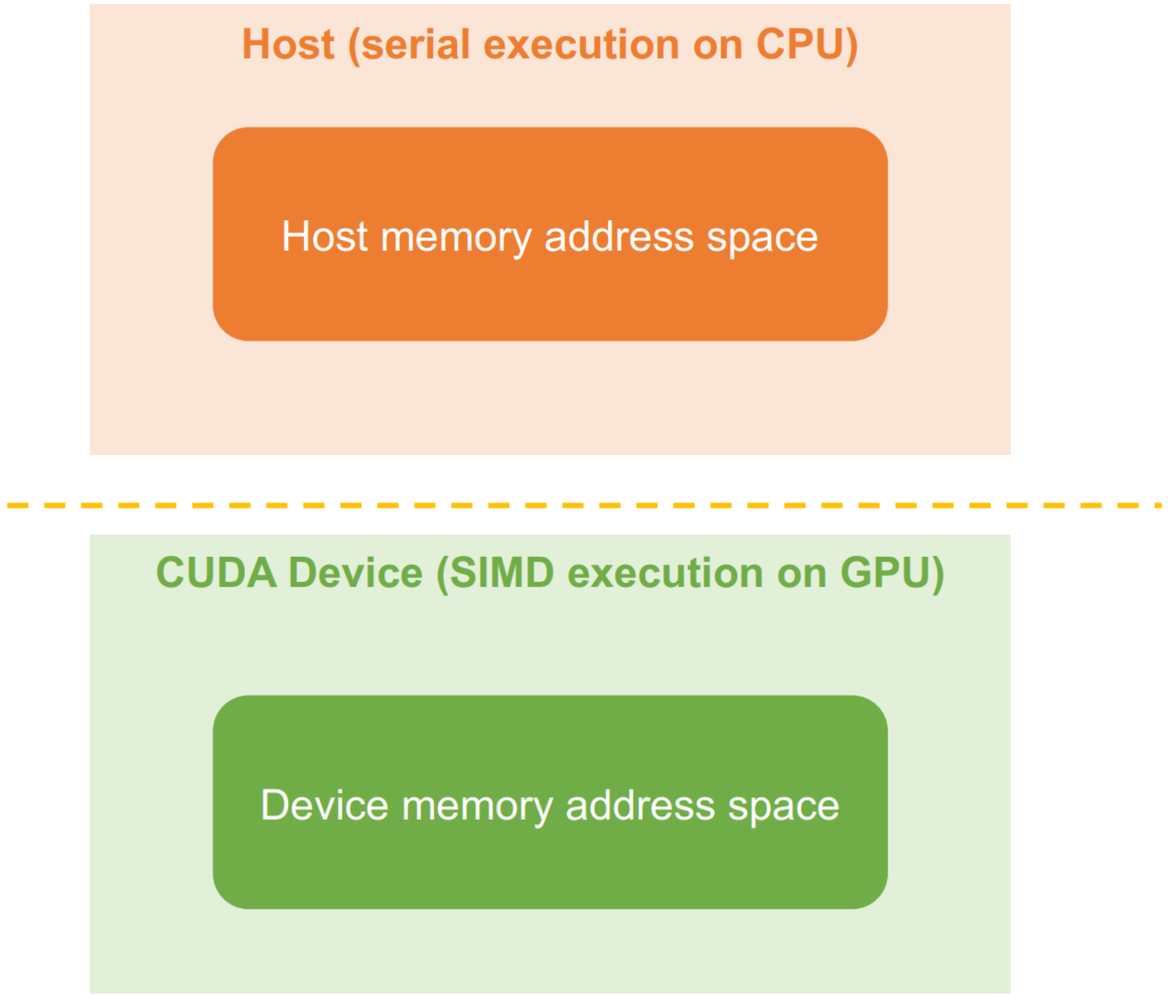
GPU and CUDA

- Basic concepts and Architecture
  - Concepts
  - Execution Model
  - **Memory**
- Programming abstraction
- Case study: Matmul
- Case study: parallel reduction

CUDA Memory Model



CUDA Memory Model



Concepts:

- Host memory: RAM
- Device memory: GPU memory

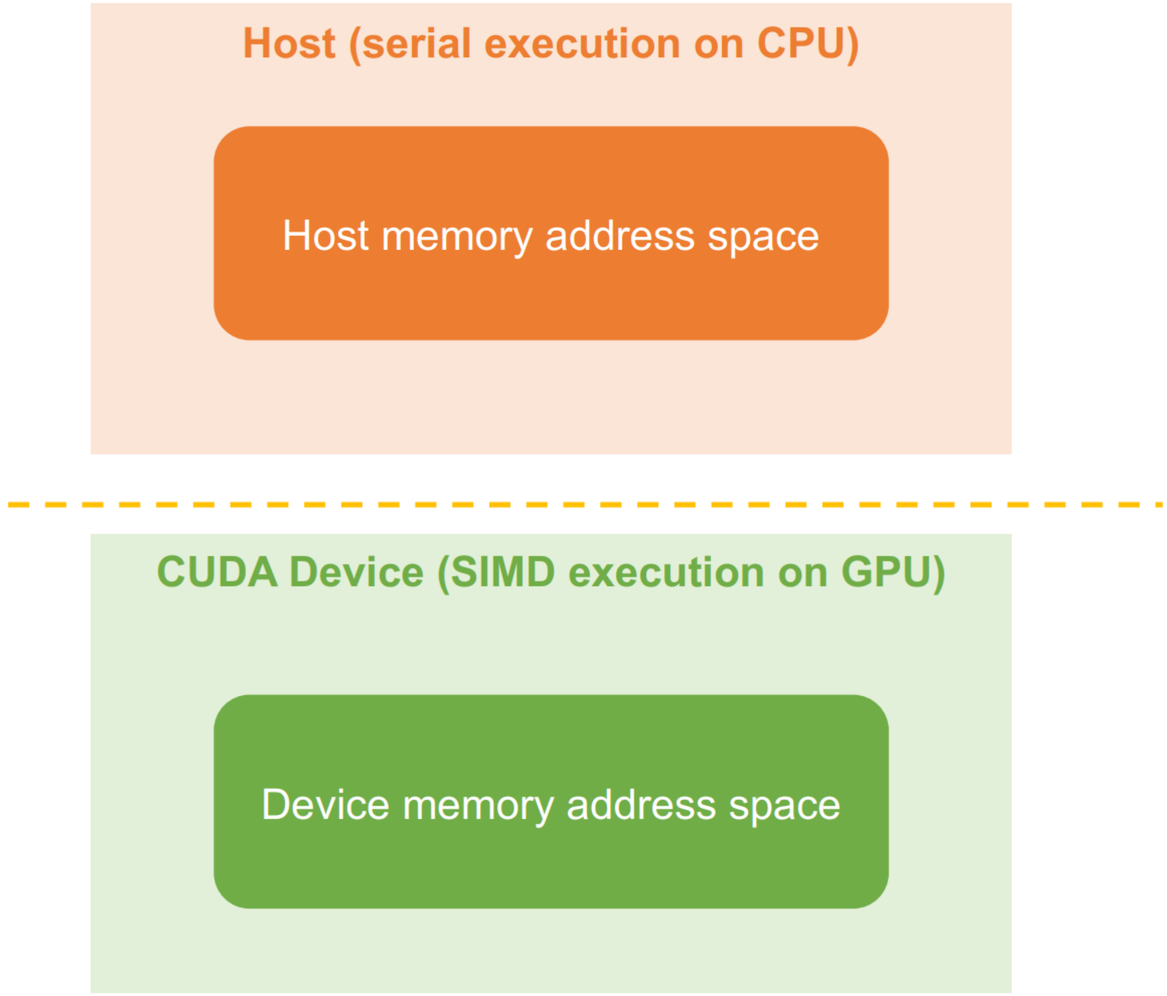
Q:

- 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



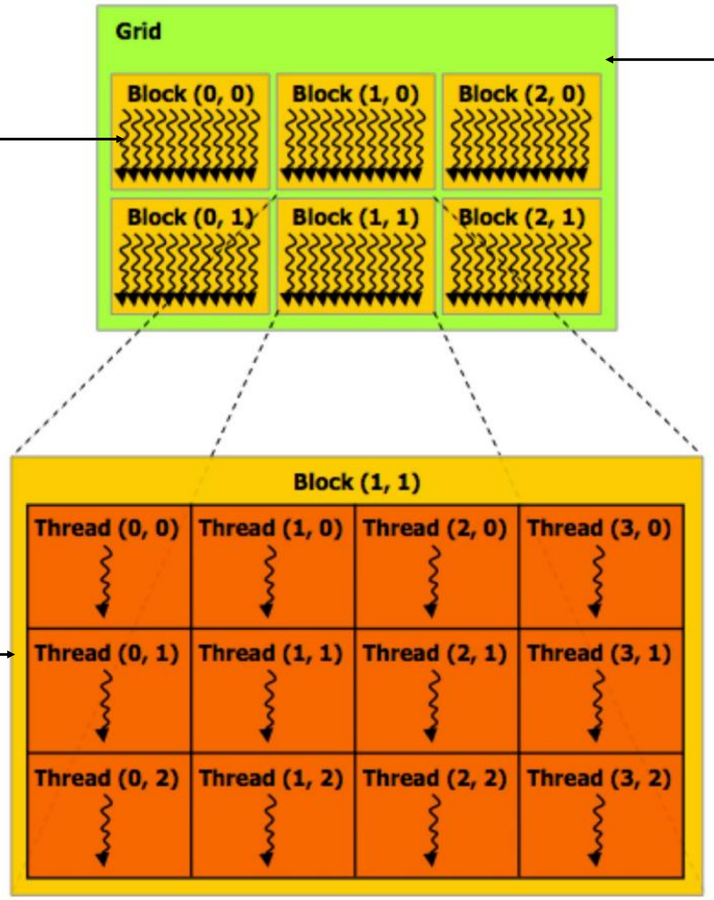

```
float* A = new float[N];

// 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)



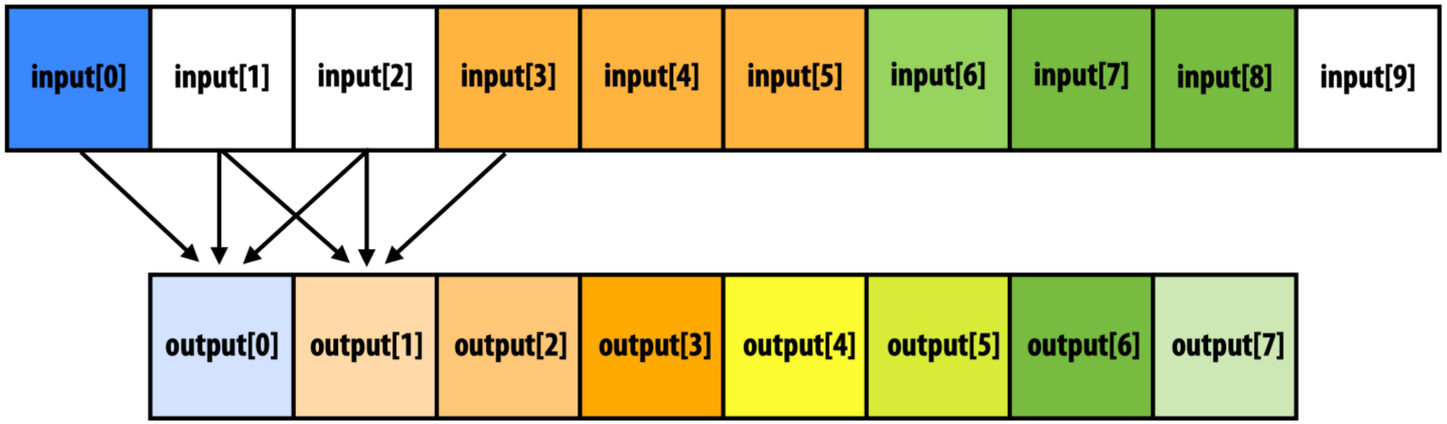
Global device memory
(r/w by all threads)

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

Why make it so complex:

- Balance between speed and shared-memory parallelism

Why So Complex: Example

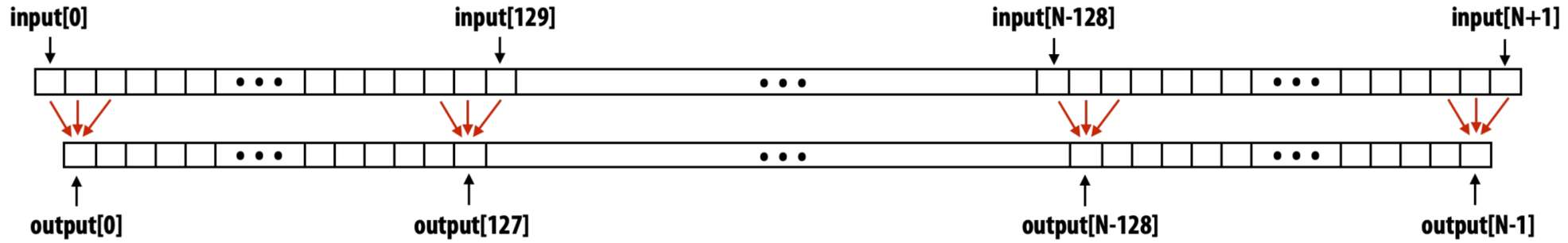


- CPU implementation:

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

Q: what is the parallelizable part?

GPU Version 1



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

GPU Version 1

```
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 blocks?
- What if number of thread requested > total threads in GPUs?

GPU Version 1

```
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

GPU Version 2

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:

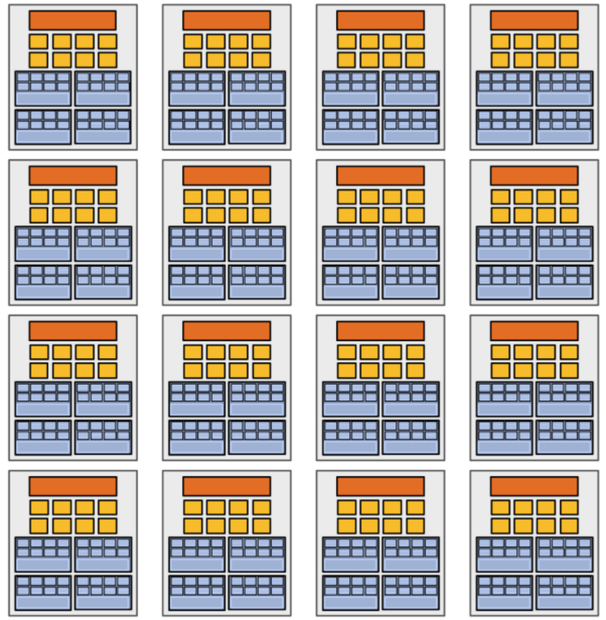
Program text (instructions)
Information about required resources:

- 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

- Yet the user asks for a static number of blocks

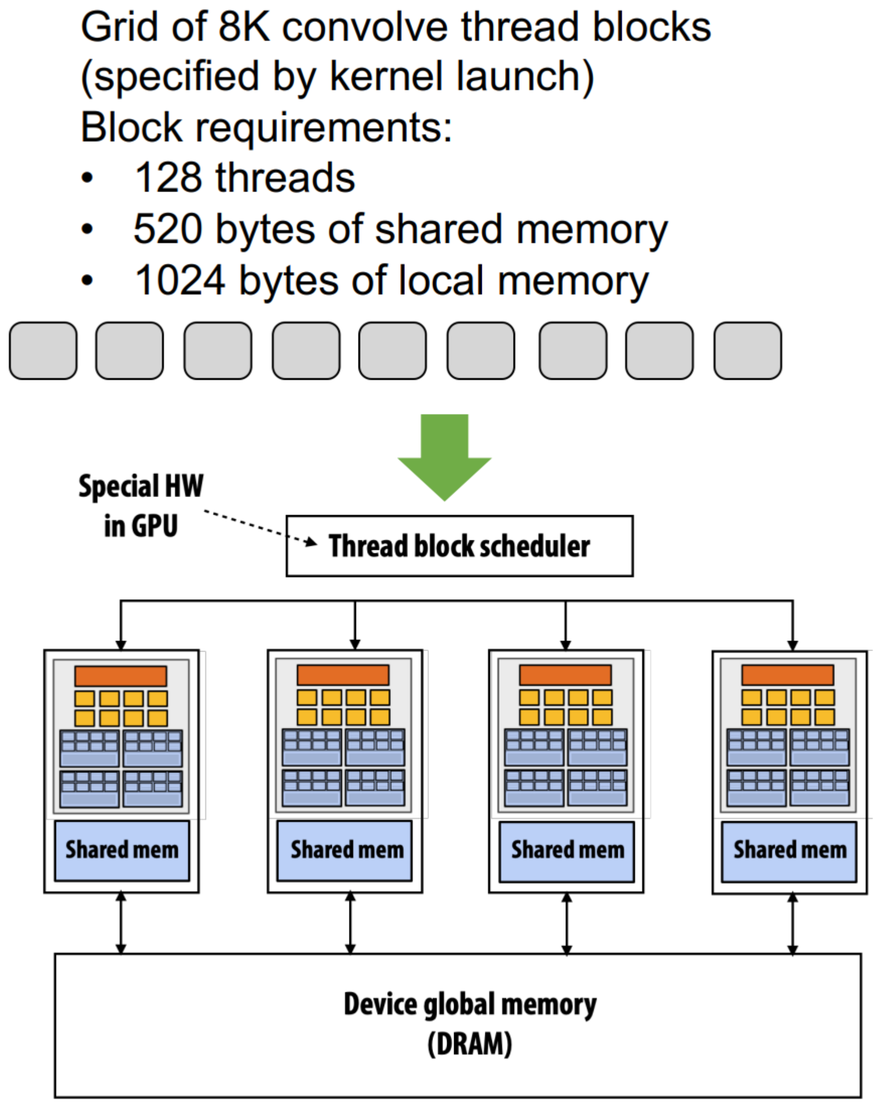
Mid-range GPU (6 cores)



High-end GPU (16 cores)

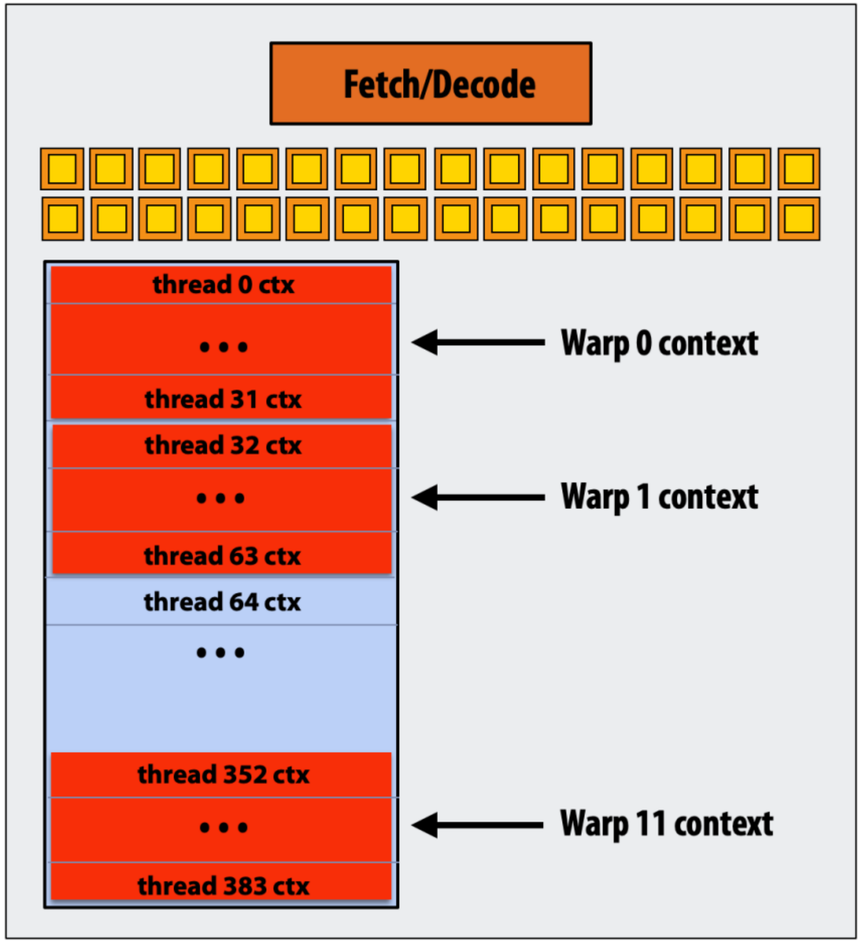
Scheduling on CUDA

- 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
- Each SM: 96KB of shared memory
- Max warp context: 64



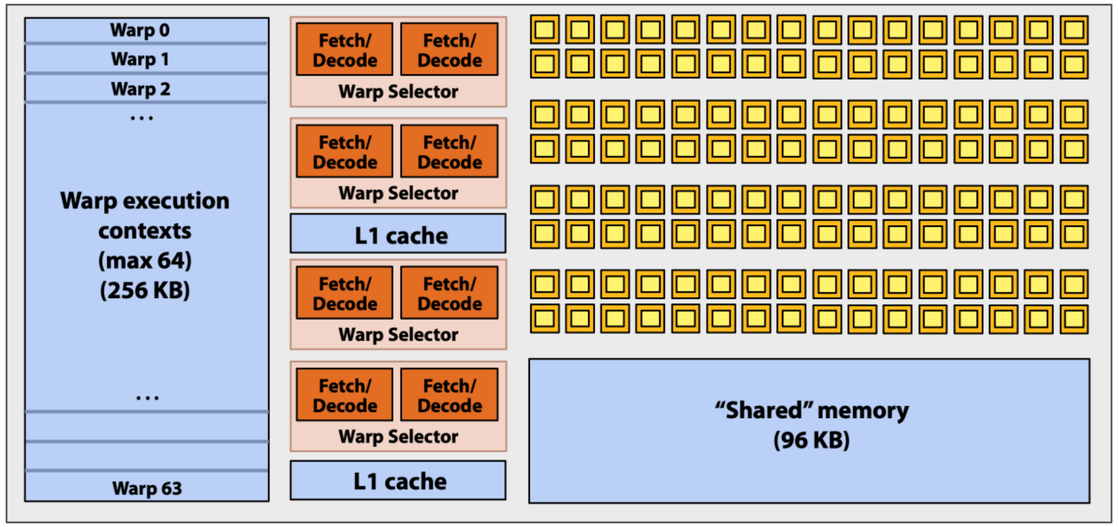
What is a warp?

- A warp is a CUDA implementation detail on NVIDIA GPUs
- •On modern NVIDIA GPUs, groups of 32 CUDA threads in a thread block are executed simultaneously using 32-wide SIMD execution



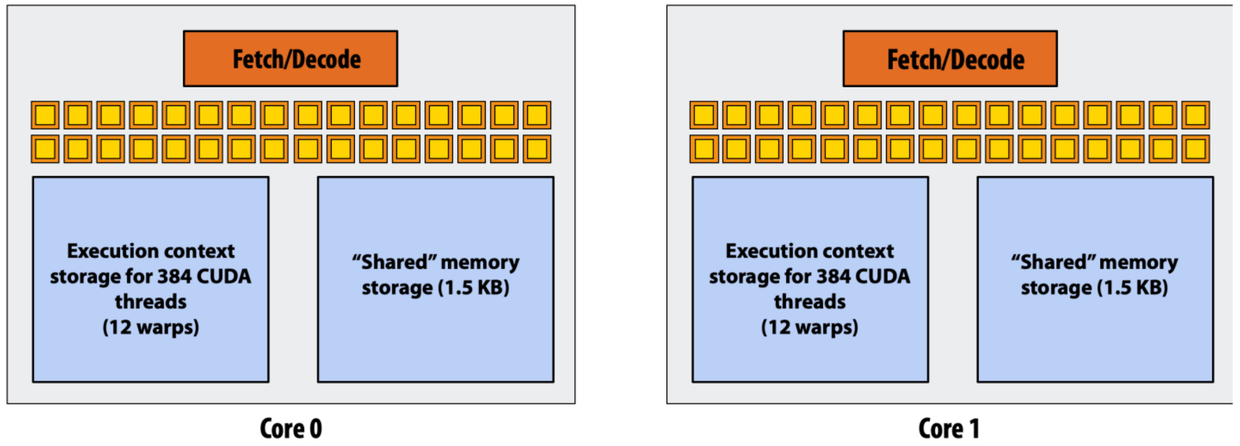
Running a ThreadBlock on an SMP

- Warp: a group of 32 CUDA threads shared an instruction stream
  - In our case: we need 4 warps (4 x 32 /warp = 128 threads)
- SMP operation each clock:
  - Select up to 4 runnable warps from 64 resident on an SMP
  - For each warp, 32 threads to execute the instructions



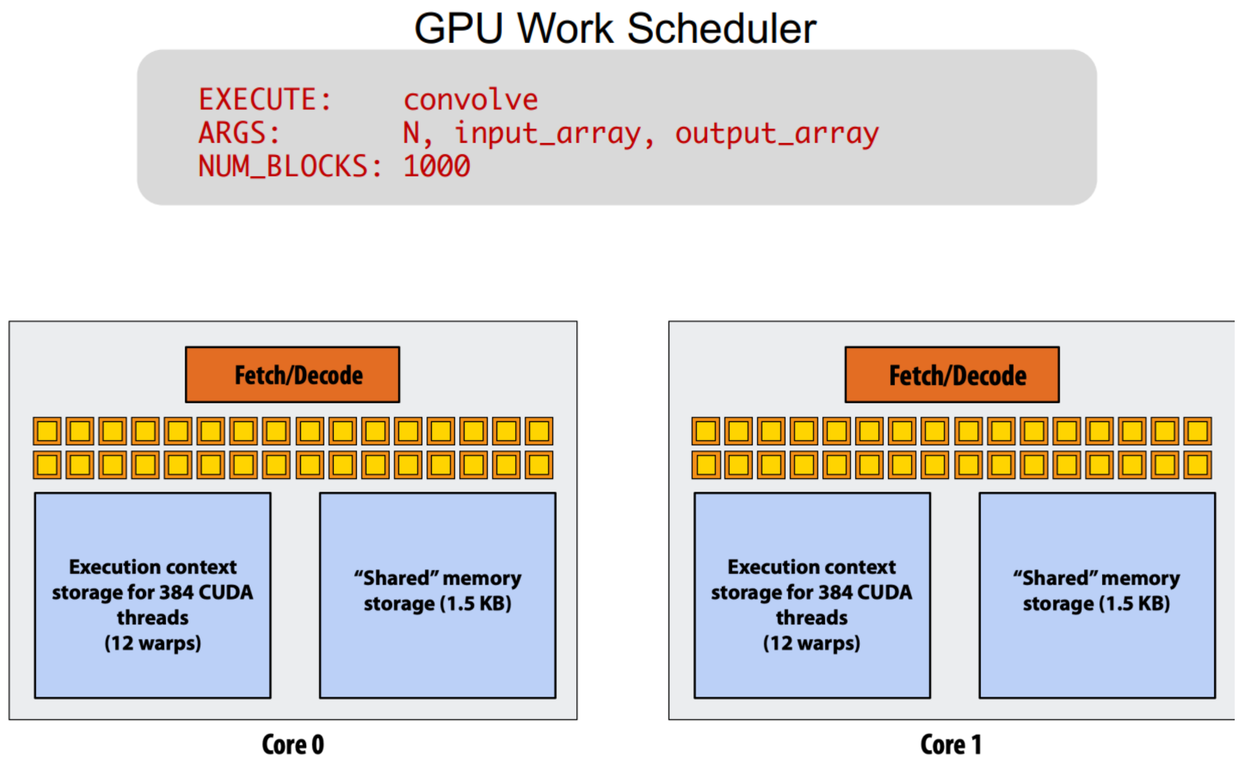
Deep Dive into CUDA scheduling

- Conv1d spec on 1024 x 1024
  - 128 CUDA threads / threadblock
  - 1024 blocks
  - Each threadblock asks for 130 * 4 = 420 bytes of shared memory
- Given: a GPU with two SMs, specs below
- How is the scheduling looking like?



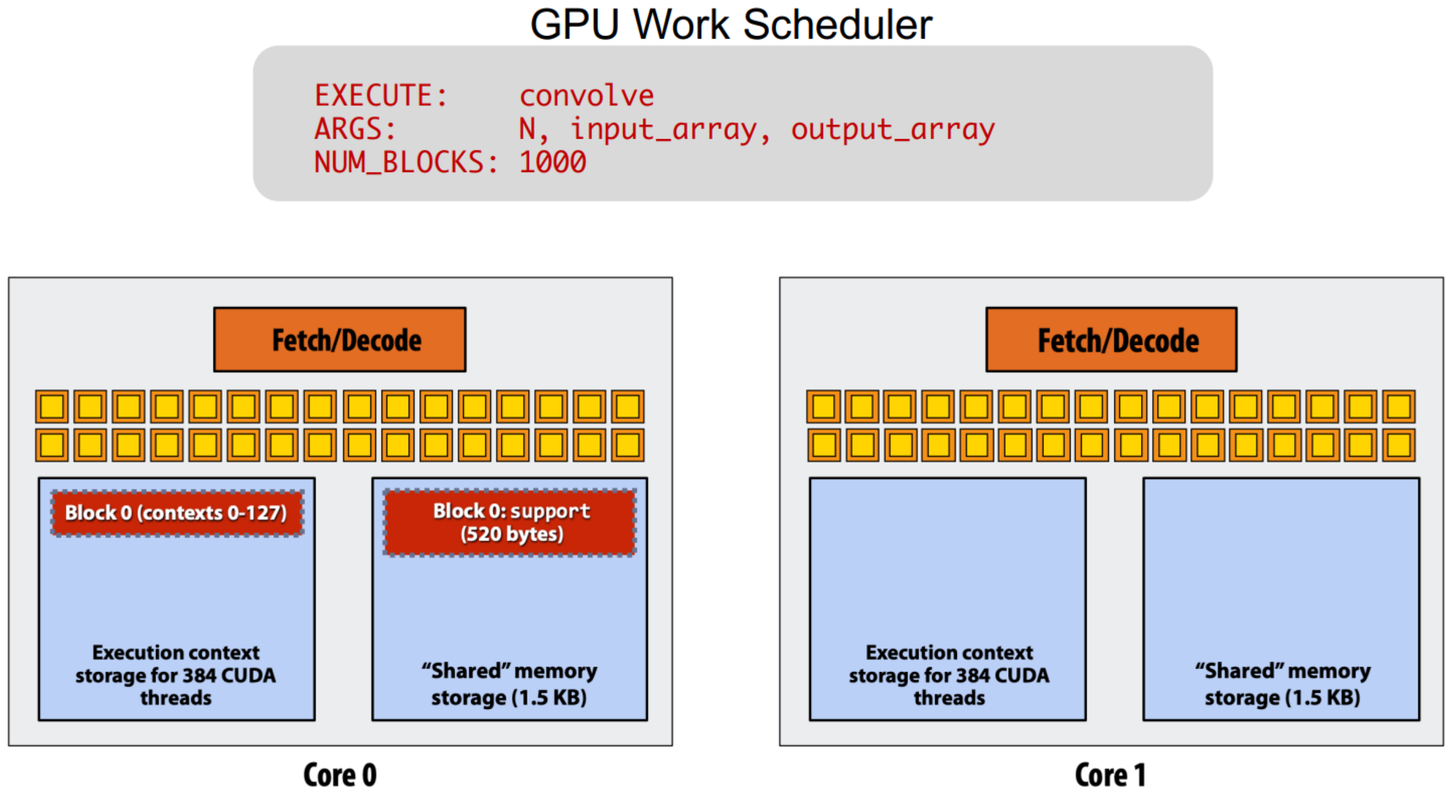
Deep Dive into CUDA scheduling

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



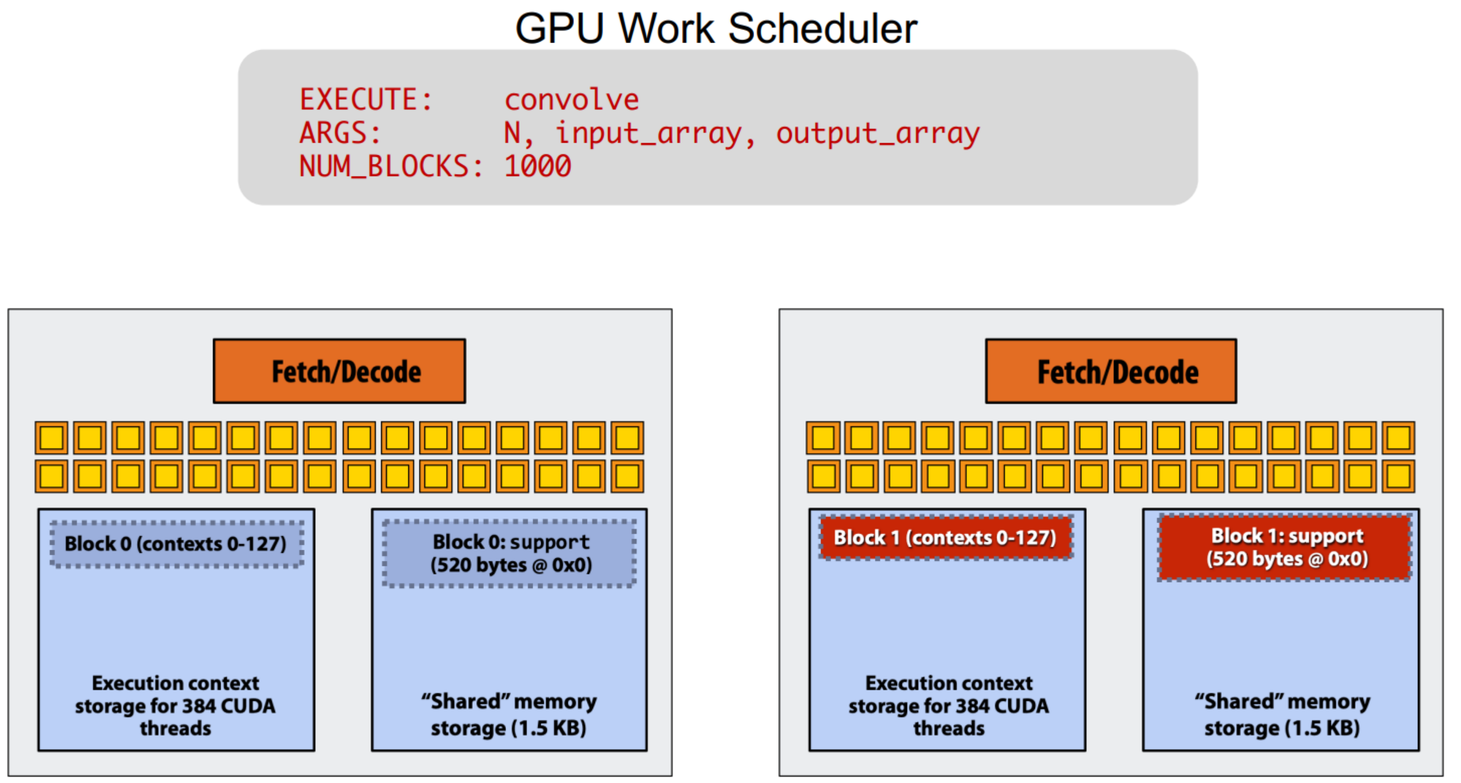
Deep Dive into CUDA scheduling

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



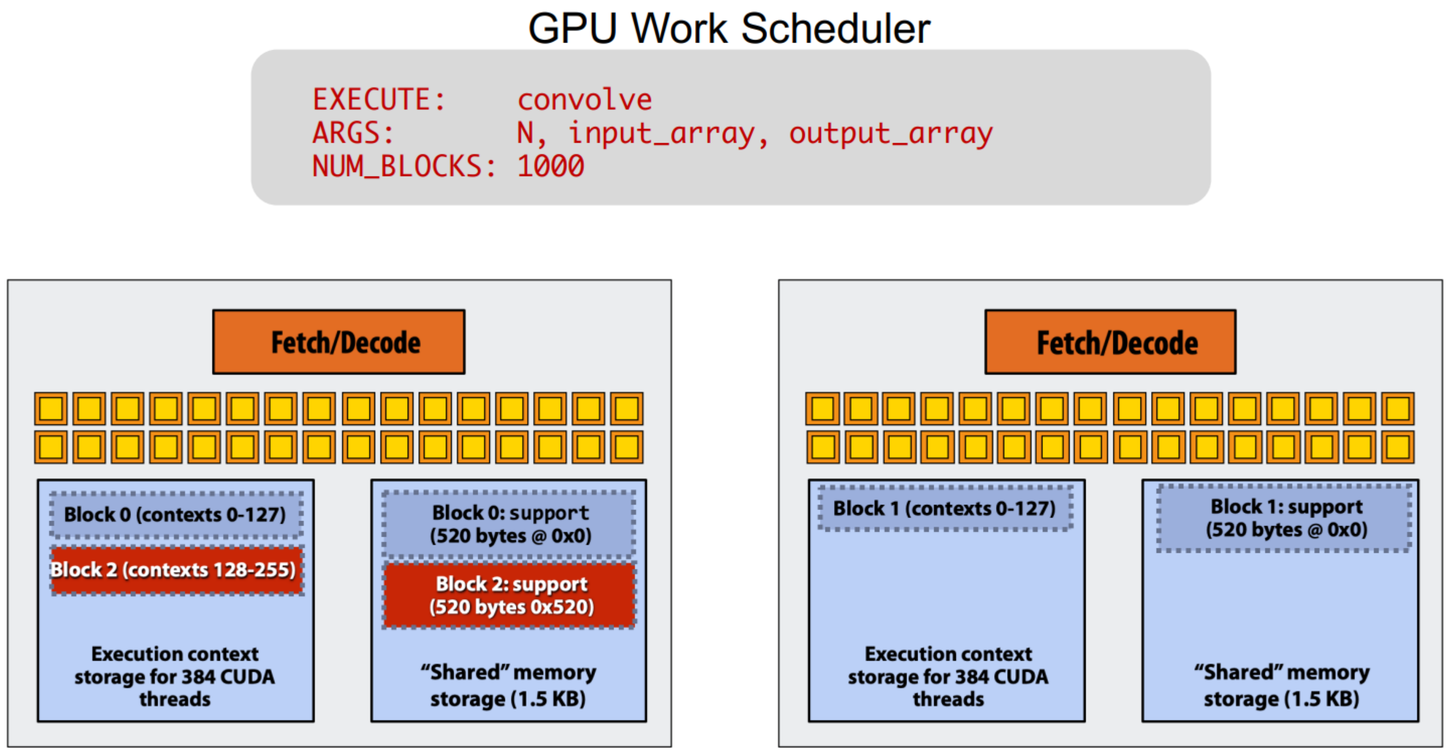
Deep Dive into CUDA scheduling

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



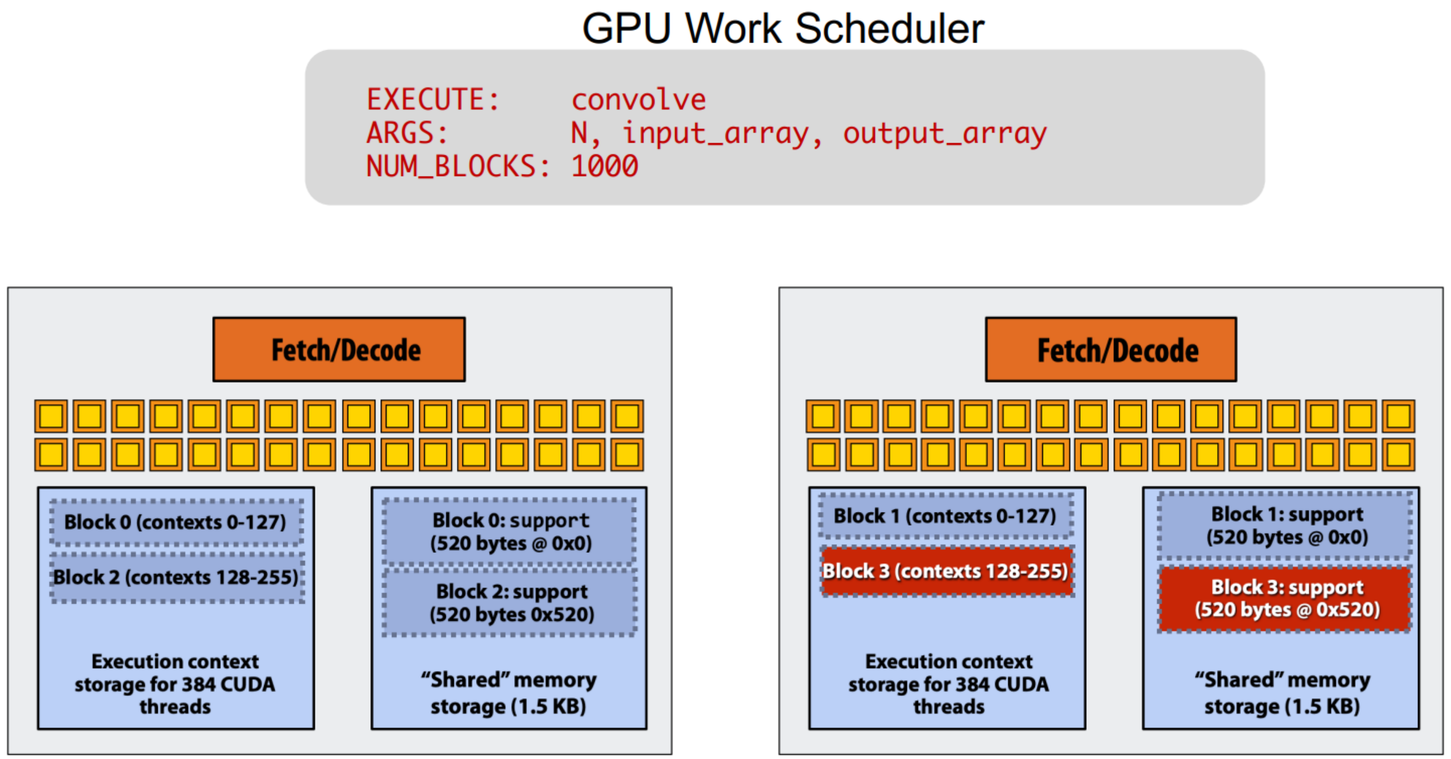
Deep Dive into CUDA scheduling

- 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

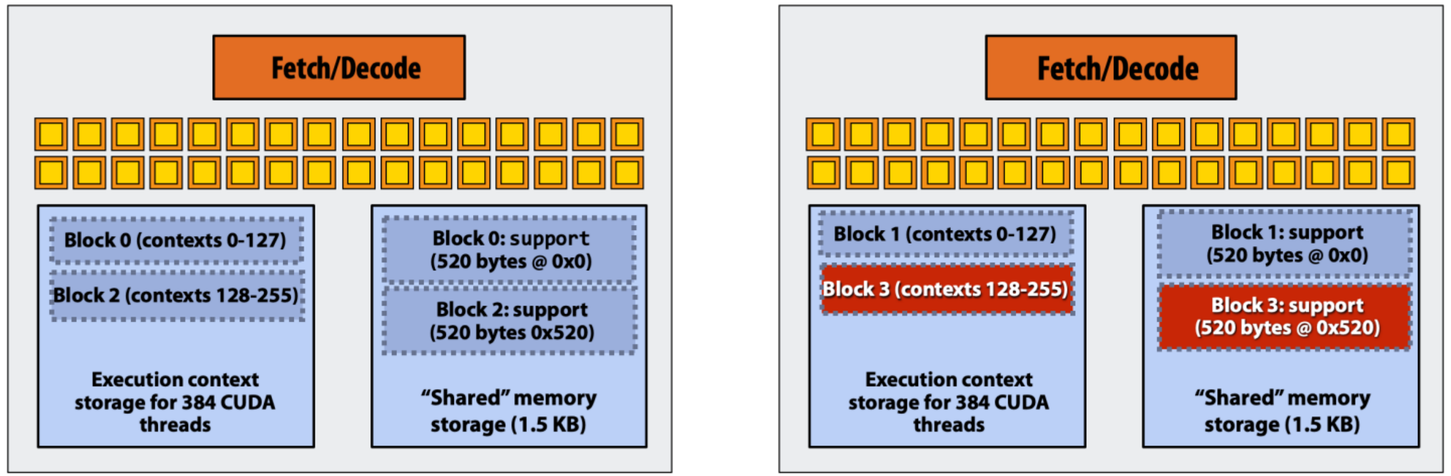


Deep Dive into CUDA scheduling

- Step 3: scheduler continues to map blocks to execution contexts
- But: cannot schedule a third block on core 0 or core 1. Why?

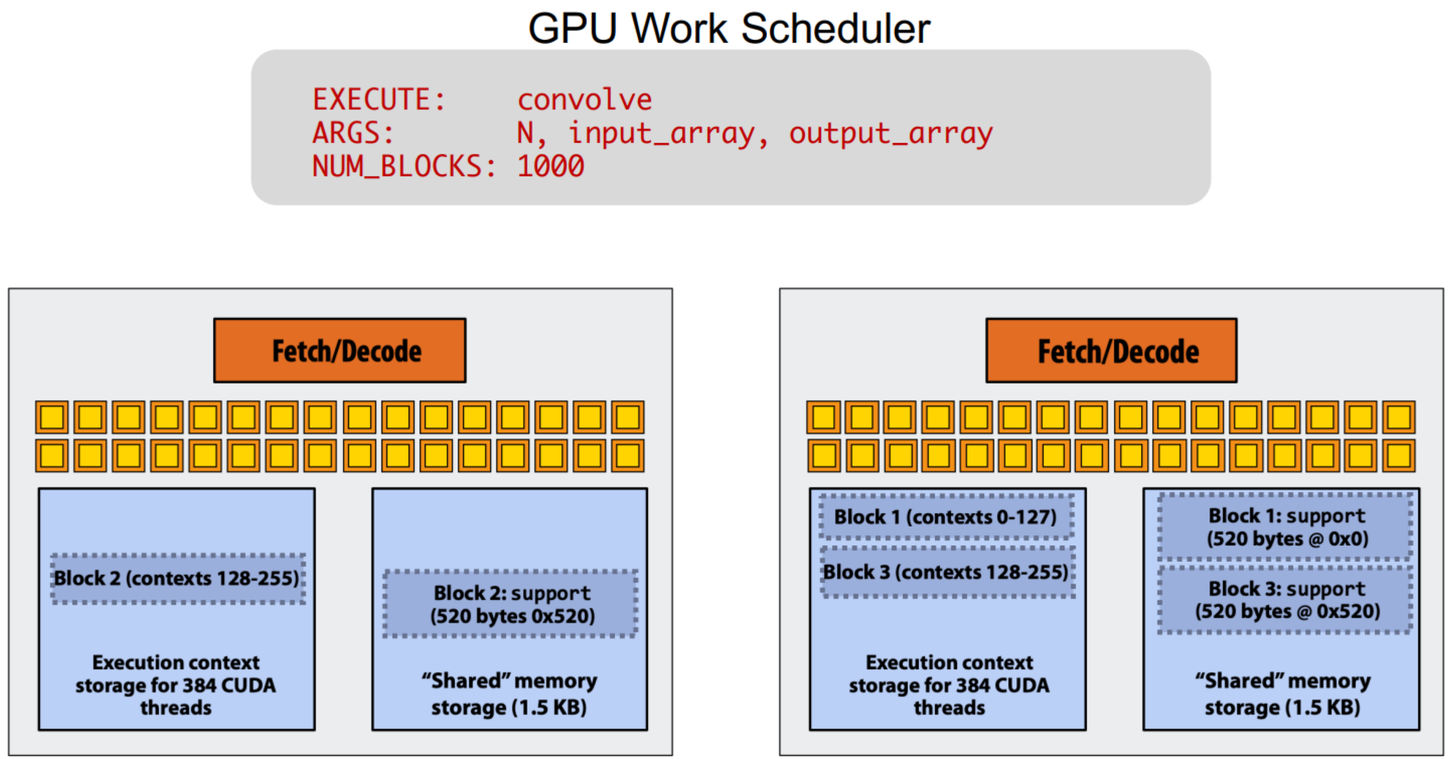
GPU Work Scheduler

```
EXECUTE:    convolve
ARGS:       N, input_array, output_array
NUM_BLOCKS: 1000
```



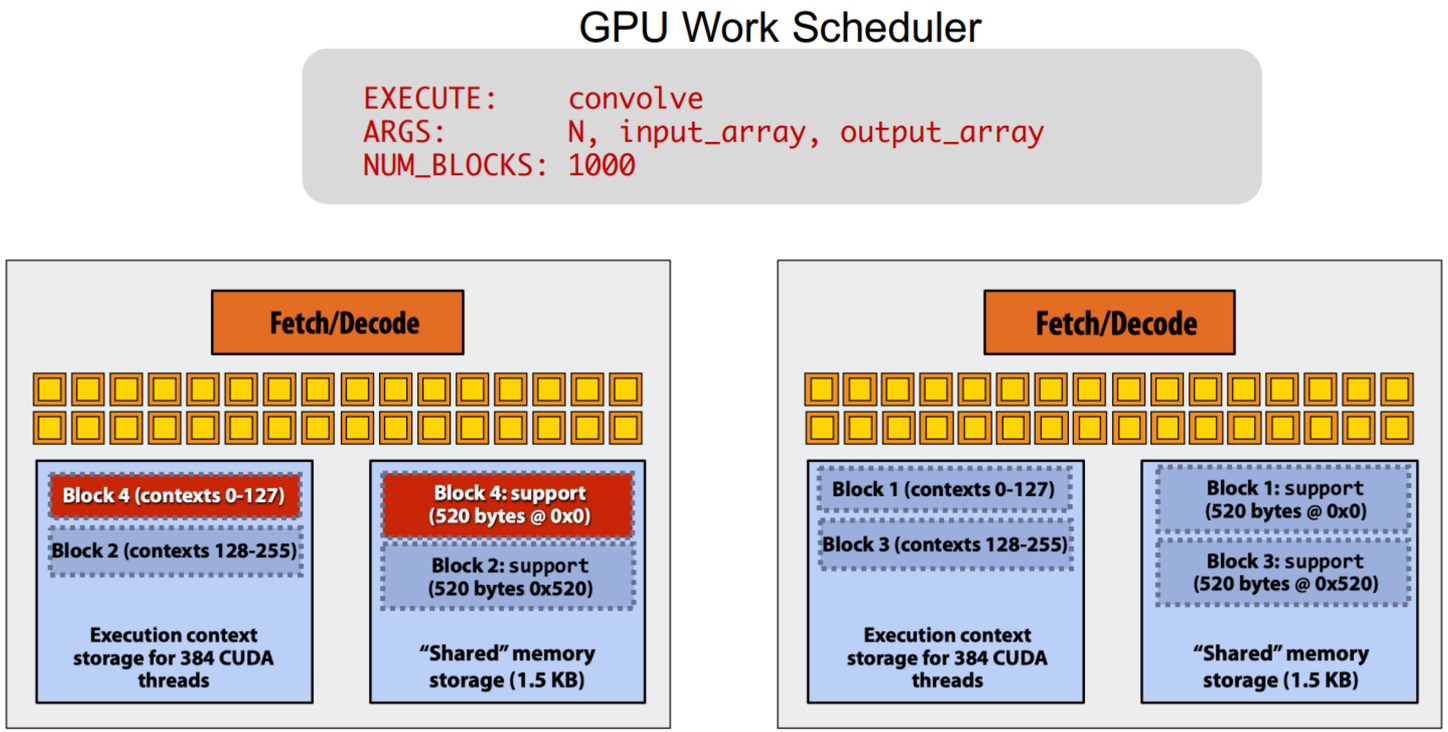
Deep Dive into CUDA scheduling

- Step 4: thread block 0 completes on core 0



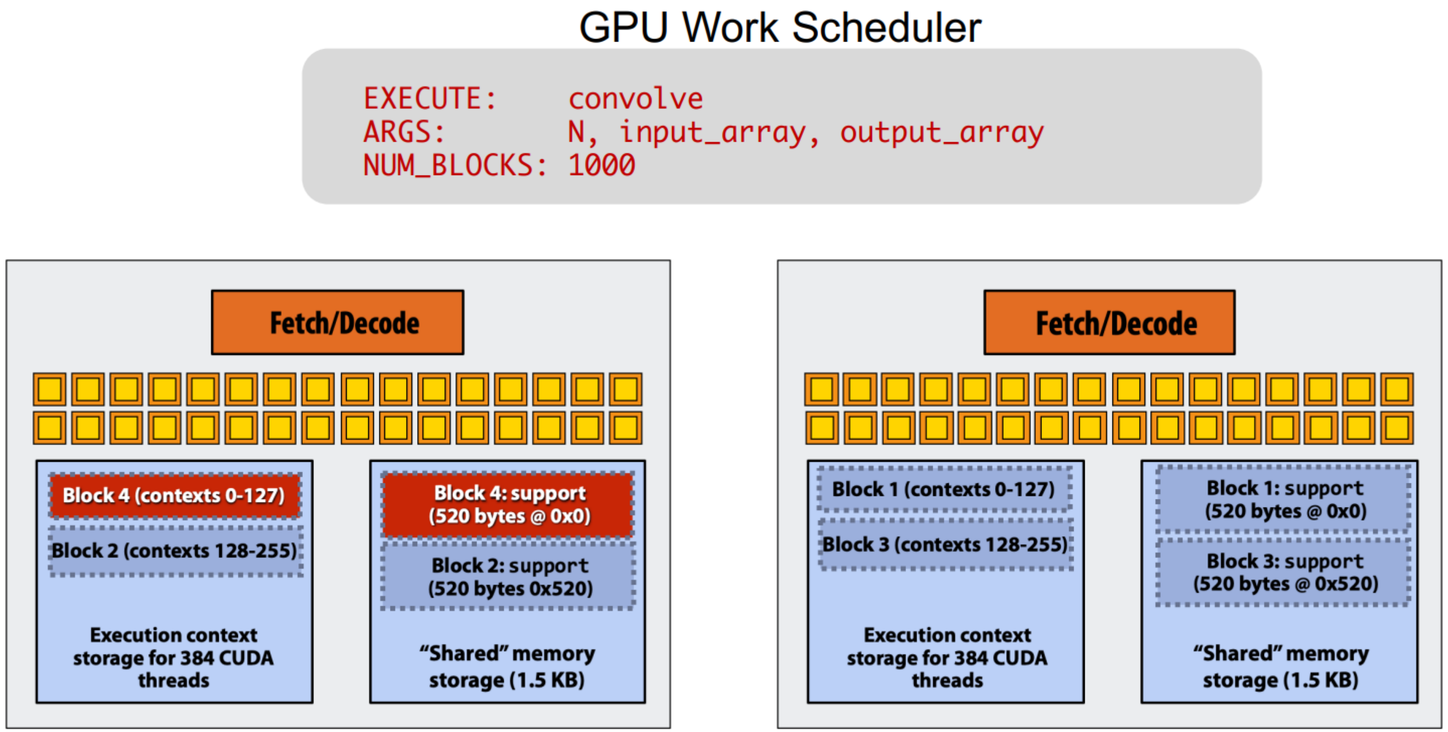
Deep Dive into CUDA scheduling

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



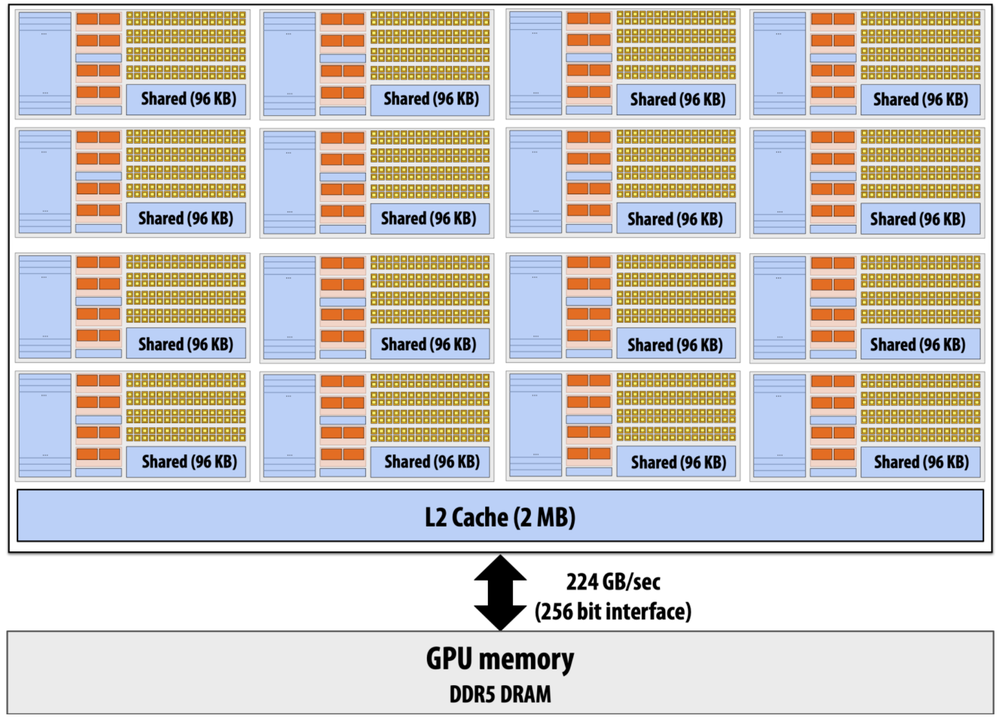
Deep Dive into CUDA scheduling

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



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

- SMM resource:
  - # warp context: 64
  - # threads: 64 *32 = 2048 total threads
  - 96KB of shared memory
  - 16 SMPs
    - 1.1GHz clock
    - 16 x 4 warps x 32 t/w = 2048 mul-add ALUs



GTX 980 (2014) -> H100 (2022)

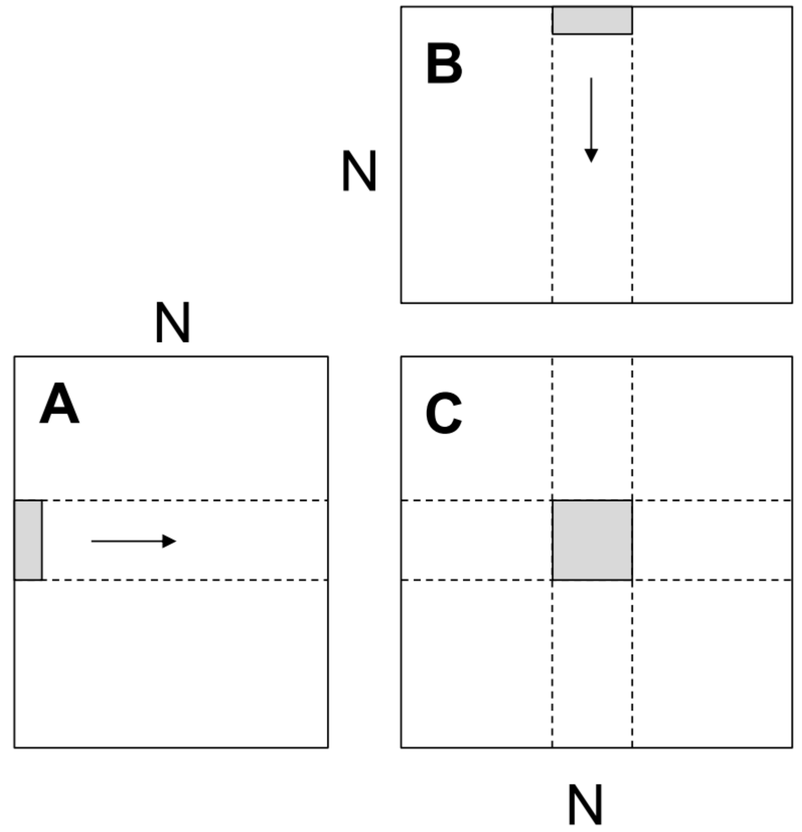
- SMMs remain the same
  - Clock speed: 1064 MHz -> 1110 MHz
  - Map warps per SMM: 64 -> 64
  - Threads per warp: 32 -> 32
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)
- #SMs: 16 SMMs -> 132 SMMs
- Flops: 4.6 TFLOPs -> 1000 TFLOPs (mainly because of tensor core)
  - Q: what is tensorcore – how does it work?

Case study: GPU Matmul

- Starwman solution:
  - C = A x B
  - Each thread computes one element

```
int N = 1024;
dim3 threadsPerBlock(32, 32, 1);
dim3 numBlocks(N/32, N/32, 1);

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

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  result = 0;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```

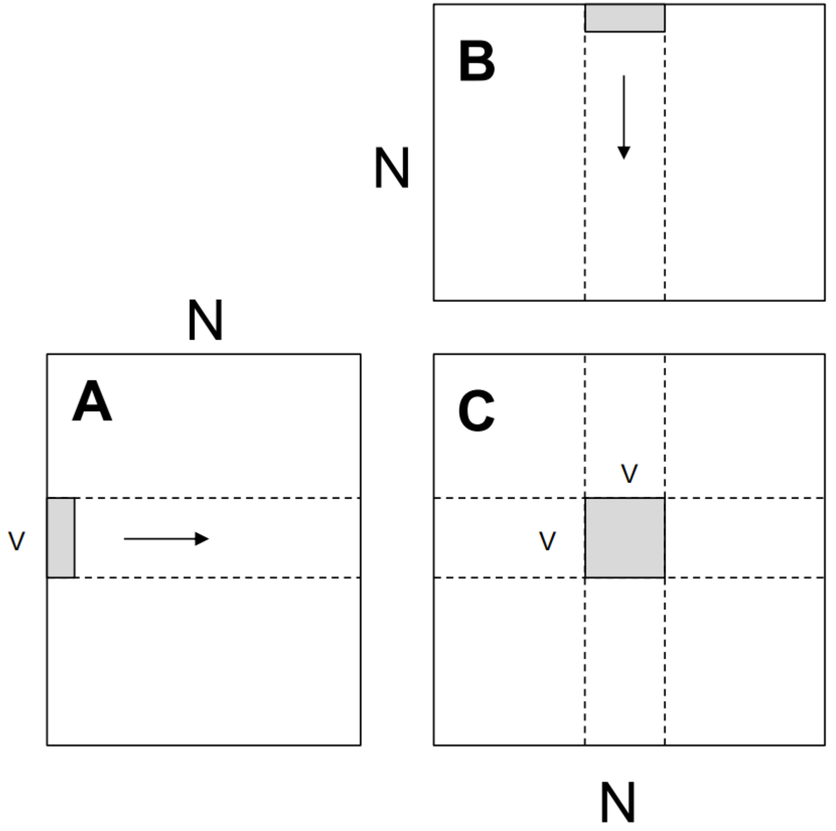


Global memory access per thread: 2*N
Number of threads: N^2
Total global memory access: $2N^3$

Recall register tiling

- Each thread computes a VxV submatrix

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  float c[V][V] = {0};
  float a[V], b[V];
  for (int k = 0; k < N; ++k) {
    a[:] = A[xbase*V : xbase*V + V, k];
    b[:] = B[k, ybase*V : ybase*V + V];
    for (int y = 0; y < V; ++y) {
      for (int x = 0; x < V; ++x) {
        c[x][y] += a[x] * b[y];
      }
    }
  }
  C[xbase * V : xbase*V + V, ybase*V : ybase*V + V] = c[:];
}
```



Global memory access per thread: $2NV$
Number of threads: N^2/V^2
Total global memory access: $2N^3/V$

To be continued

- Q: apparently we do another layer of tiling at block level.
- Think About How to do it?