https://haoailab.com/cse291-s26/

CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

Our First CUDA Program

```
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];
}
```

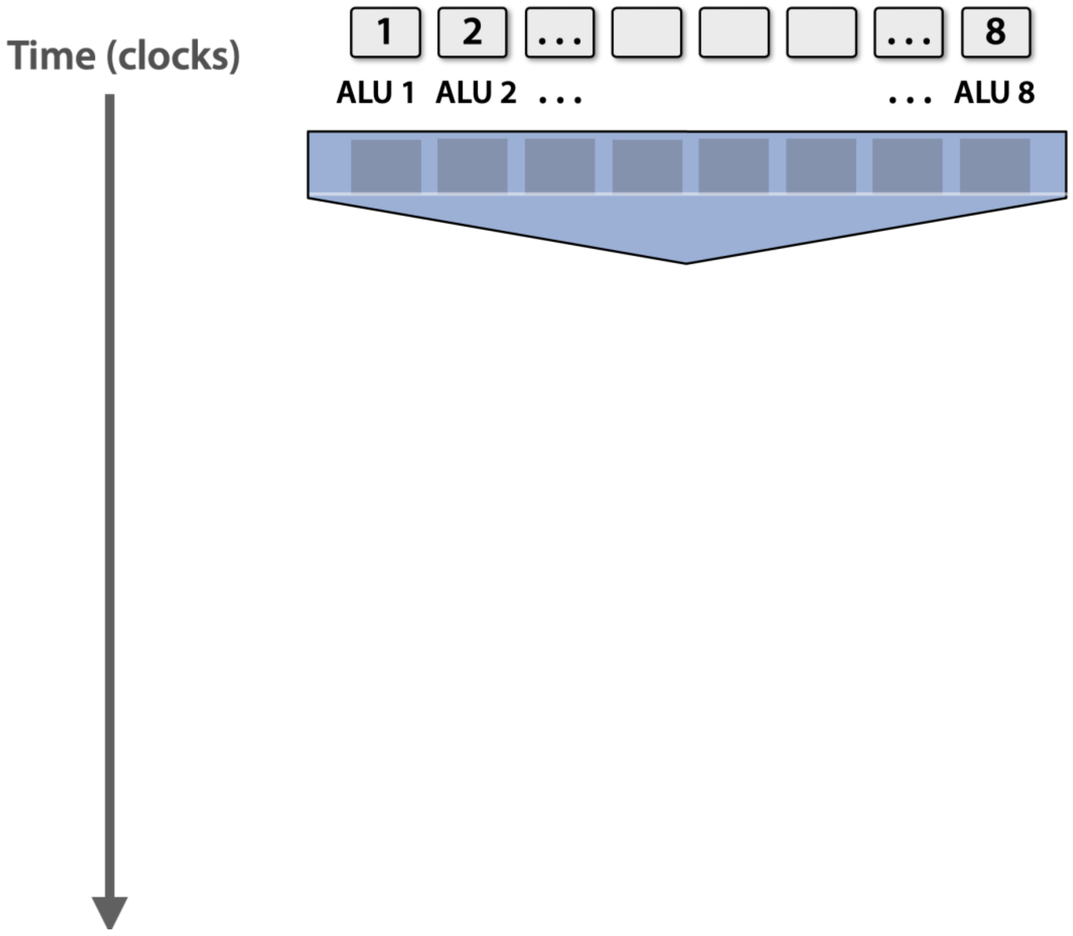
SIMD Constraints: how to handle control flow?

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

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

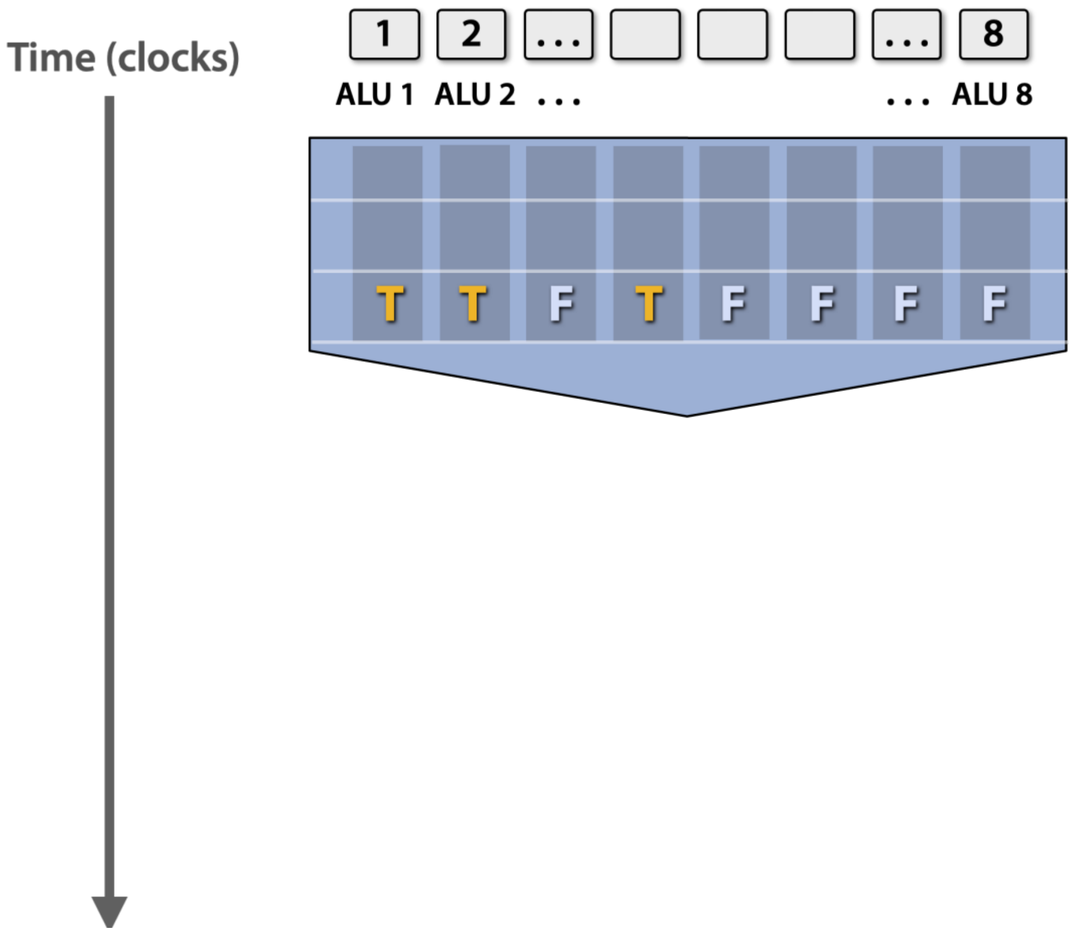
```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



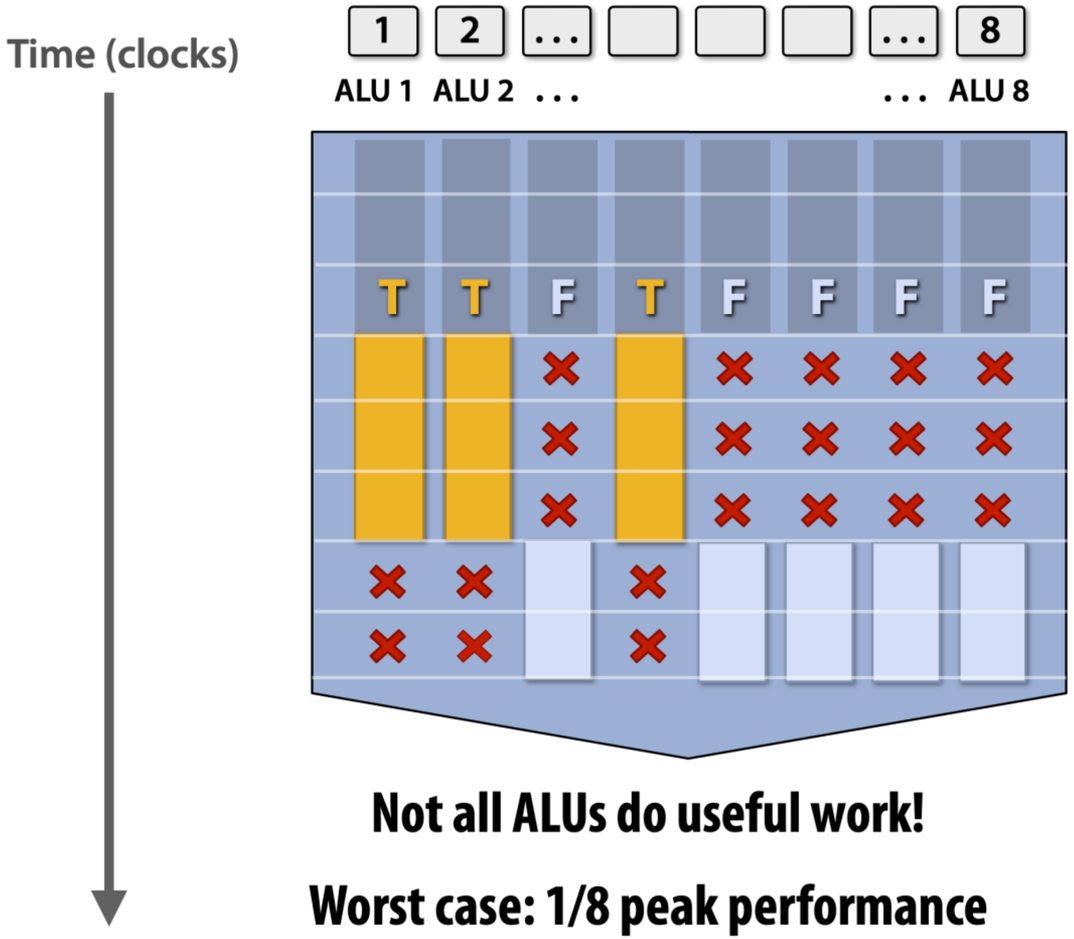

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Handling Control Flow



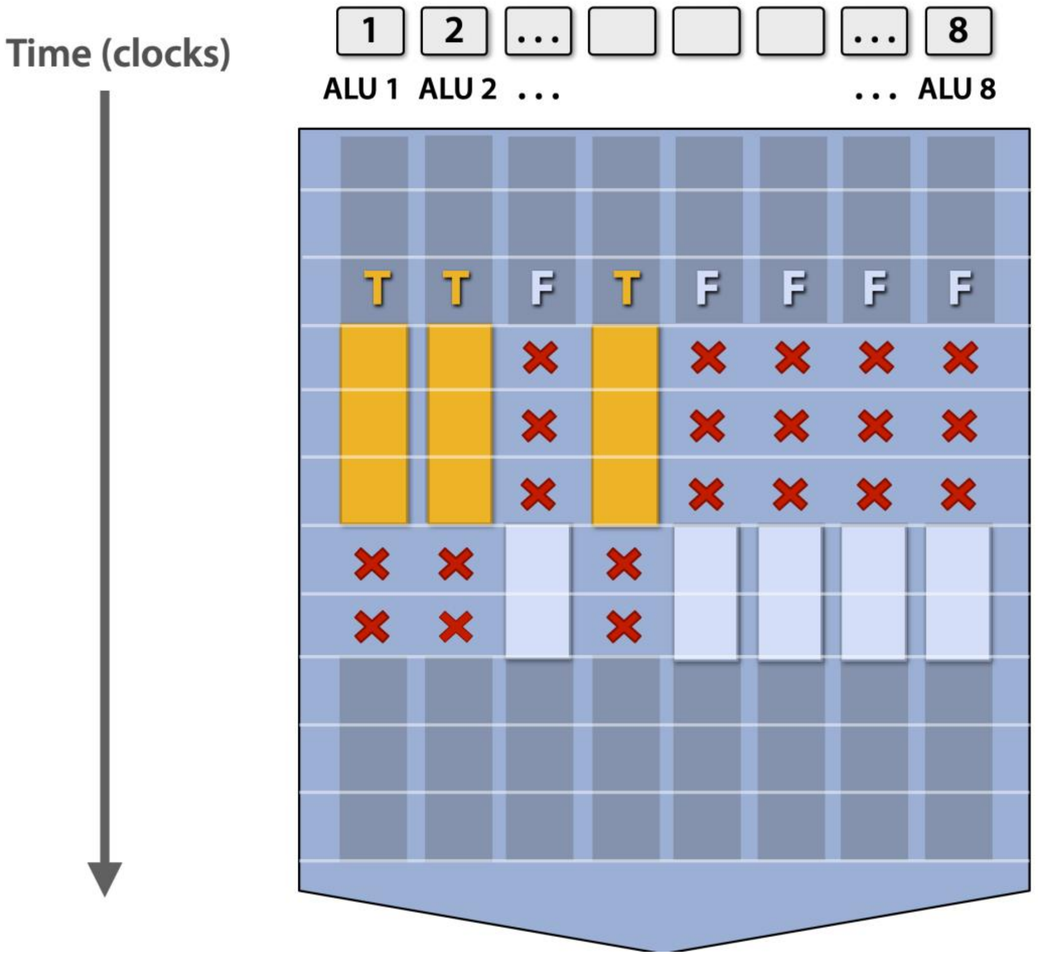

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

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;
}
```

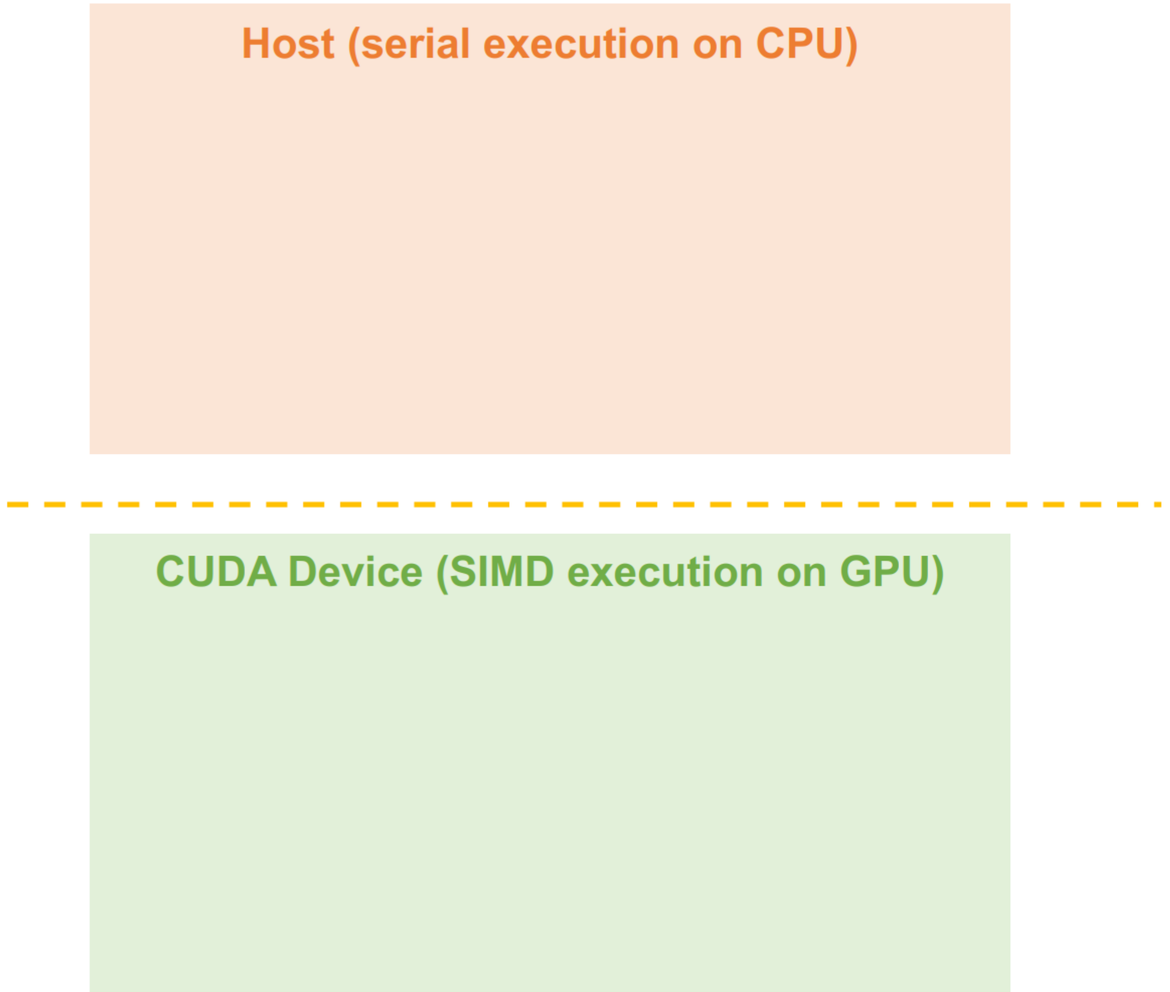
Coherent vs. Divergent

- 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

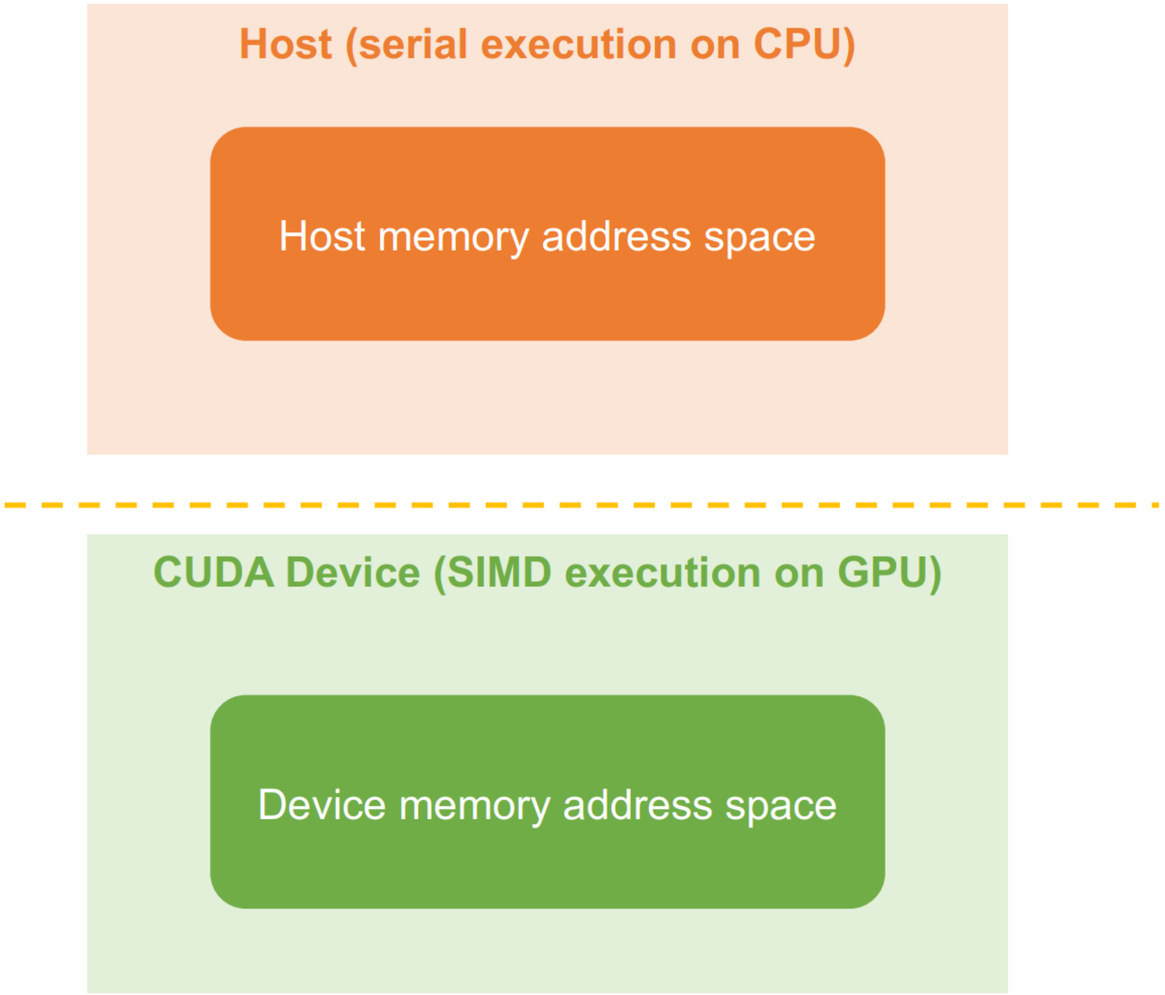
GPU and CUDA

- 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

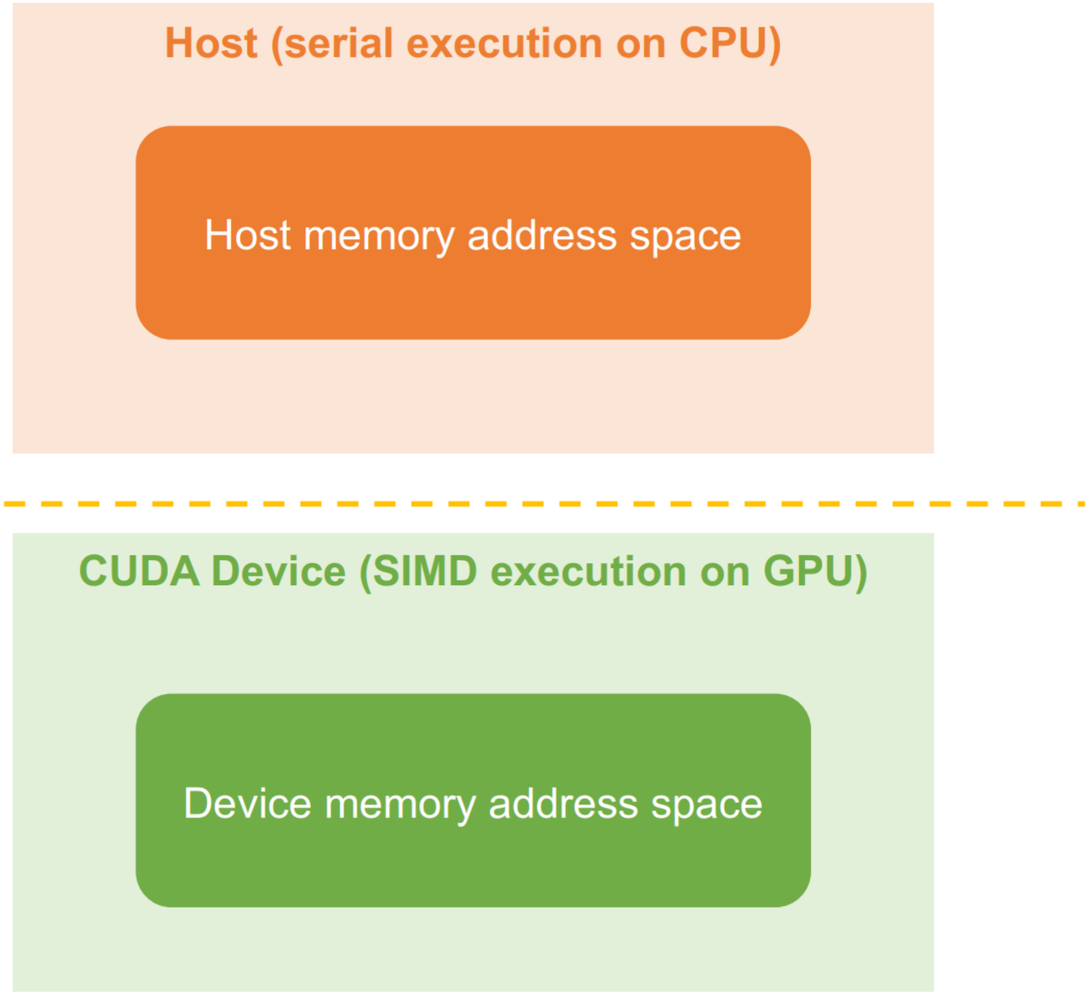
Recap:

- 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

If to populate content: cudaMemcpy



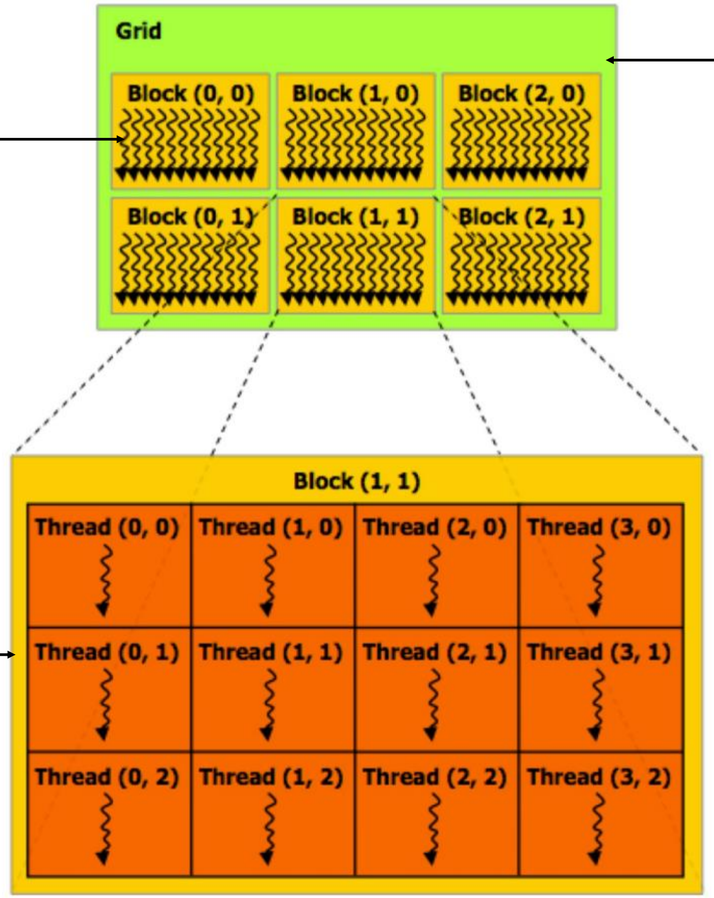

```
float* A = new float[N];

// 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)



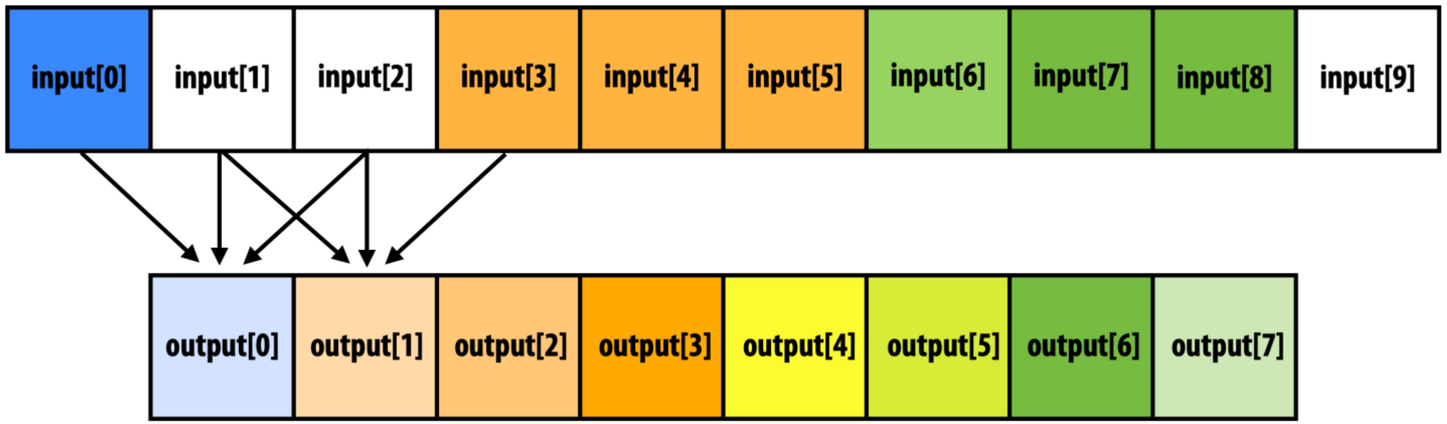
Global device memory
(r/w by all threads)

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

Why make it so complex:

- Balance between speed and shared-memory parallelism

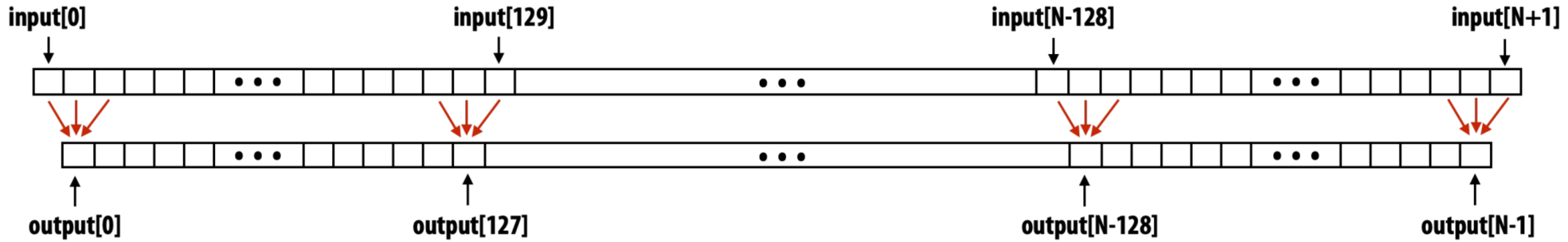
Example Program: Window Average




```
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

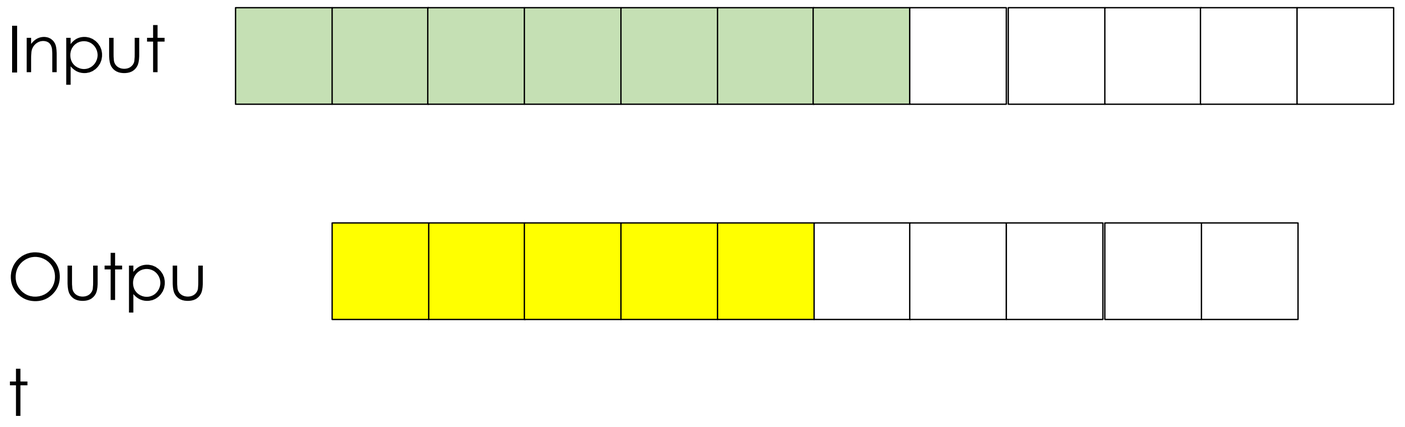
```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// 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:

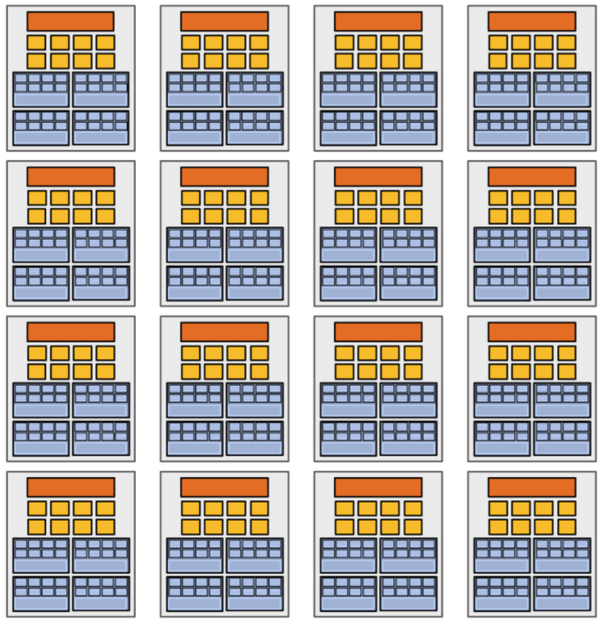
Program text (instructions)
Information about required resources:

- 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

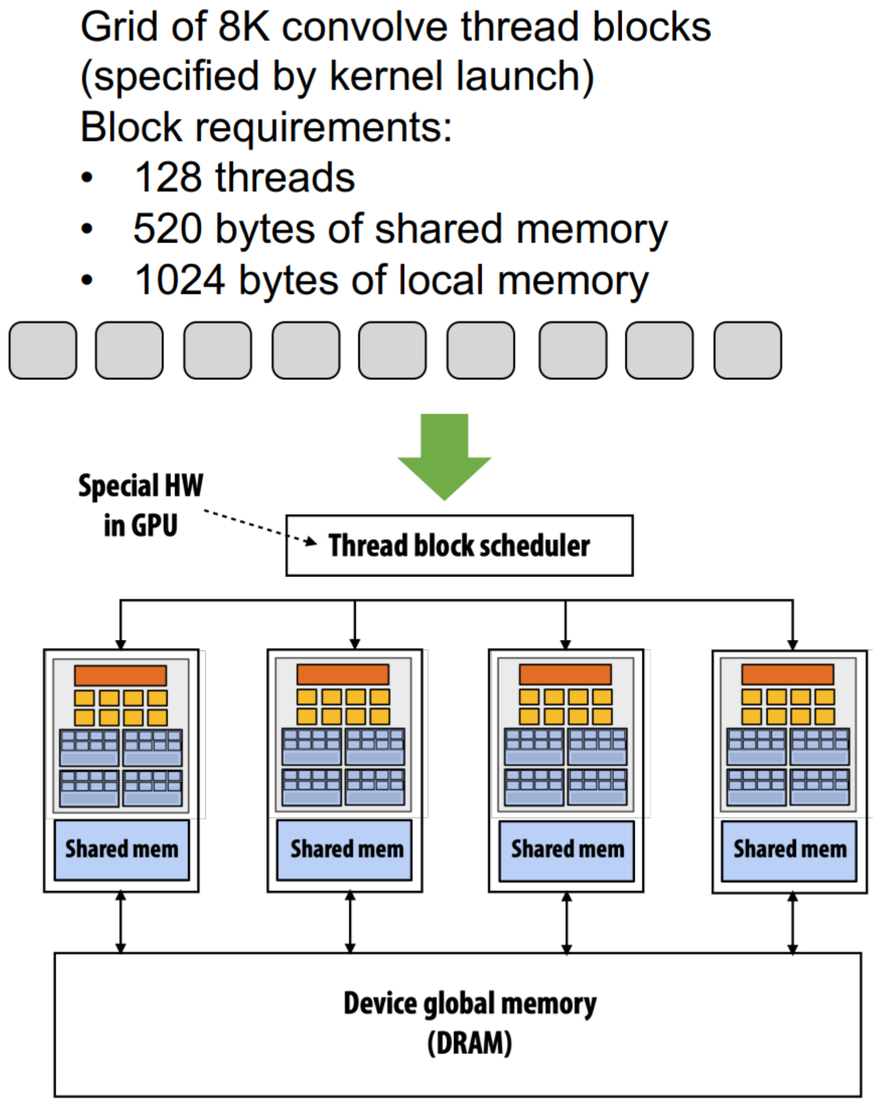
Mid-range GPU (6 cores)



High-end GPU (16 cores)

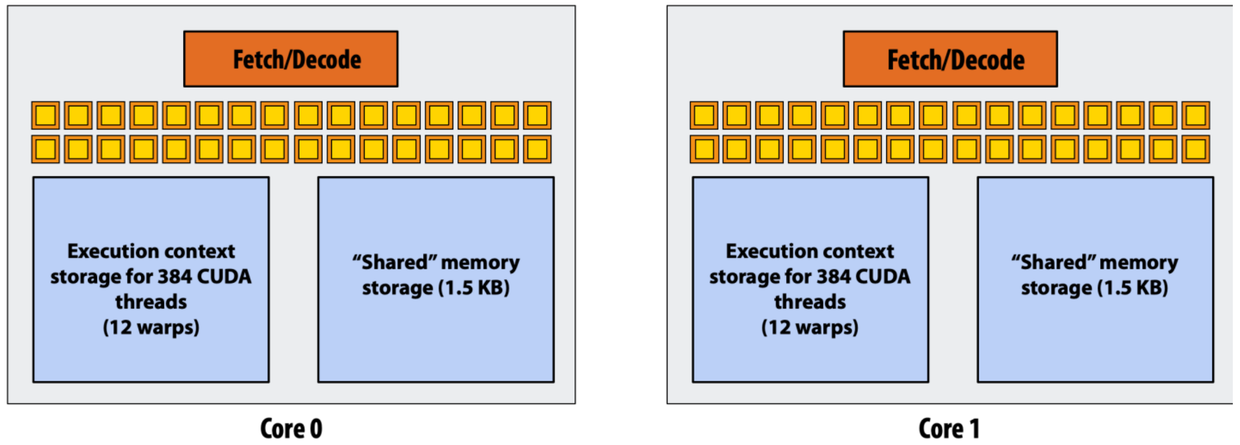
Scheduling on CUDA

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



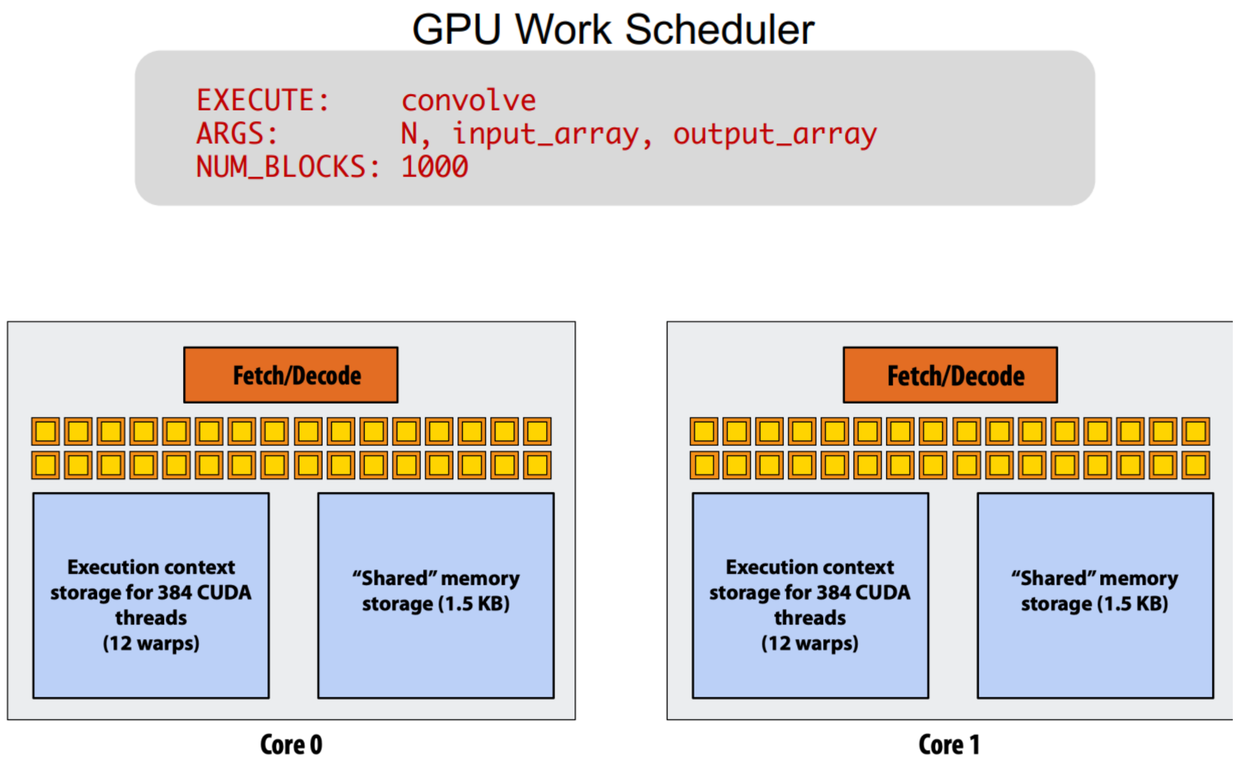
Deep Dive into CUDA scheduling

- 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?



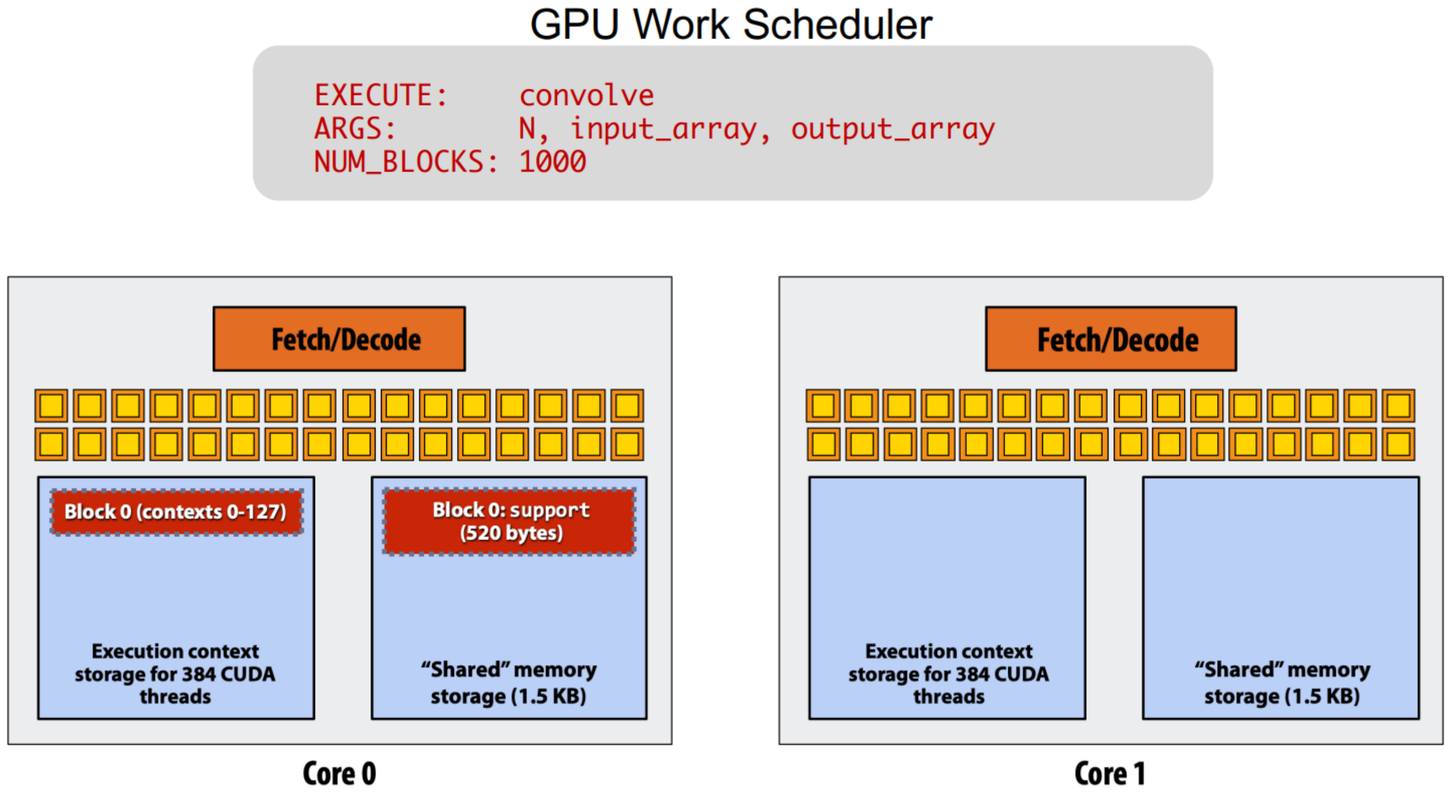
Deep Dive into CUDA scheduling

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



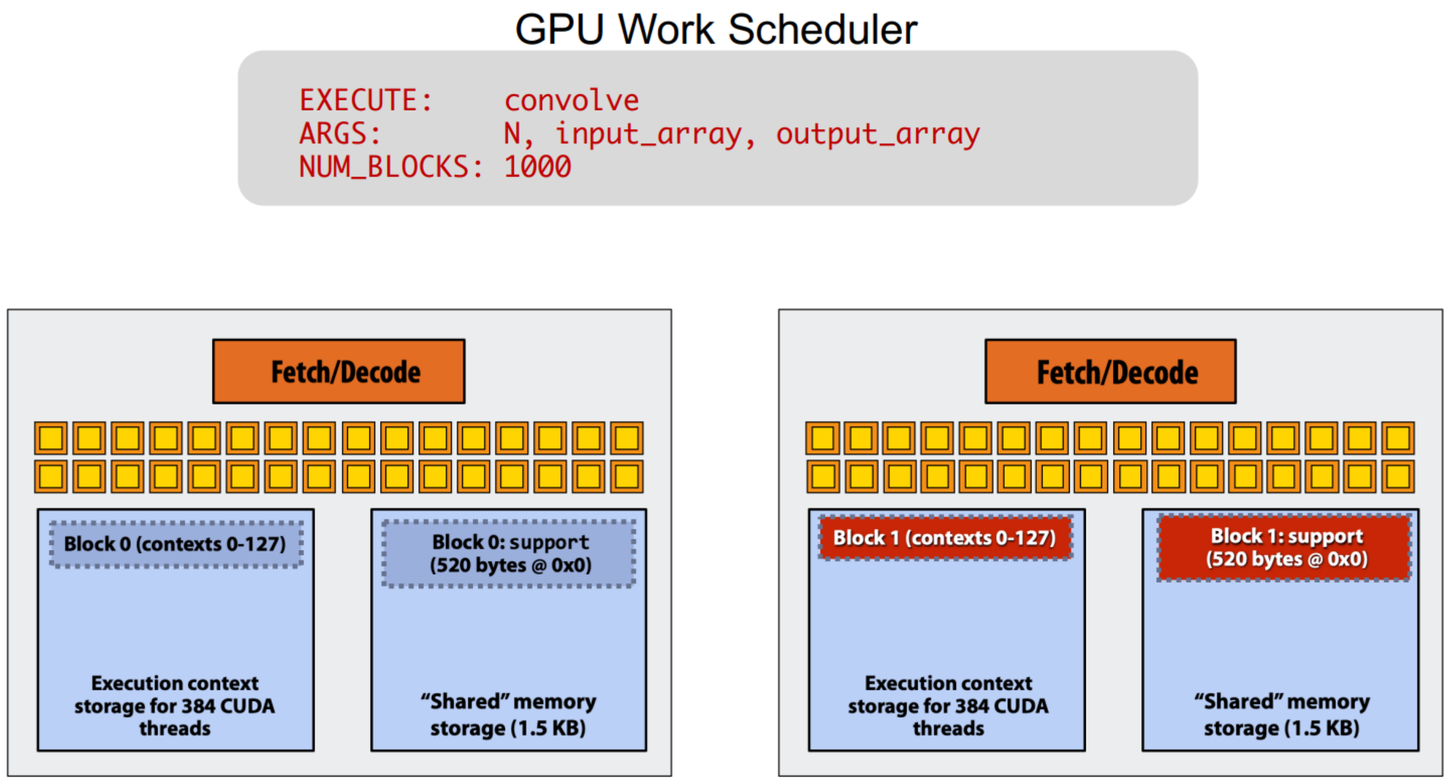
Deep Dive into CUDA scheduling

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



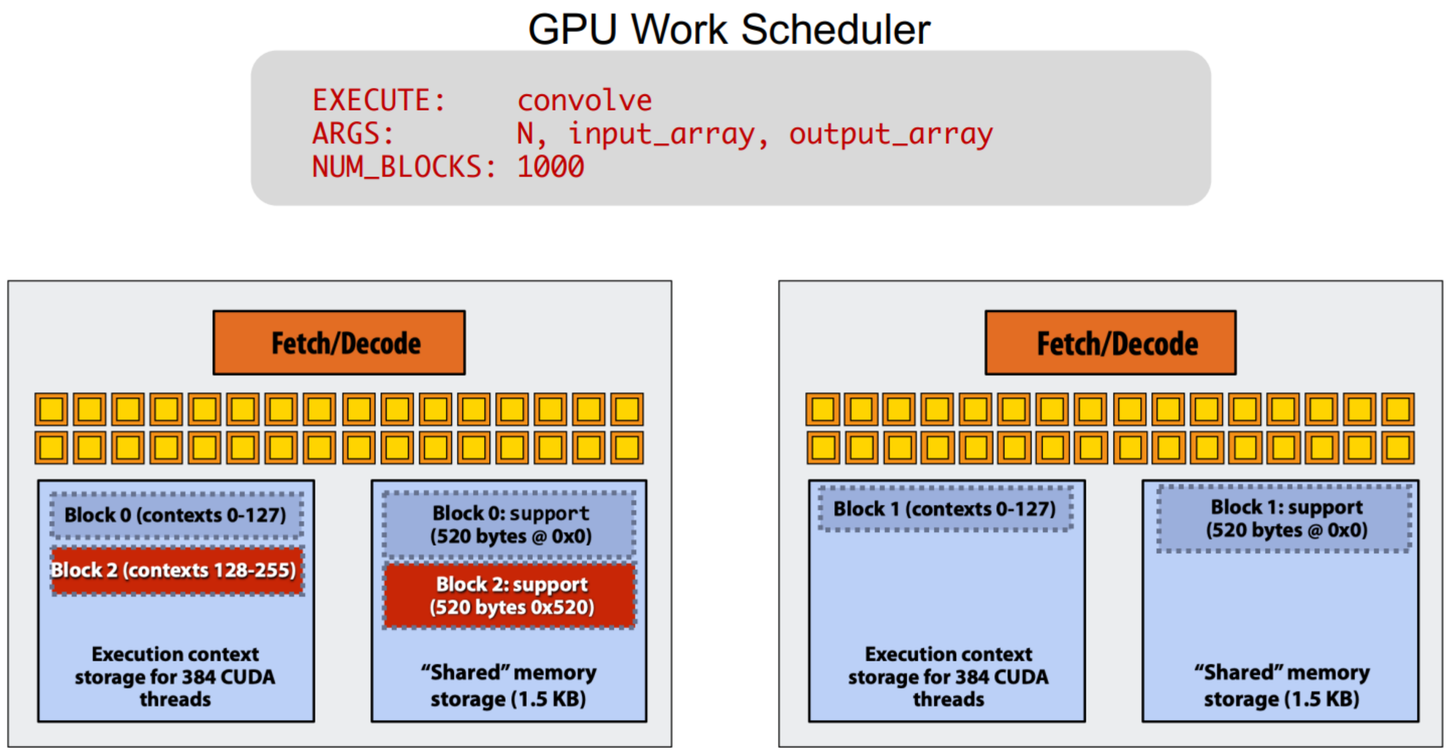
Deep Dive into CUDA scheduling

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



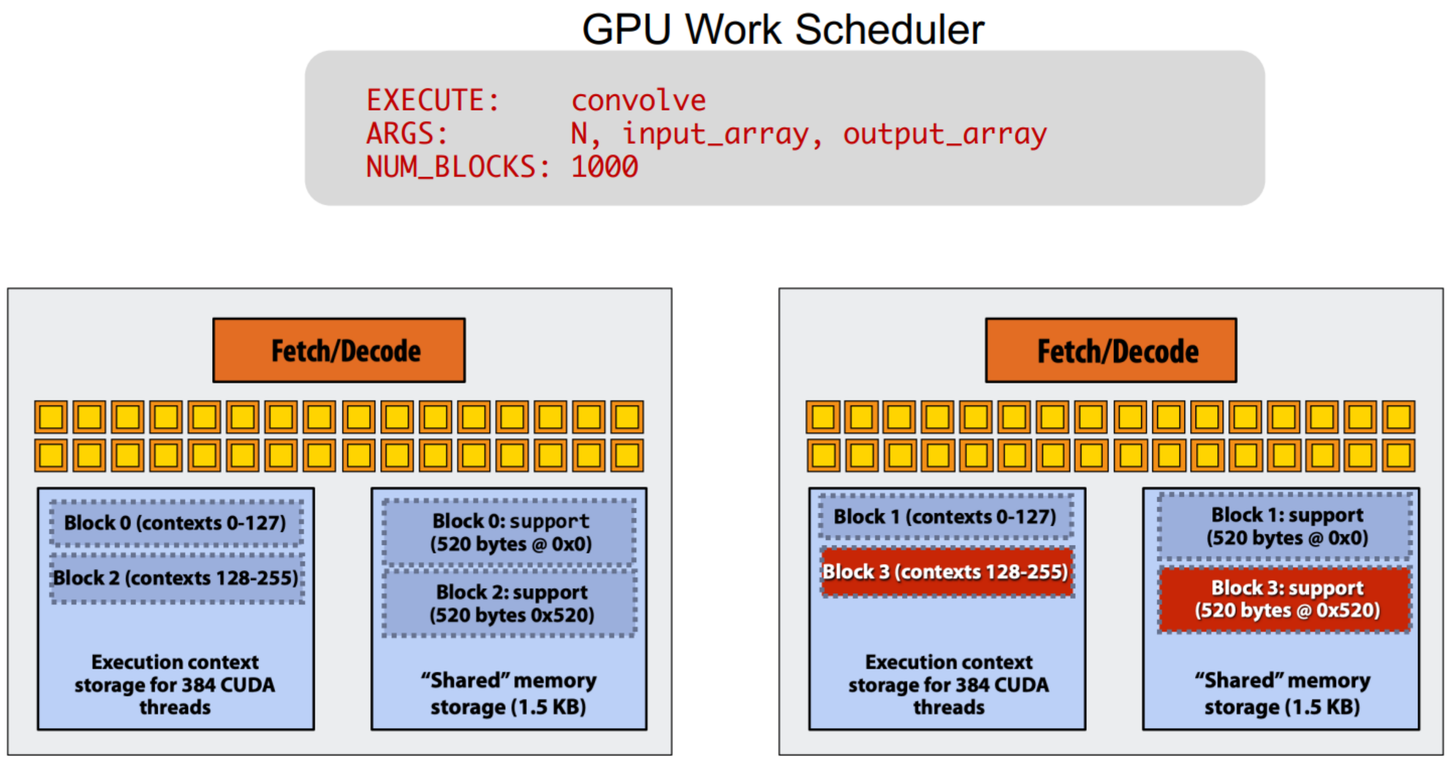
Deep Dive into CUDA scheduling

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



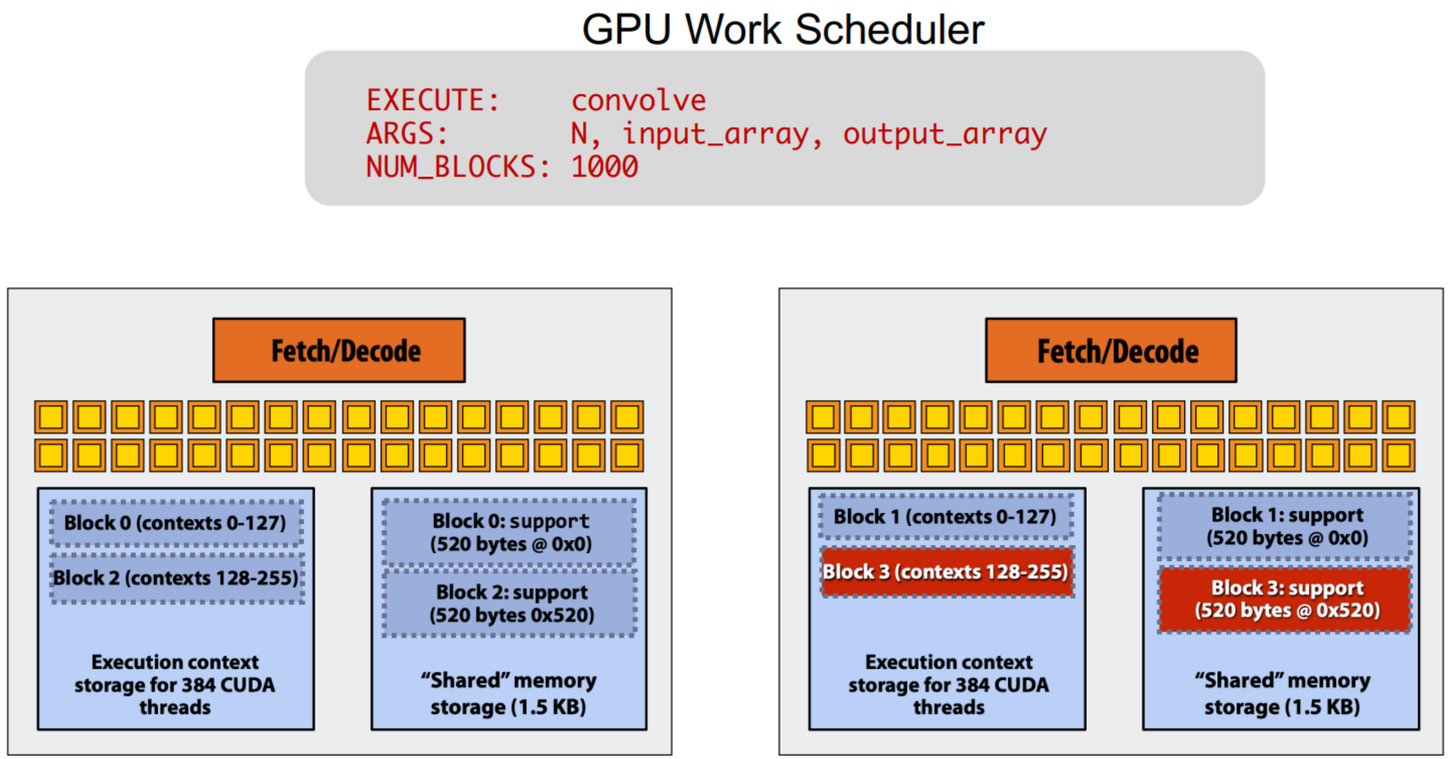
Deep Dive into CUDA scheduling

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



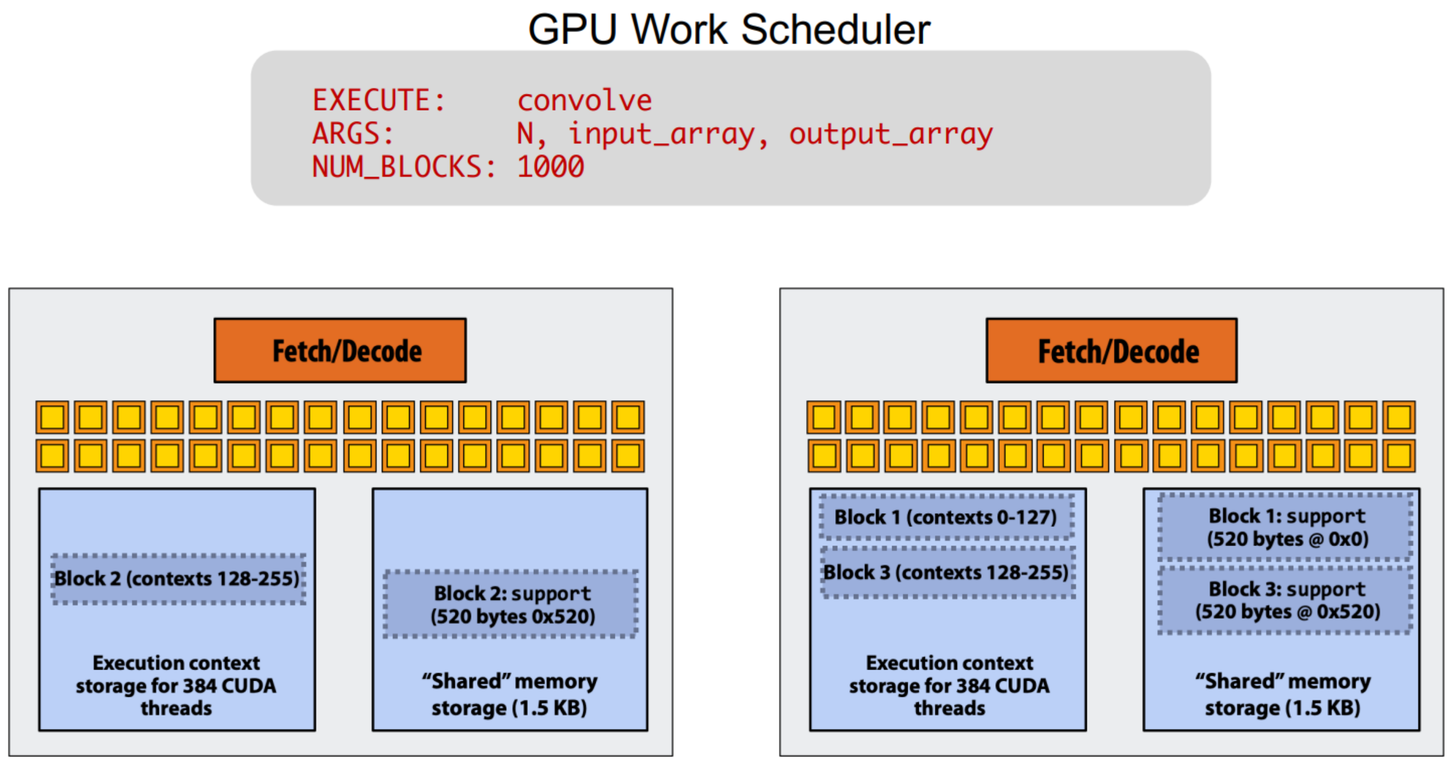
Deep Dive into CUDA scheduling

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



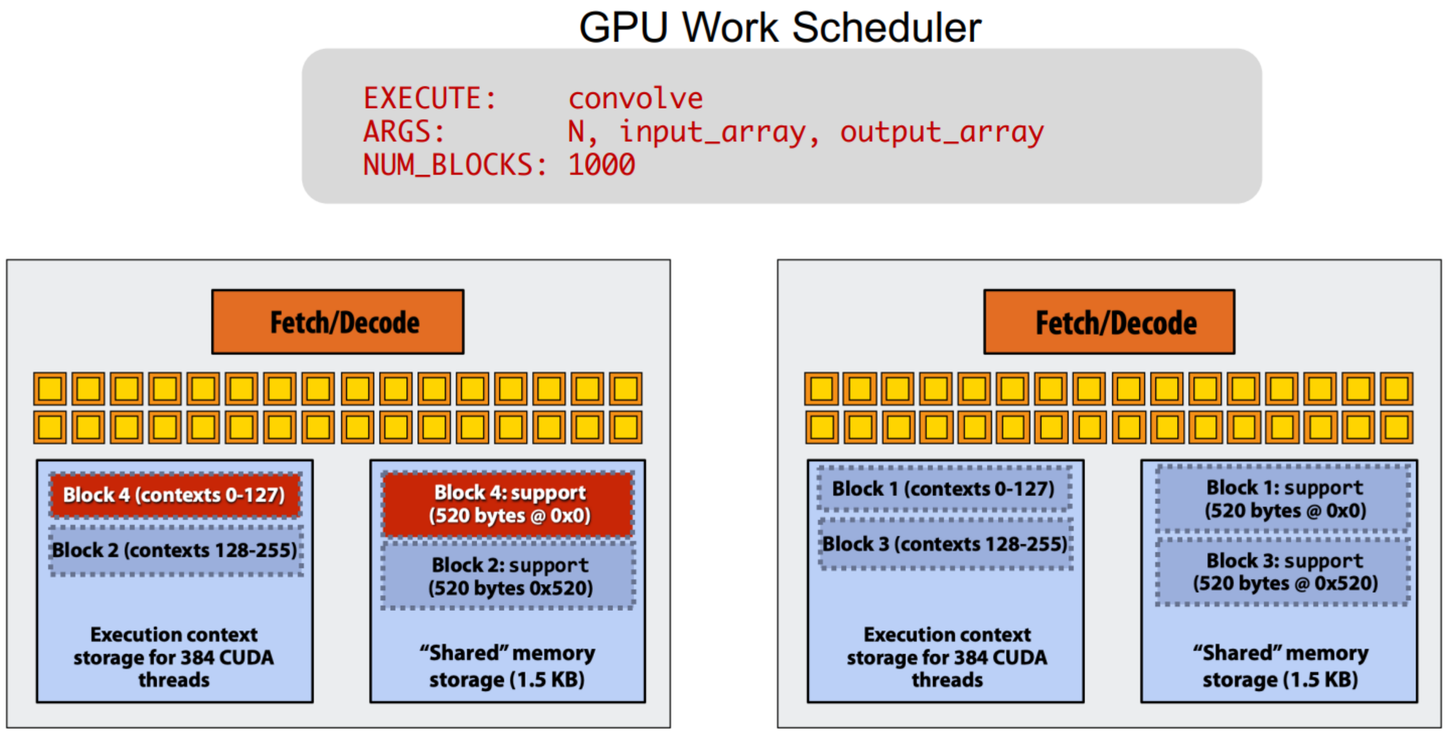
Deep Dive into CUDA scheduling

- Step 4: thread block 0 completes on SM 0



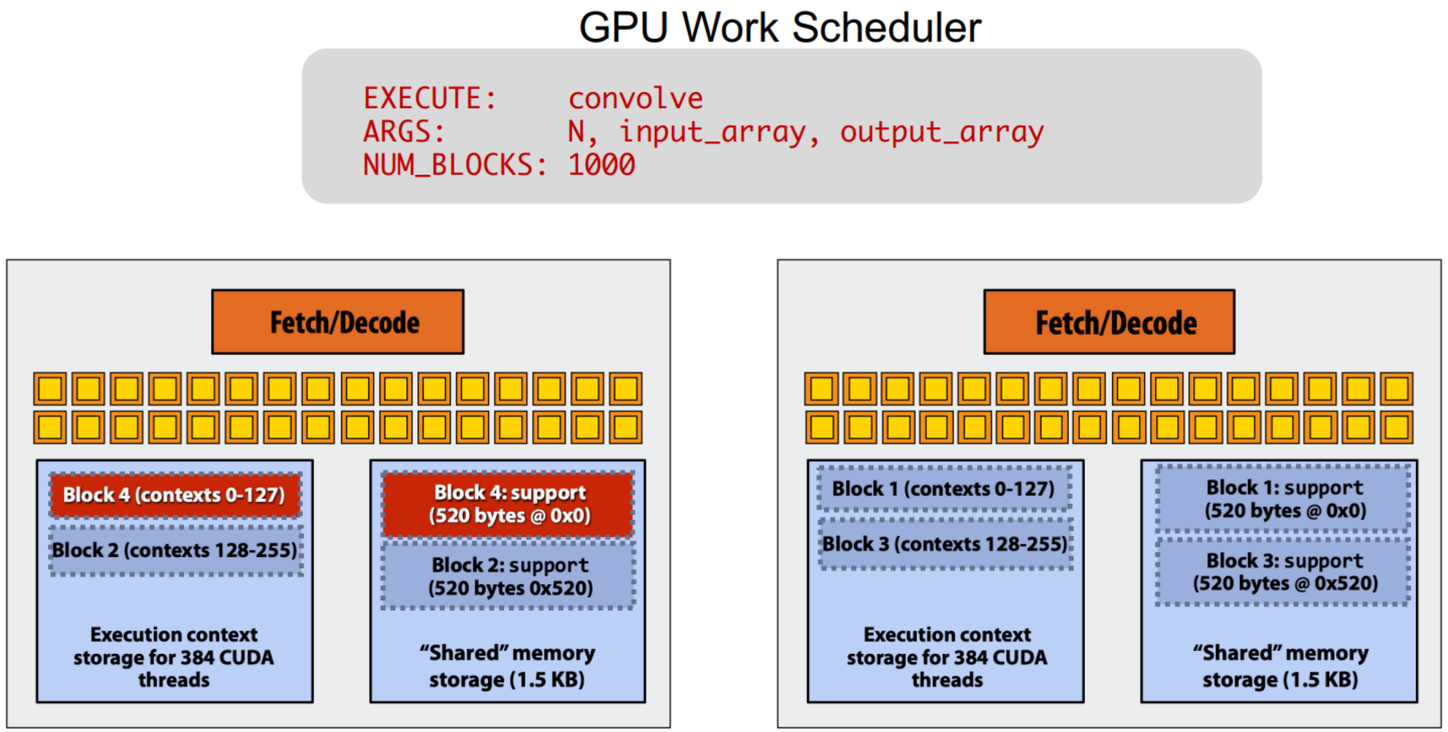
Deep Dive into CUDA scheduling

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



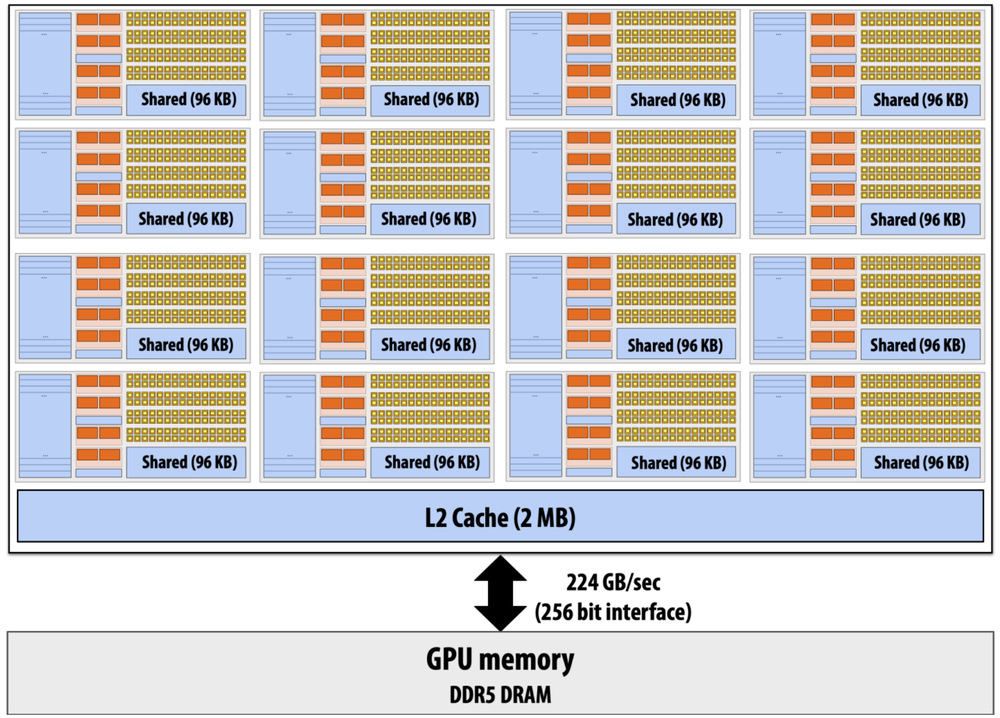
Deep Dive into CUDA scheduling

- 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 largely remain the same
  - Threads per block: 2048 -> 2048
  - CUDA cores: 128 -> 128
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)
- But #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

So Far So Good?

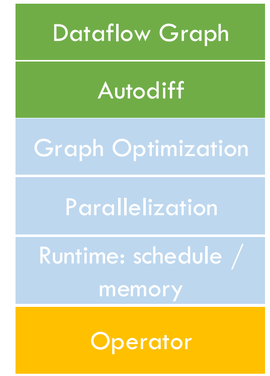
- Basic concepts in GPUs
- Execution Model
  - Launch kernel code to grids with many threadblocks
- Memory hierarchy
  - Shared memory – SRAM
- Two example code: matrix-add and conv1d

So Far So Good?

- GPU Execution Model: thread hierarchy
  - Bulk launch of many threads
  - Two-level hierarchy: threads are grouped into thread blocks
- Distributed address space
  - Built-in memcpy primitives to copy between host and device address spaces (cudamalloc, cudamemcpy, pinned memory)
- Three different types of device address spaces
  - Per thread, per block (“shared”, SRAM), or per device (“global”, HBM)
- Barrier synchronization primitive for threads in thread block and cpu <-> gpu
- First GPU program: window average (== conv1d)

Next

- **Case study: Matmul on GPU**



Develop the Thought Process when CUDA-ing

Convert your brain to be SIMD:

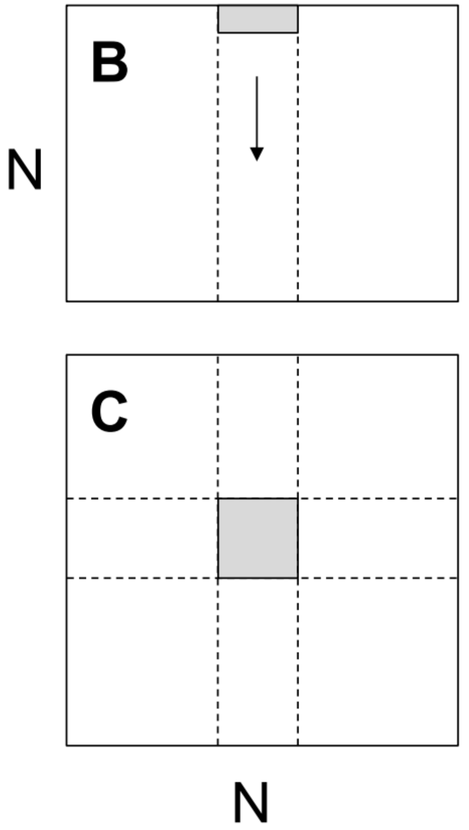
1. Identify work that can be performed in parallel
2. Partition work (and data associated with the work)
3. Manage data access, communication, and synchronization

And make sure

1. Oversubscription: create enough tasks to keep all execution units on a machine busy
2. Mitigate straggler: Balance workload (because GPU cores does not know control flow)
3. Minimize “communication”: reduce I/O across memory hierarchies

Case study: GPU Matmul v1

- C = A x B
- Q: what's the work that can be parallelizec
- 💡Each thread computes one element!

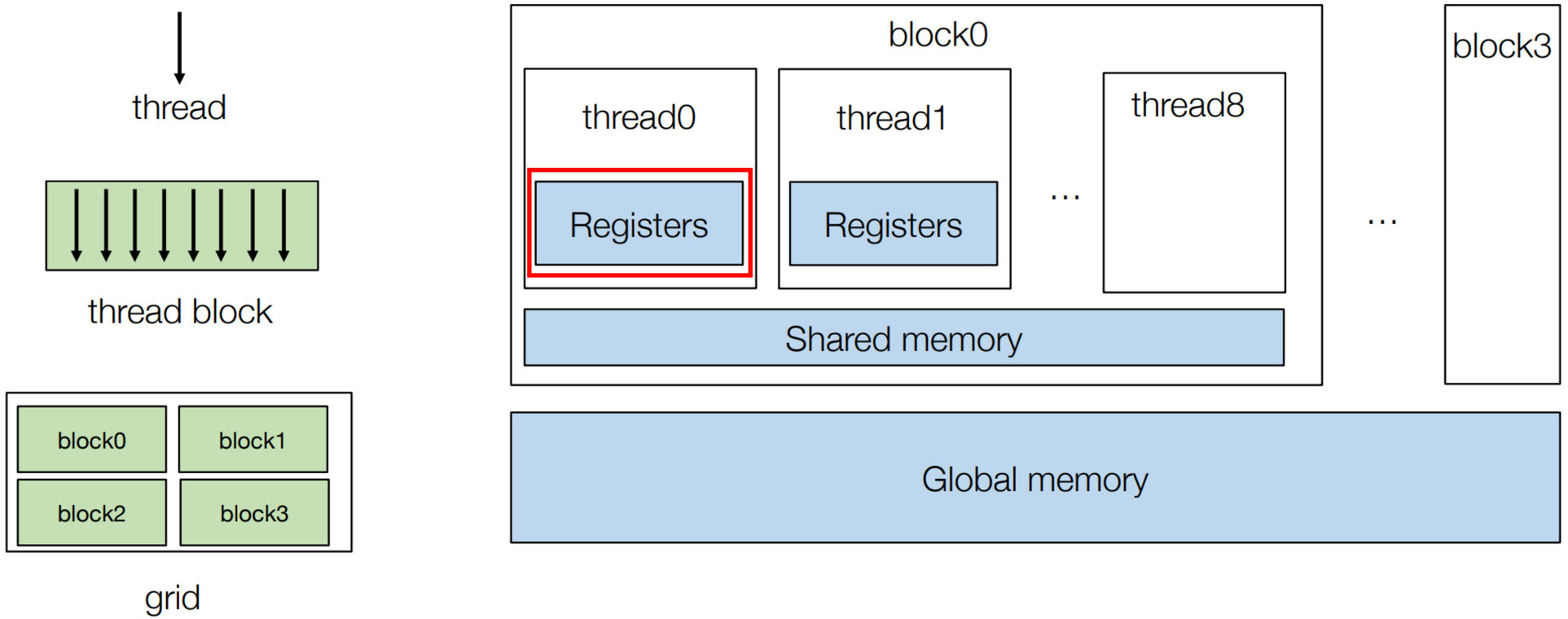



```
int N = 1024;
dim3 threadsPerBlock(32, 32, 1);
dim3 numBlocks(N/32, N/32, 1);

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 read per thread?
  - N + N = 2N
- # threads?
  - N^2
- Total global memory access?
  - N^2 * 2N = 2N^3
- Memory?
  - 1 float per thread

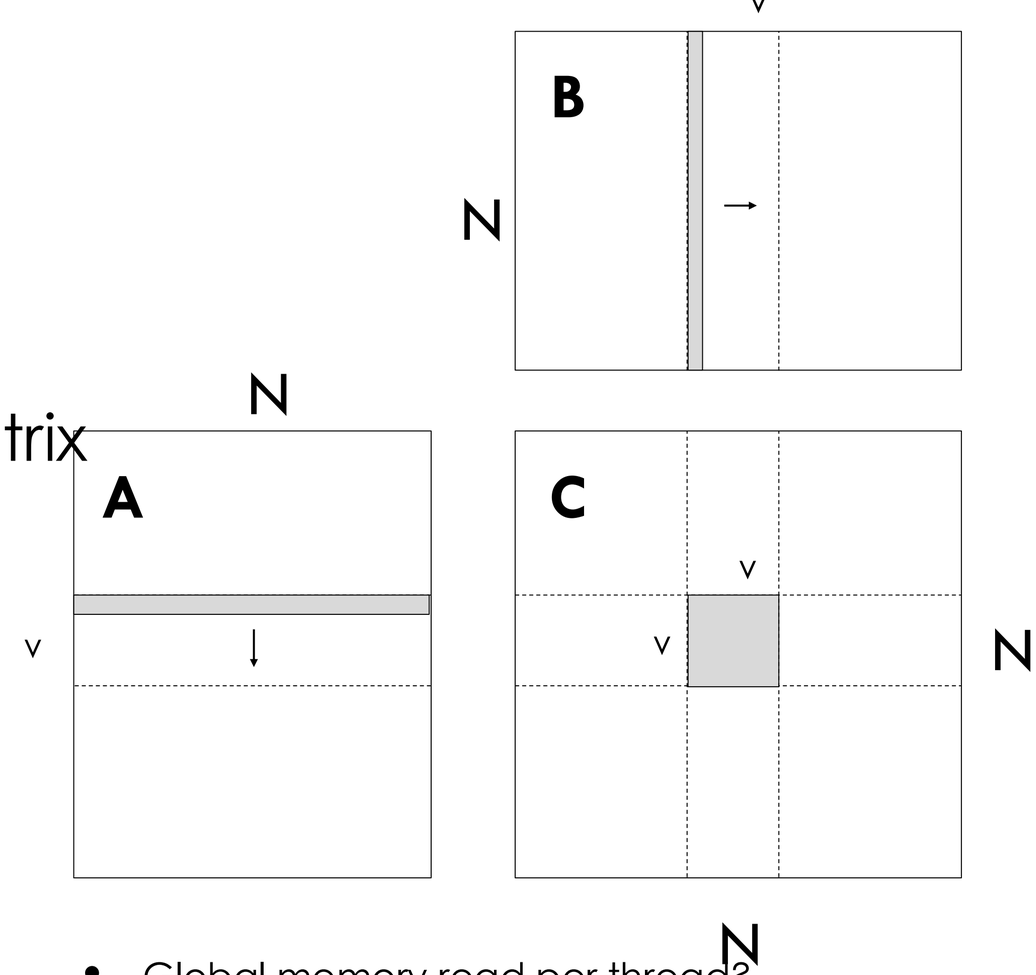
Recall Memory Hierarchy and Register tiling



💡Each thread uses more thread-level registers to compute outputs to save I/o

GPU Matmul v1.5: Thread Tiling

- Each thread computes a VxV submatrix

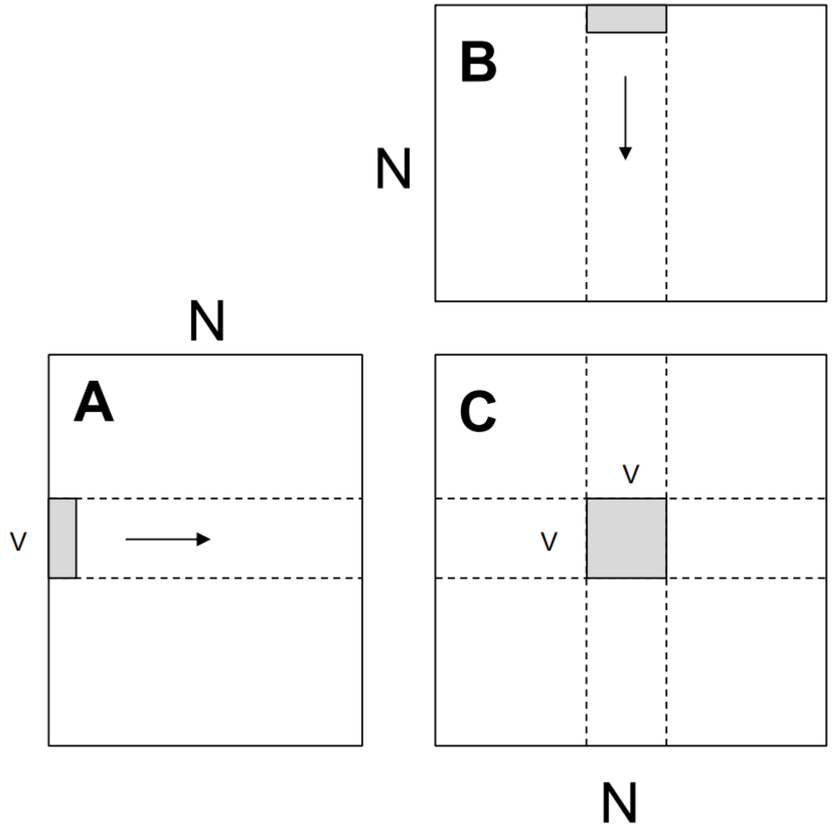



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

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

- Global memory read per thread?
  - NV + NV^2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * (NV + NV^2) = N^3/V + N^3
- Memory?
  - V^2 + 2N float per thread

GPU Matmul v2: Can we do better?

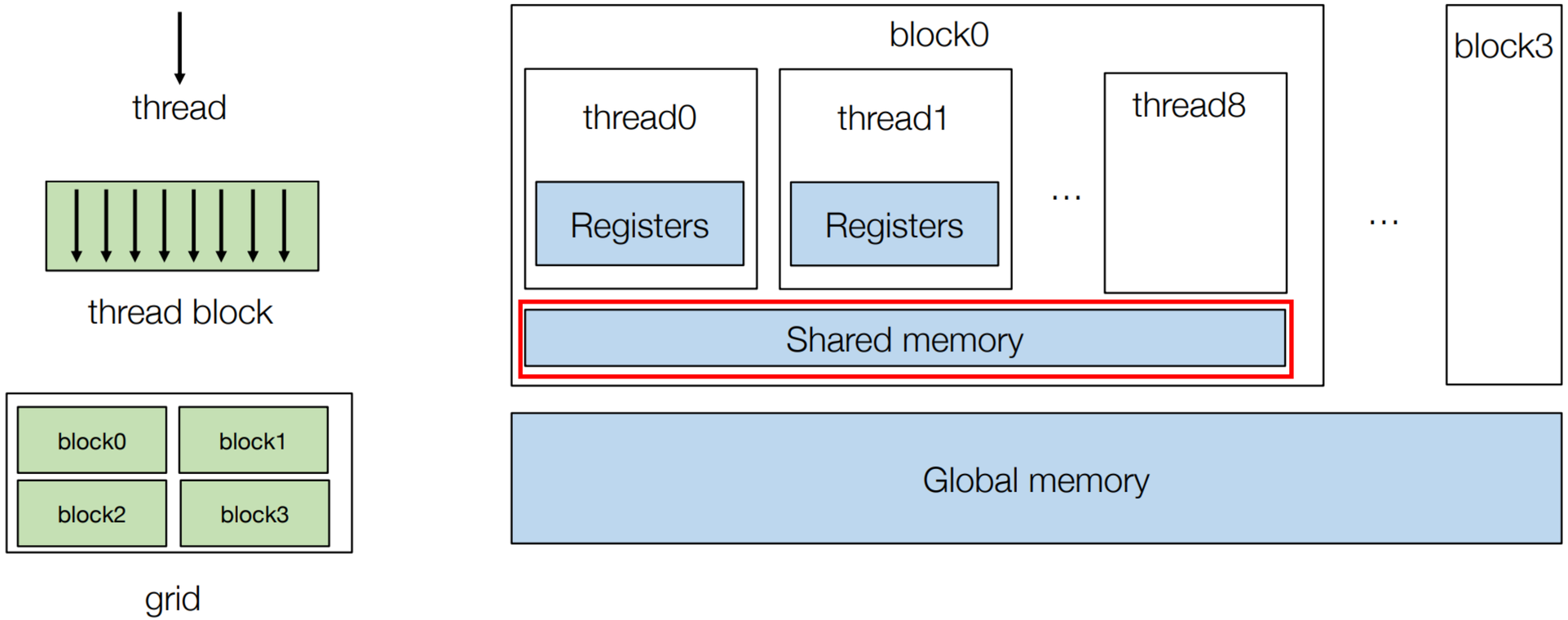
- Each thread computes a VxV submatrix
- 💡compute partial sum: $[X_1, X_2]\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = X_1Y_1 + X_2Y_2$




```
__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 read per thread?
  - NV * 2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * 2NV = 2N^3/V
- Memory?
  - V^2 + 2V float per thread

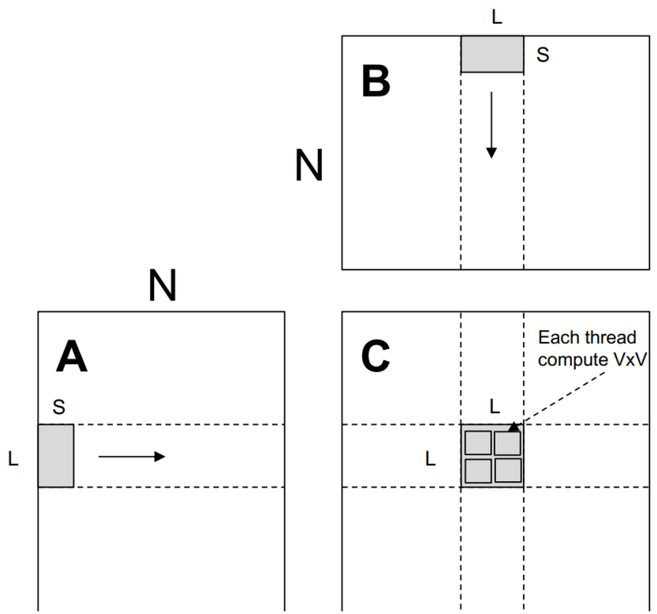
Recall Memory Hierarchy and Cache tiling



💡Try to utilize block-level shared memory (SRAM)

GPU Matmul v3: SRAM Tiling (GPU)

- Use block shared mem
- A block computes a L x L submatrix
- Then a thread computes a V x V submatrix and reuses the matrices in shared block memory




```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[y][x] += a[y] * b[x];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Memory overhead?

- Global memory access per threadblock
  - 2LN
- Number of threadblocks:
  - N^2 / L^2
- Total global memory access:
  - 2N^3 / L
- Shared memory access per thread:
  - 2VN
- Number of threads
  - N^2 / V^2
- Total shared memory access:
  - 2N^3 / V

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[y][x] += a[y] * b[x];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Cooperative Fetching

```
sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
```

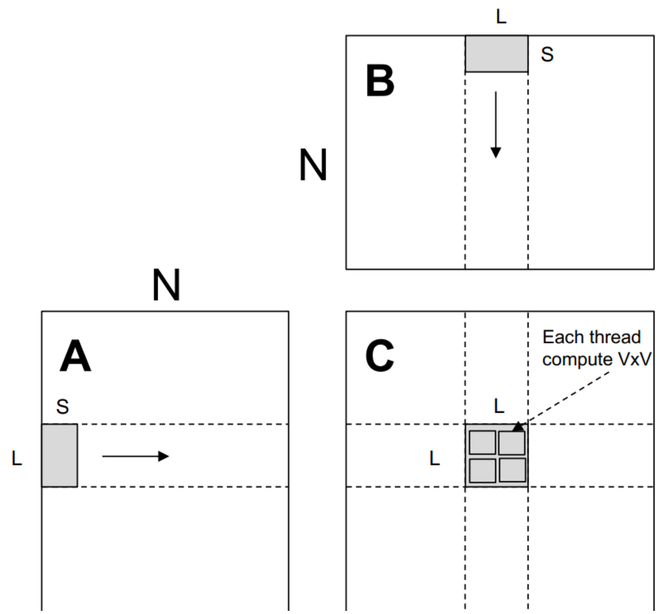
```
int nthreads = blockDim.y * blockDim.x;
int tid = threadIdx.y * blockDim.x + threadIdx.x;

for(int j = 0; j < L * S / nthreads; ++j) {
  int y = (j * nthreads + tid) / L;
  int x = (j * nthreads + tid) % L;
  s[y, x] = A[k + y, yblock * L + x];
}
```

Many More GPU Optimizations

- Global memory continuous read
- Shared memory bank conflict
- Pipelining
- Tensor core
- Lower precision

Core Problems Here

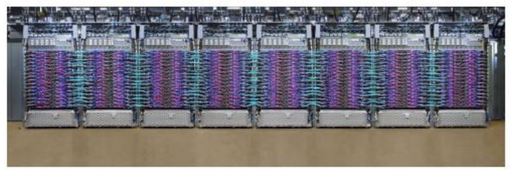
- How to choose L/V? Tradeoffs:
  - #threads
  - #registers
  - Amount of SRAM




```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[y][x] += a[y] * b[x];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

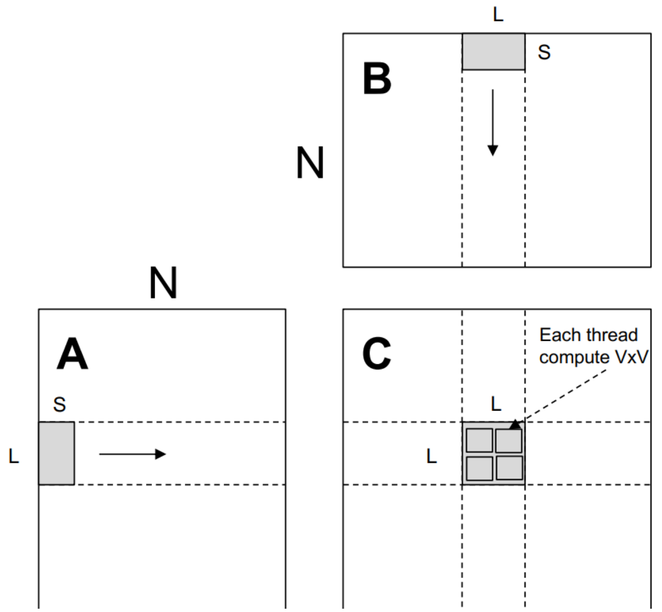
In Reality

MKL-DNN | cuDNN | ARM-Compute | TPU Backends



Back to Today's Problem

- How to implement an highly efficient kernel
- How to choose configs.
  - #threads
  - #registers
  - Amount of SRAM



- Solution 1:
  - expert-craft -> Enumerate configs -> profile
- Solution 2: Operator compilation