Mariia Mohylna

Non-portable kernel-based models

1

Key concepts

HIP vs CUDA
Hardware and grip
First program

2

Optimizations

Shared memory
Matrix multiplication
Parallel reduction

3

Honorable mentions

Memory types
Streams
…and more

HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs

HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs
➢ NVCC compiler	➢ HIPCC compiler

HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs
➢ NVCC compiler	➢ HIPCC compiler
➢ *.cuh or *.cu	➢ *.hip.hpp or *.hip.cpp

HIP vs CUDA

✓ Focus **solely** with and provide tools to **optimize** massive parallelization on GPUs

CUDA

- Supported on NVDIA GPUs
- NVCC compiler
- *.cuh or *.cu

compulsory

HIP

- Supported on AMD and (theoretically) NVIDIA GPUs
- HIPCC compiler
- *.hip.hpp or *.hip.cpp

convinient

HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs
➢ NVCC compiler	➢ HIPCC compiler
➢ *.cuh or *.cu	➢ *.hip.hpp or *.hip.cpp
➢ cudaMalloc, cudaMemcpy, cudaFree ….	➢ hipMalloc, hipMemcpy, hipFree ….

HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs
➢ NVCC compiler	➢ HIPCC compiler
➢ *.cuh or *.cu	➢ *.hip.hpp or *.hip.cpp
➢ cudaMalloc, cudaMemcpy, cudaFree ….	➢ hipMalloc, hipMemcpy, hipFree ….
➢ #include<cuda_runtime.h>	➢ #include<hip/hip_runtime.h>

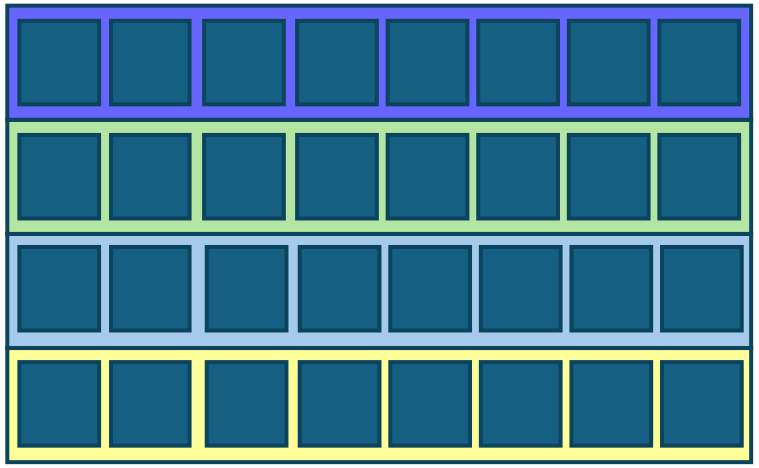
HIP vs CUDA

✓ Focus **solely** and provide tools to **optimize** massive parallelization on GPUs

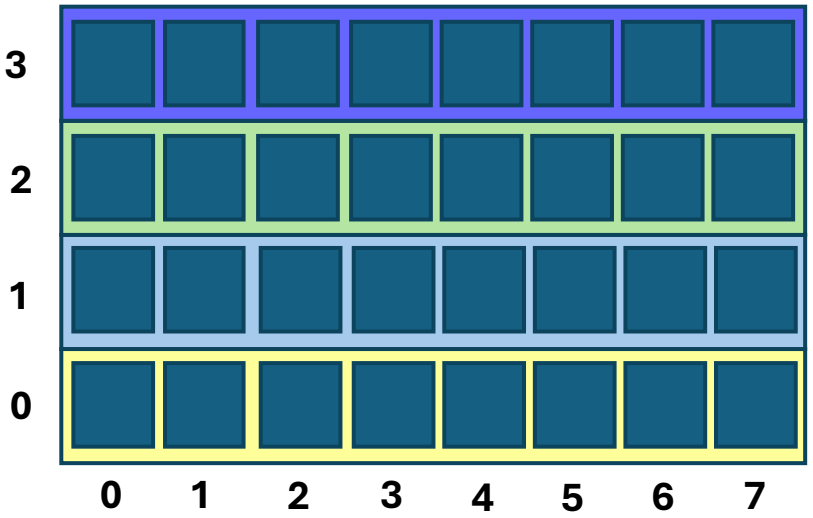
CUDA	HIP
➢ Supported on NVDIA GPUs	➢ Supported on AMD and (theoretically) NVIDIA GPUs
➢ NVCC compiler	➢ HIPCC compiler
➢ *.cuh or *.cu	➢ *.hip.hpp or *.hip.cpp
➢ cudaMalloc, cudaMemcpy, cudaFree ….	➢ hipMalloc, hipMemcpy, hipFree ….
➢ #include<cuda_runtime.h>	➢ #include<hip/hip_runtime.h>

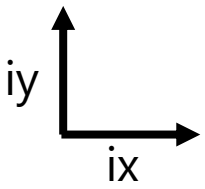
✓ Mostly have the same functionality, but better to make sure

Interlude: rank addressing

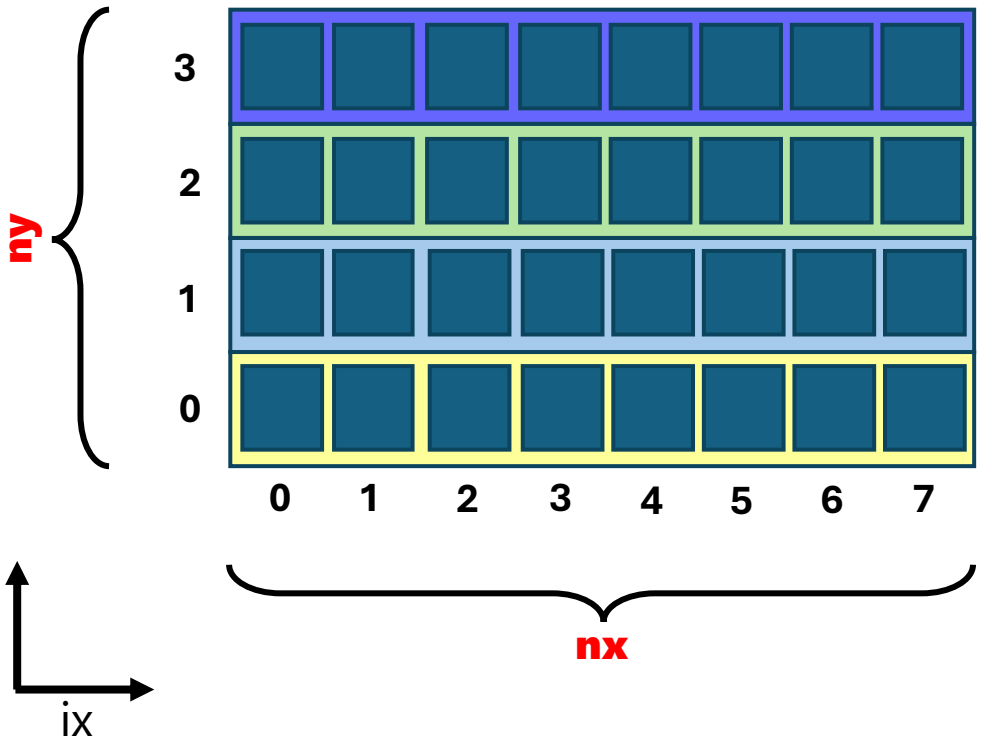


Interlude: rank addressing

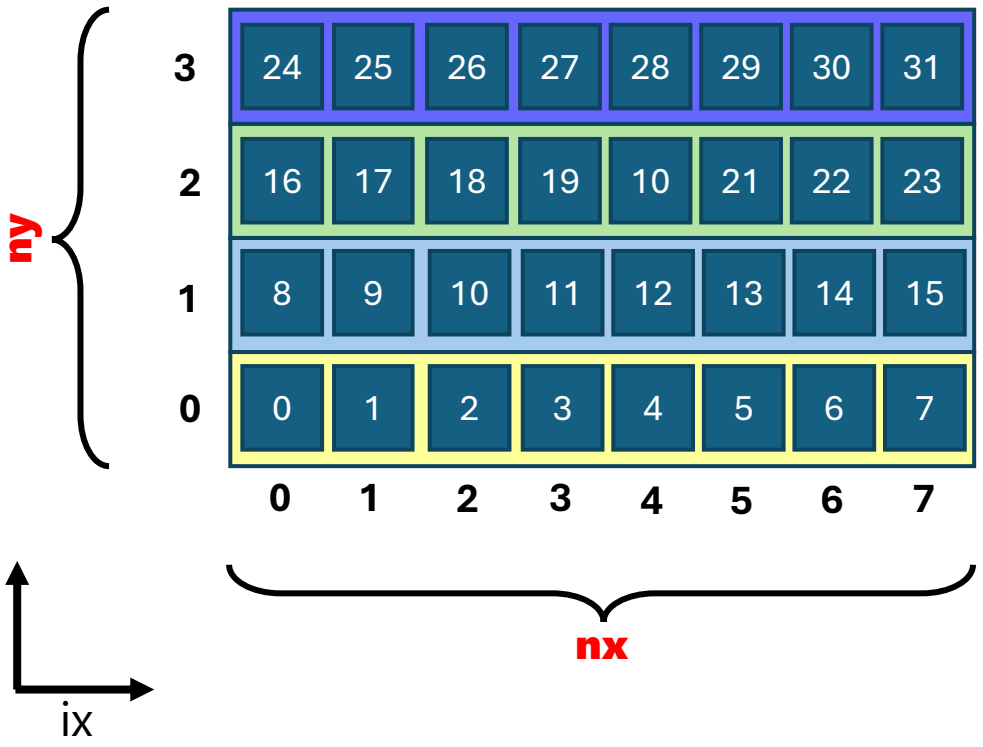




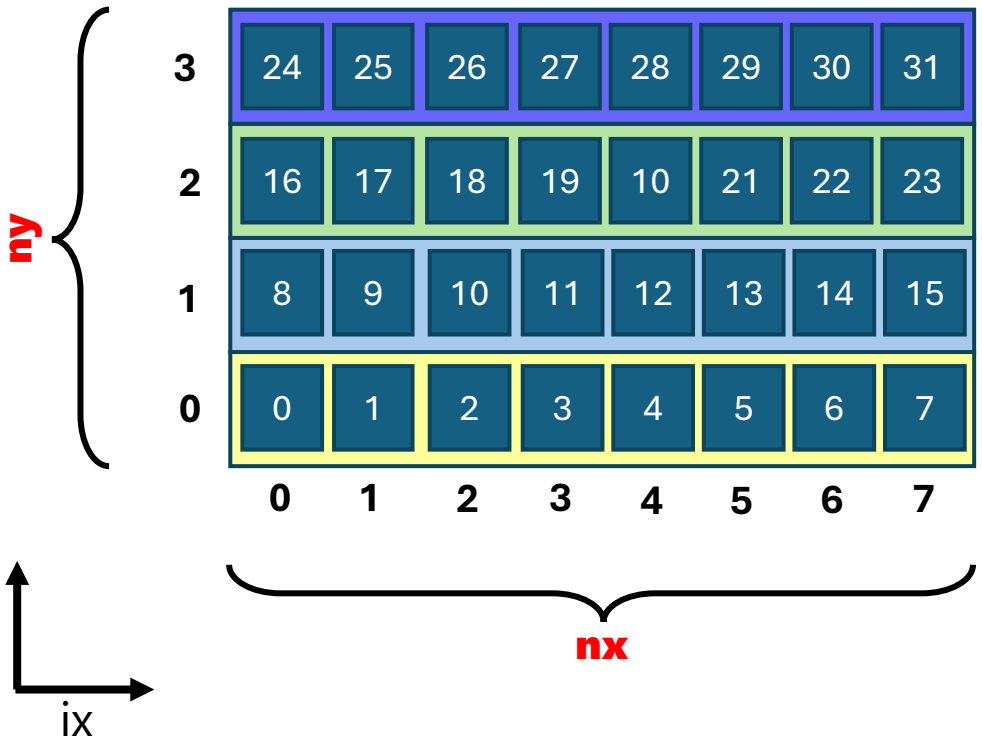
Interlude: rank addressing



Interlude: rank addressing



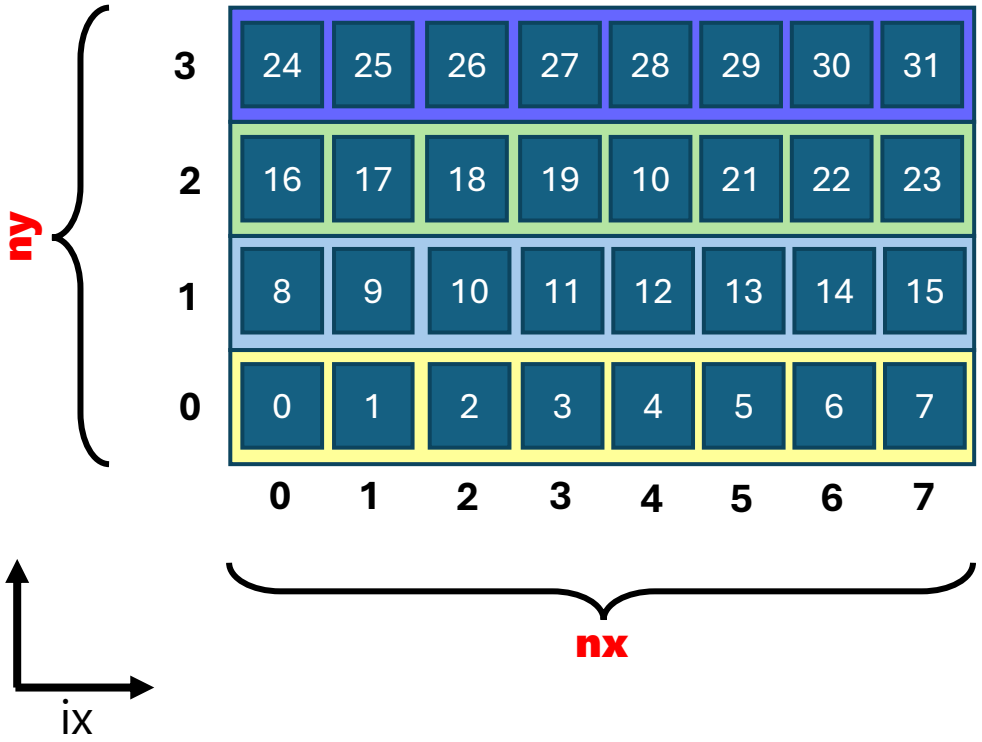
Interlude: rank addressing



idx = iy * nx + ix

“global” rank

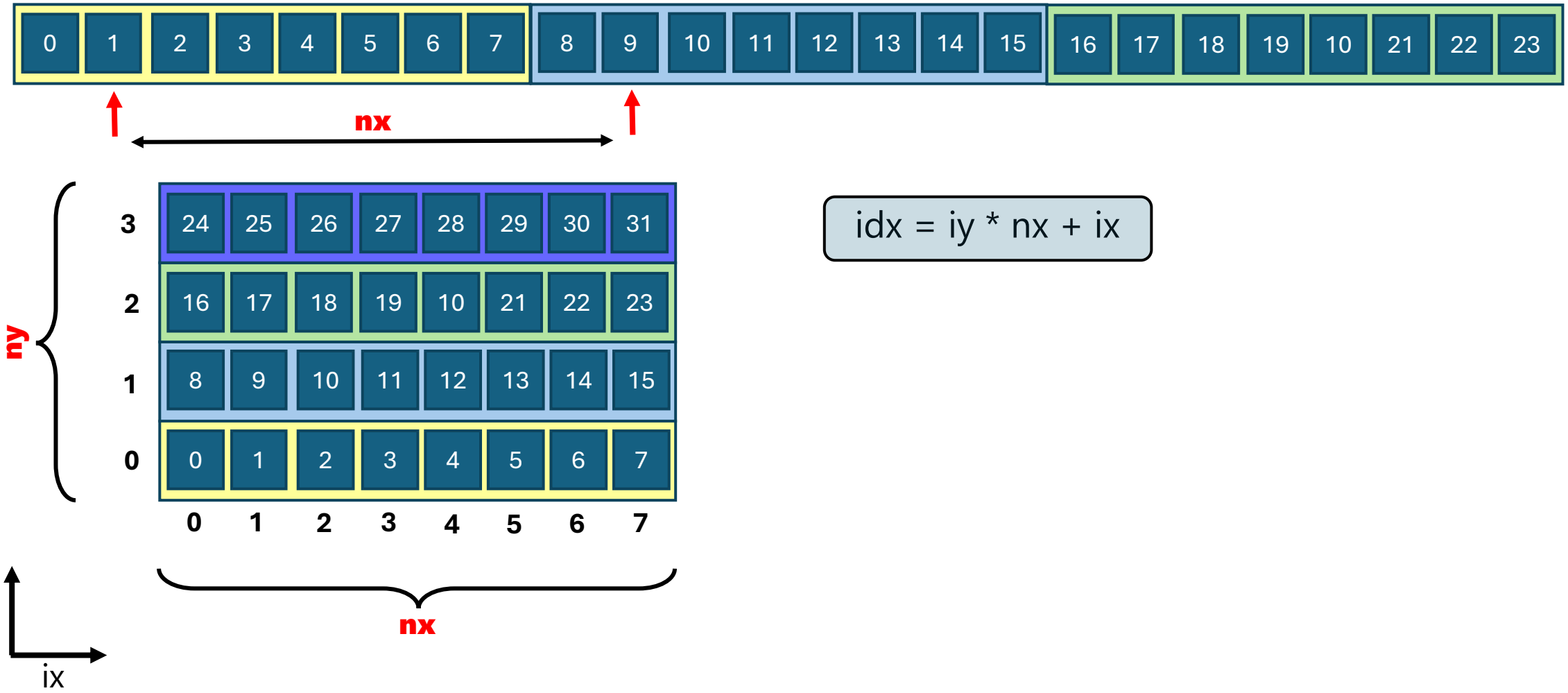
Interlude: rank addressing



idx = iy * nx + ix

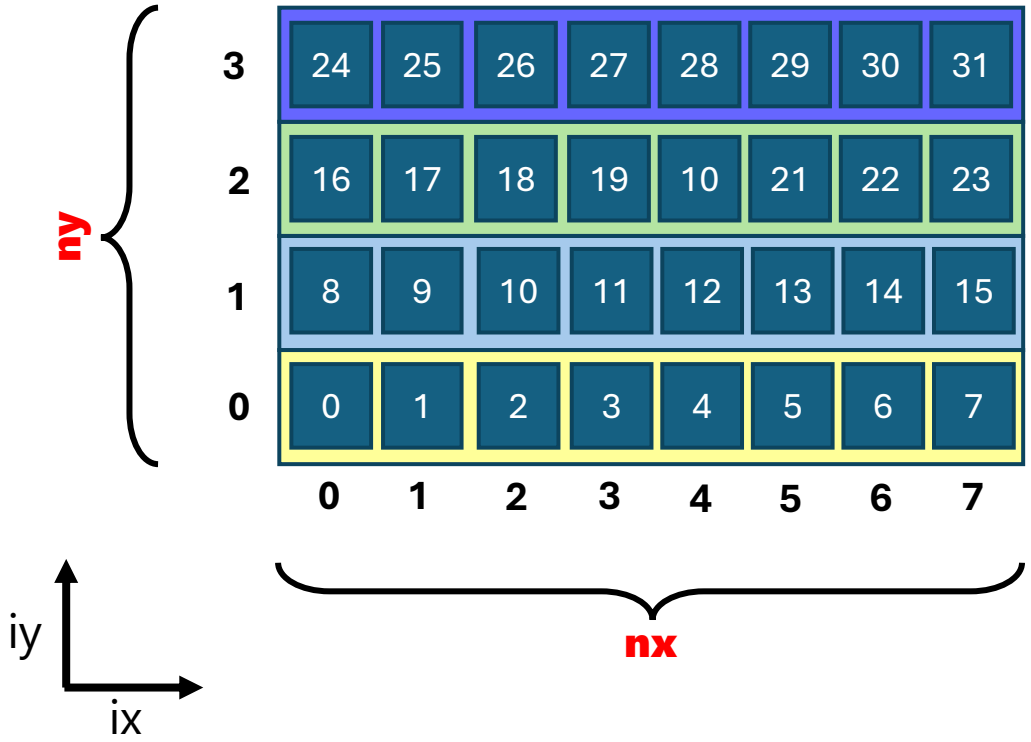
“global” rank

Interlude: rank addressing



idx = iy * nx + ix

Interlude: rank addressing



idx = iy * nx + ix

- Reverse:

  iy = idx / nx
  ix = idx % nx

Hardware and grid

HIP CUDA

core

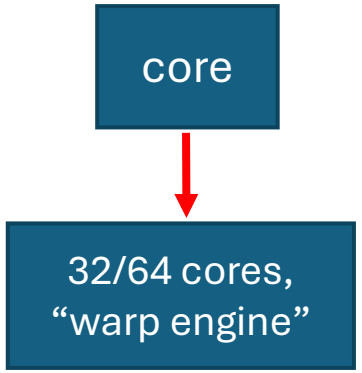
ALU **Compute core**

- ✓ Do not have program counters
- ✓ FP32/int

Hardware and grid

HIP

CUDA



Hardware and grid

HIP

CUDA

core

64 or 32-core warps

32/64 cores, “warp engine”

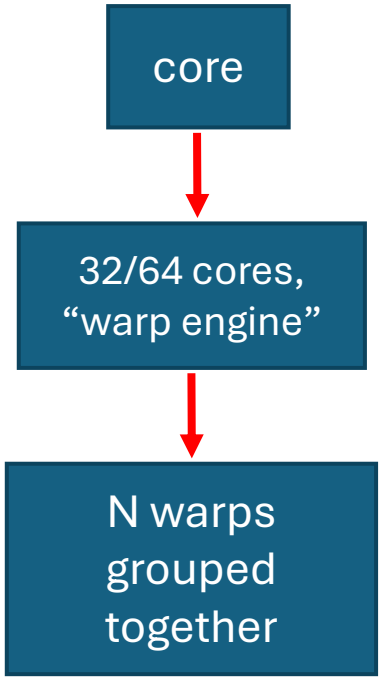
32-core processing blocks

- ✓ Share a single program counter
- ✓ Add FP64

Hardware and grid

HIP

CUDA



Hardware and grid

HIP

CUDA

core

32/64 cores, “warp engine”

N warps grouped together

Compute unit (CU)

Symmetric multiprocessor (SM)

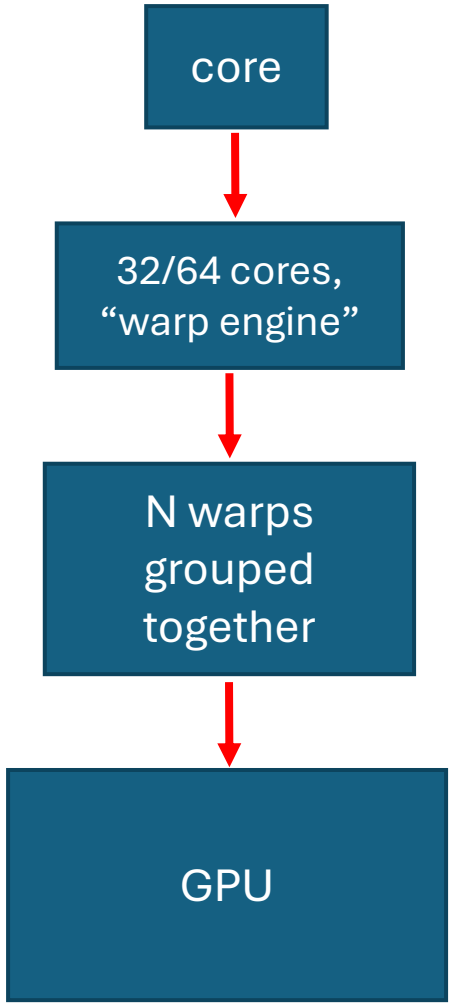
- ✓ Registers
- ✓ Shared memory
- ✓ L1 cash

typically N = 2, 4, depends on a model

Hardware and grid

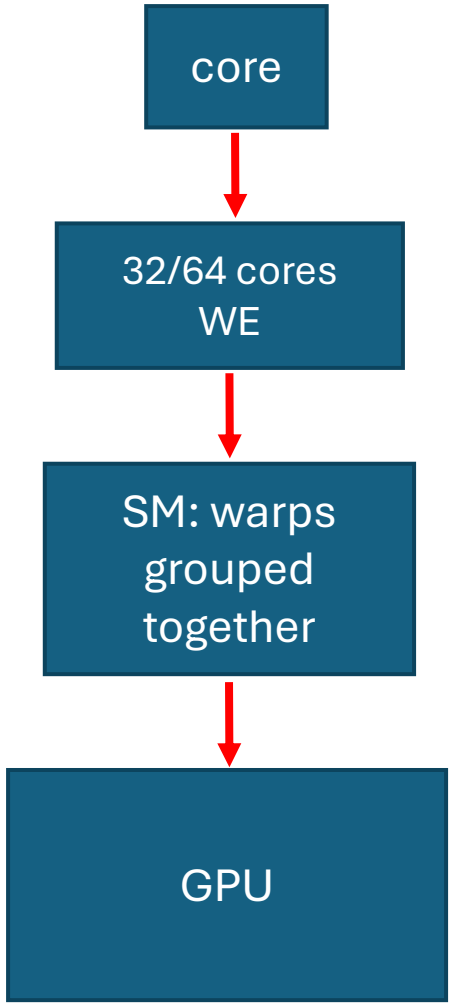
HIP

CUDA

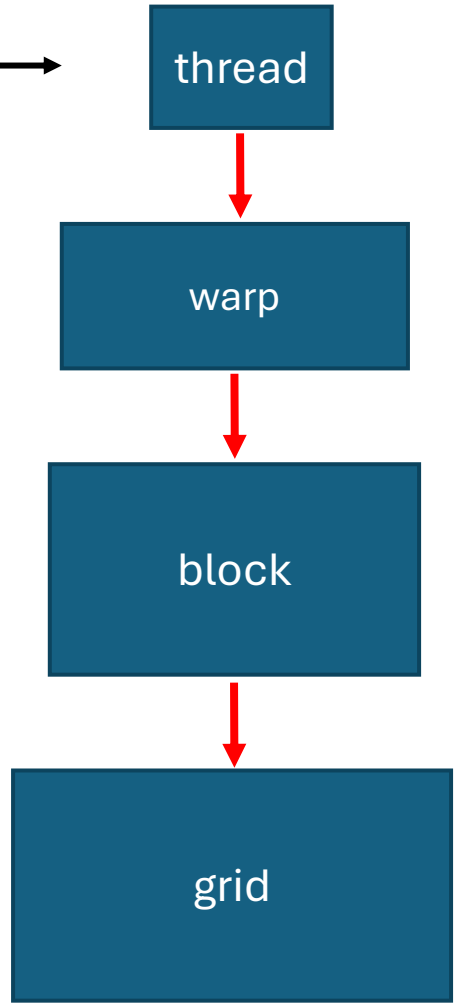


- ✓ Thousands cores
- ✓ Global memory

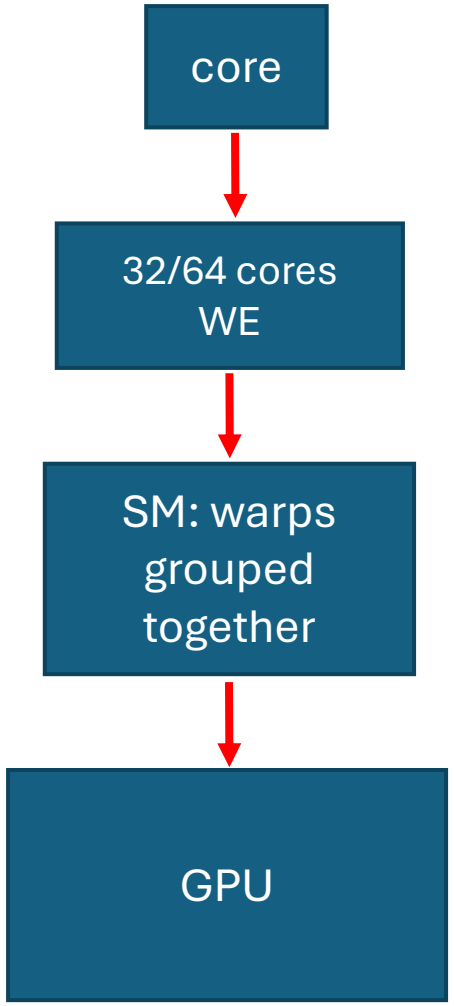
Hardware and grid



✓ Executes instructions (SIMD/T)

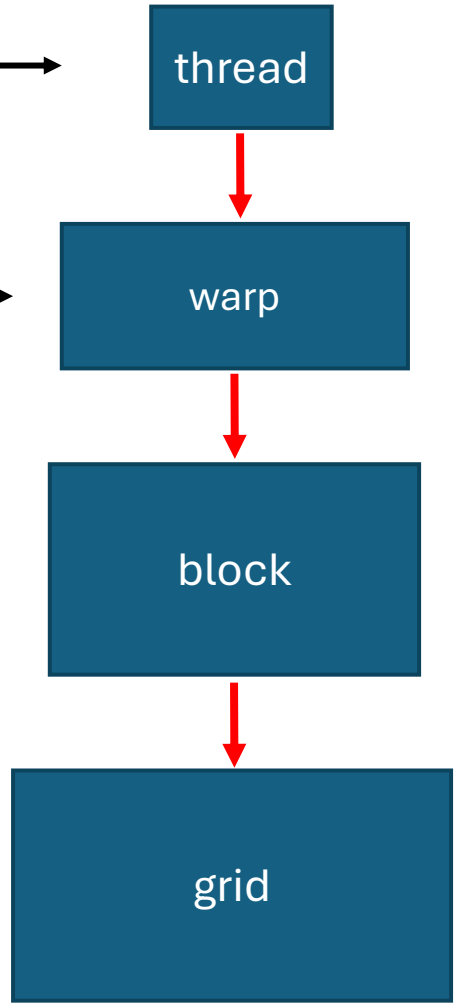


Hardware and grid

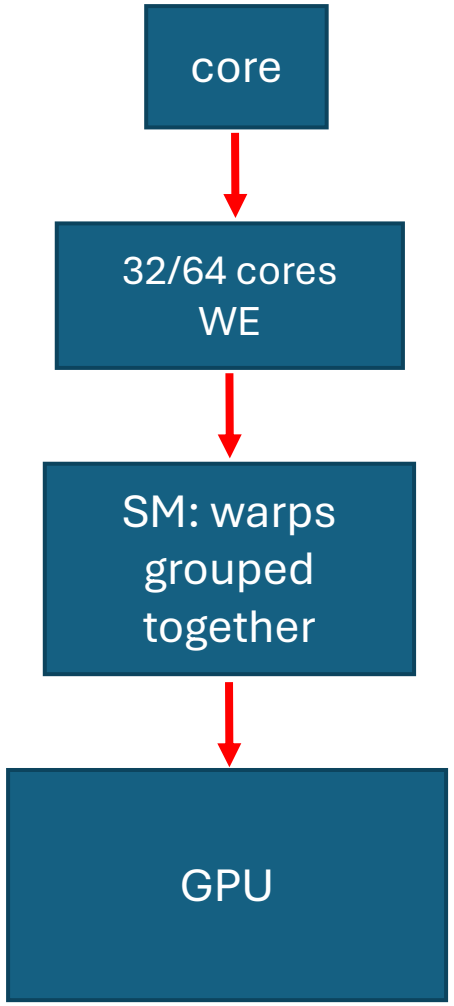


✓ Executes instructions (SIMD/T)

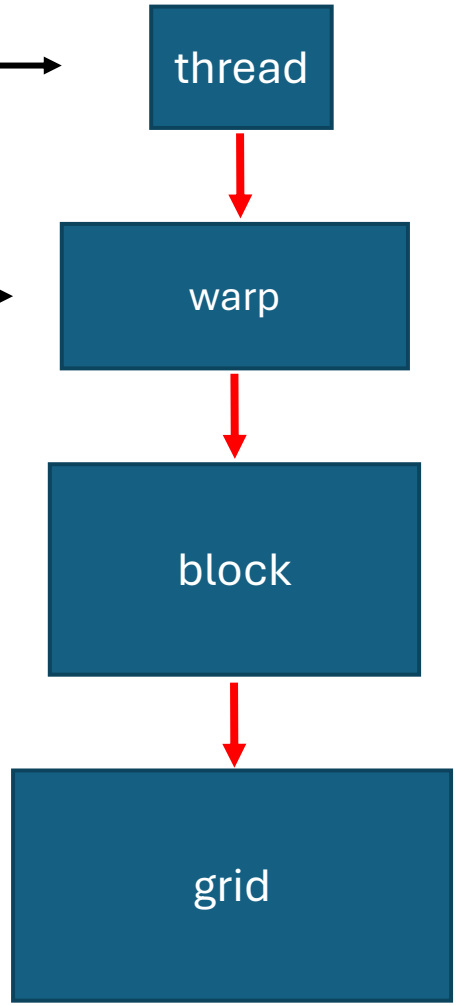
✓ Work in lock-step



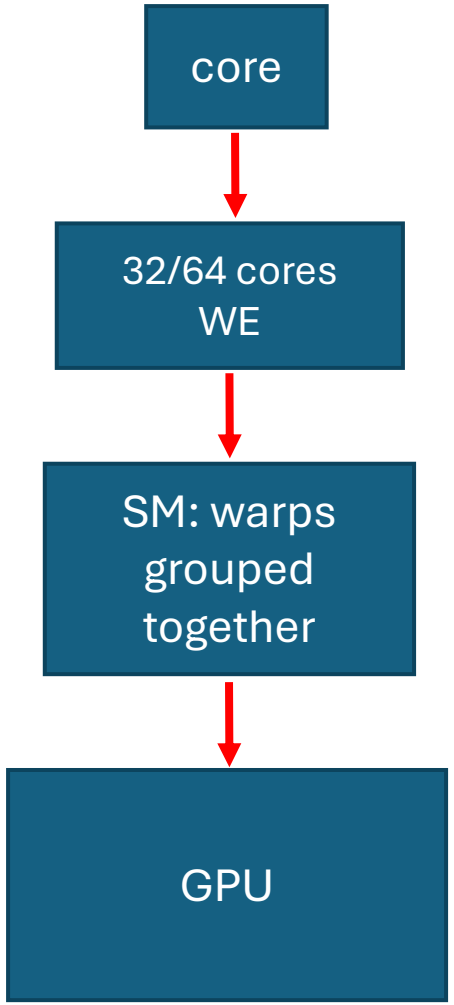
Hardware and grid



- ✓ Executes instructions (SIMD/T) → thread
- ✓ Work in lock-step → warp
- ✓ One block runs on one SM
- ✓ Different blocks may run on different SMs
- ✓ Threads in block communicate → block



Hardware and grid



✓ Executes instructions (SIMD/T) → thread

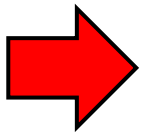
✓ Work in lock-step → warp

- ✓ One block runs on one SM
- ✓ Different blocks may run on different SMs
- ✓ Threads in block communicate → block

- ✓ The bigger, the better
- ✓ Latency hiding: resident threads
- ✓ Unpredictable thread order → grid

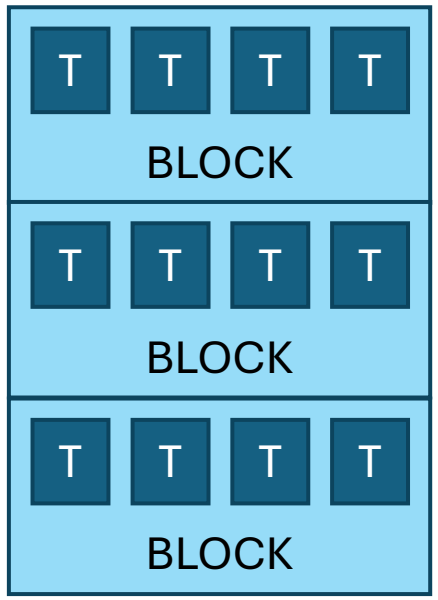
… and more about grid

- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



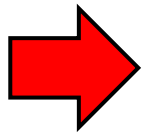
“global” thread rank

tid = blockIdx.x * blockDim.x + threadIdx.x



… and more about grid

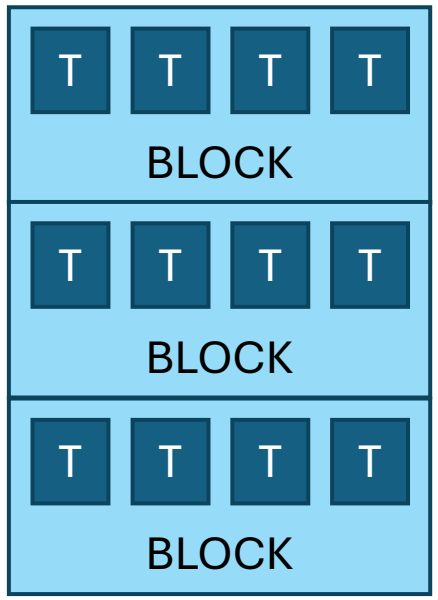
- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



"global" thread rank

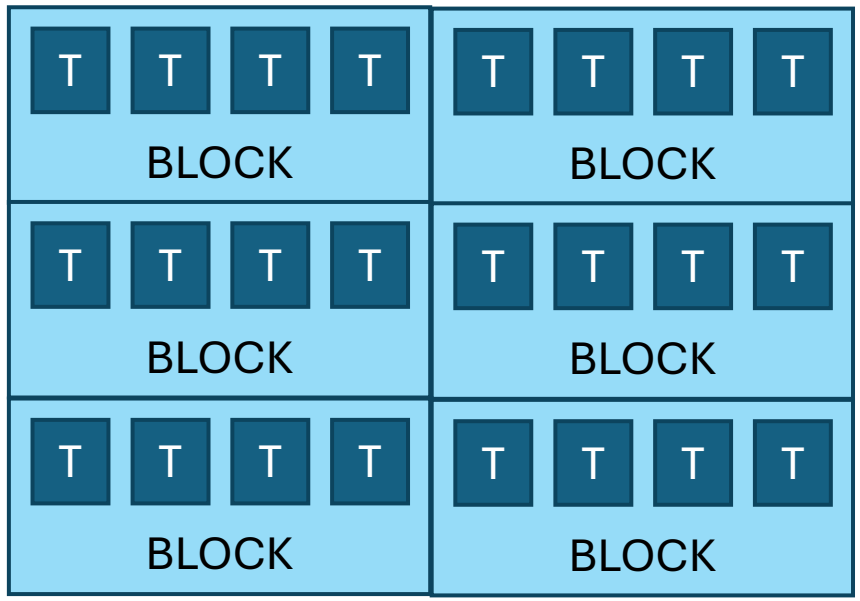
tid = blockIdx.x * blockDim.x + threadIdx.x

- ➢ Max number of threads in block: 1024
- ➢ Must be a multiple of 32 (64)



… and more about grid

- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



“global” thread rank

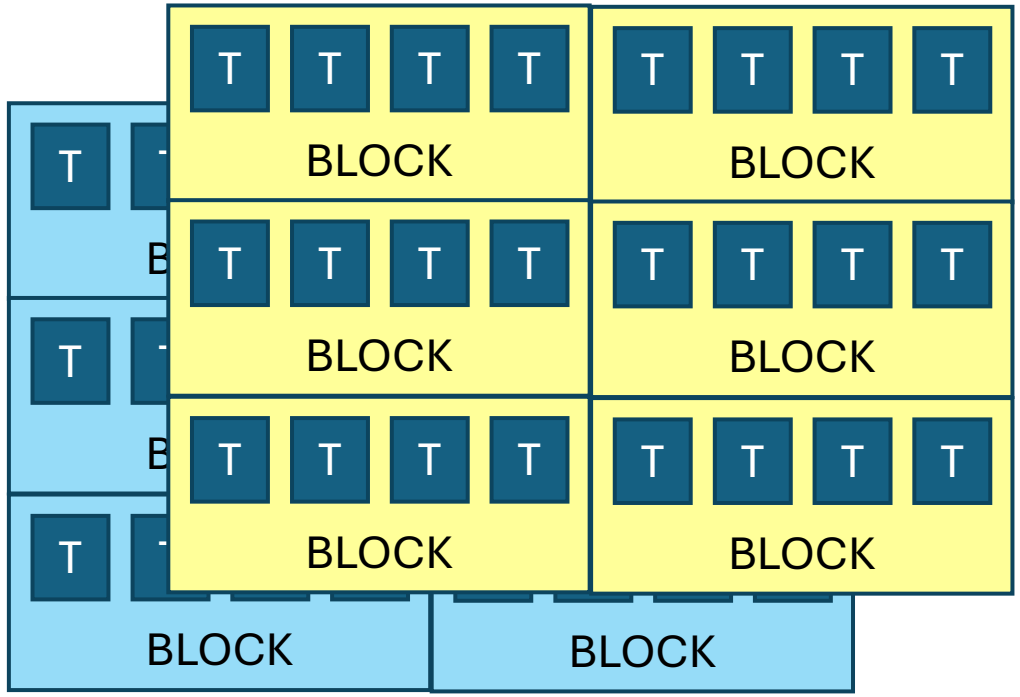
tid = blockIdx.x * blockDim.x + threadIdx.x

- ➢ Max number of threads in block: 1024
- ➢ Must be a multiple of 32 (64)

- ➢ Threads and blocks can be packed 2D …

… and more about grid

- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



“global” thread rank

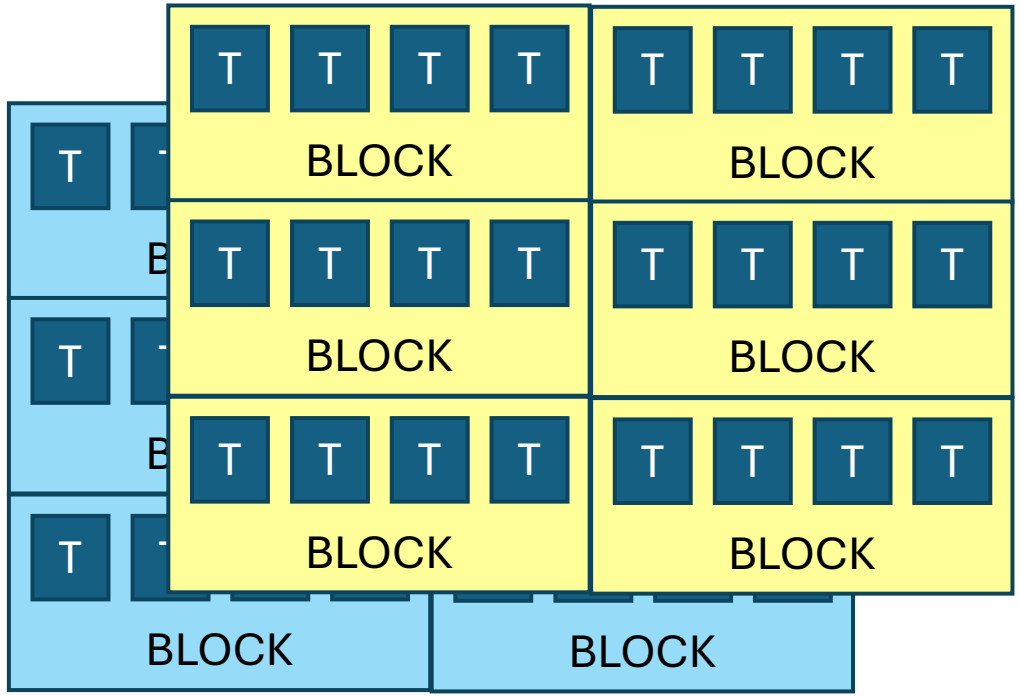
tid = blockIdx.x * blockDim.x + threadIdx.x

- ➢ Max number of threads in block: 1024
- ➢ Must be a multiple of 32 (64)

- ➢ Threads and blocks can be packed 2D and 3D

… and more about grid

- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



"global" thread rank

tid = blockIdx.x * blockDim.x + threadIdx.x

- ➢ Max number of threads in block: 1024
- ➢ Must be a multiple of 32 (64)

- ➢ Threads and blocks can be packed 2D and 3D

2D threads on 2D blocks:

blockId = (gridDim.x * blockIdx.y) + blockIdx.x

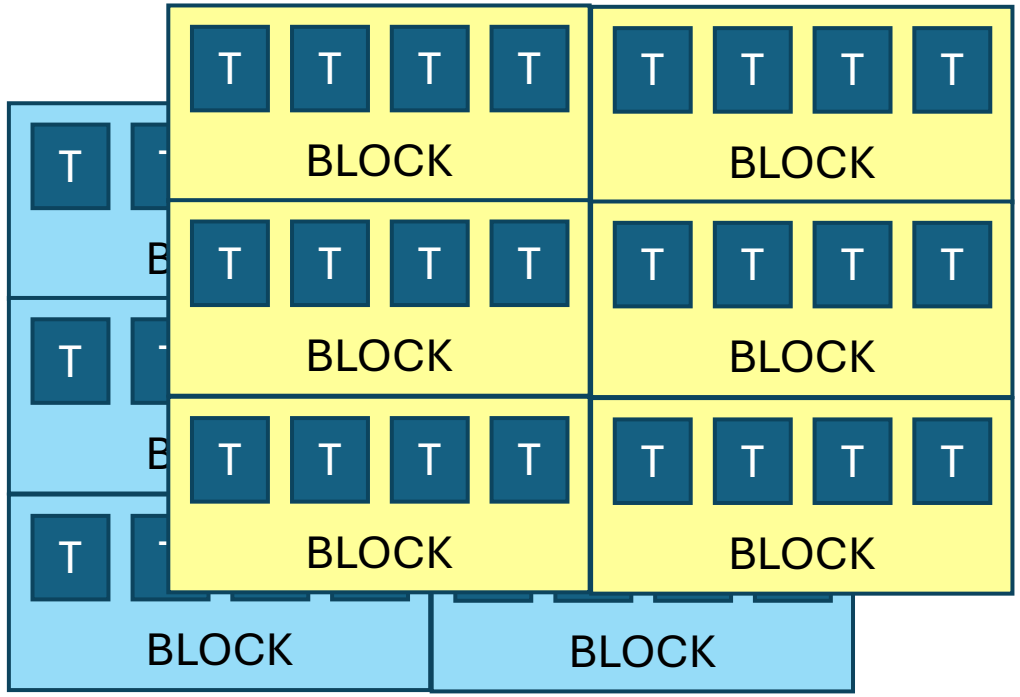
tid = (blockId * (blockDim.x * blockDim.y))

+(threadIdx.y * blockDim.x)

+ threadIdx.x

… and more about grid

- ✓ Thread rank in block: **threadIdx.x**
- ✓ Block rank in greed: **blockIdx.x**
- ✓ Number of threads in block: **blockDim.x**
- ✓ Number of blocks in grid: **gridDim.x**



"global" thread rank

tid = blockIdx.x * blockDim.x + threadIdx.x

Fastest running index –

neighbouring threads in memory

blockId = (gridDim.x * blockIdx.y) + blockIdx.x

tid = (blockId * (blockDim.x * blockDim.y))

+(threadIdx.y * blockDim.x)

+ threadIdx.x

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

h = host = CPU
d = device = GPU

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

Allocate CPU and GPU memory

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

h = host = CPU
d = device = GPU

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

Allocate CPU and GPU memory

Copy data to and from GPU

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

h = host = CPU
d = device = GPU

- ✓ *hipMemcpy*: where, from where, how much data, H2D or D2H
- ✓ It is a **blocking** instruction

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);
hipDeviceSynchronize();
hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

Allocate CPU and GPU memory

Copy data to and from GPU

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

h = host = CPU
d = device = GPU

- ✓ *hipMemcpy*: where, from where, how much data, H2D or D2H
- ✓ It is a **blocking** instruction

Other blocking instruction:

hipDeviceSynchronize

Data transfer

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

Allocate CPU and GPU memory

Copy data to and from GPU

Free memory when finished

- CPU and GPU **cannot access** and modify each other's memory directly
- But they **can copy** data from each other

h = host = CPU
d = device = GPU

- ✓ *hipMemcpy*: where, from where, how much data, H2D or D2H
- ✓ It is a **blocking** instruction

Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

Kernel launching

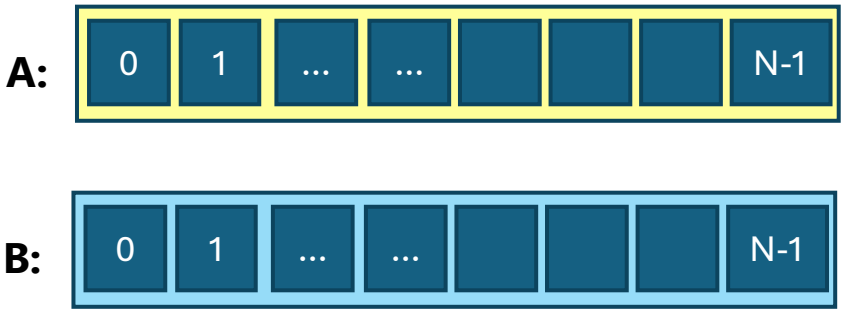
```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

- Step 1: create a grid for your task

Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

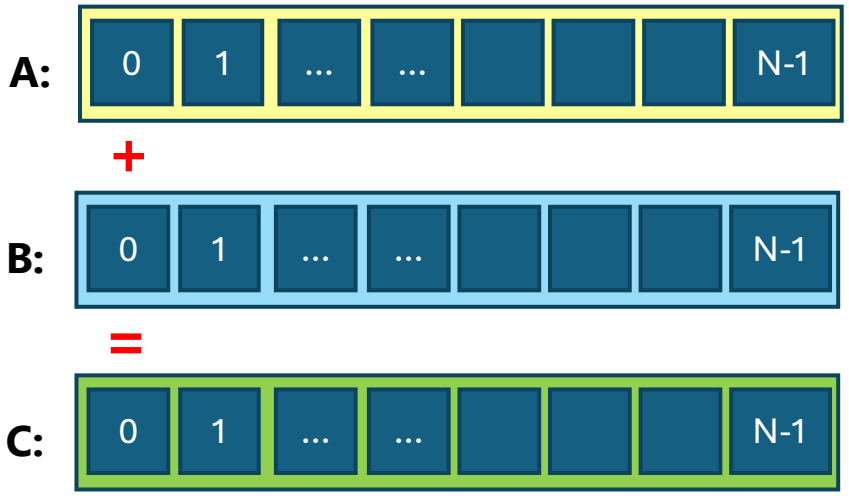
➢ Step 1: create a grid for your task



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

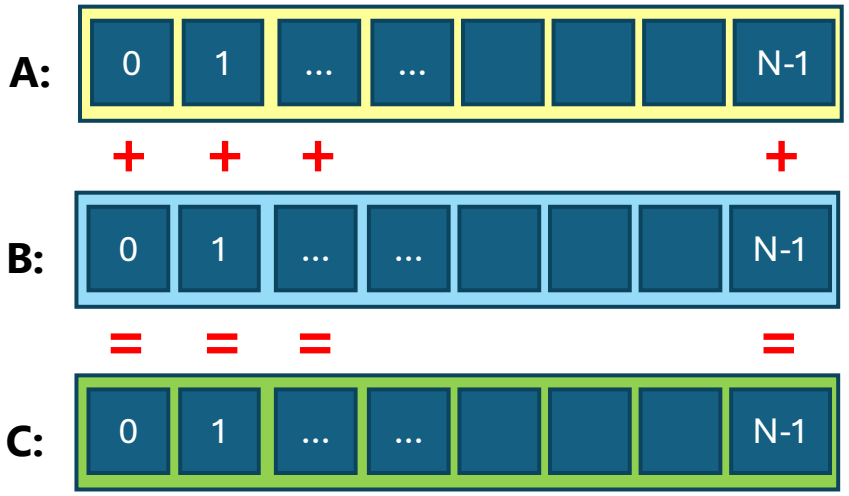
➢ Step 1: create a grid for your task



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

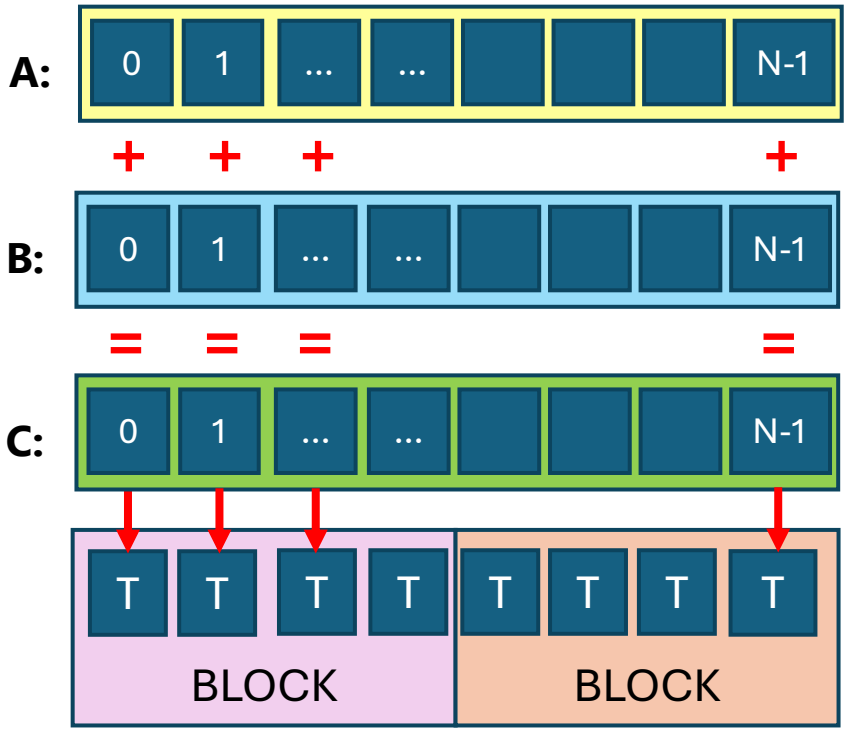
➢ Step 1: create a grid for your task



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

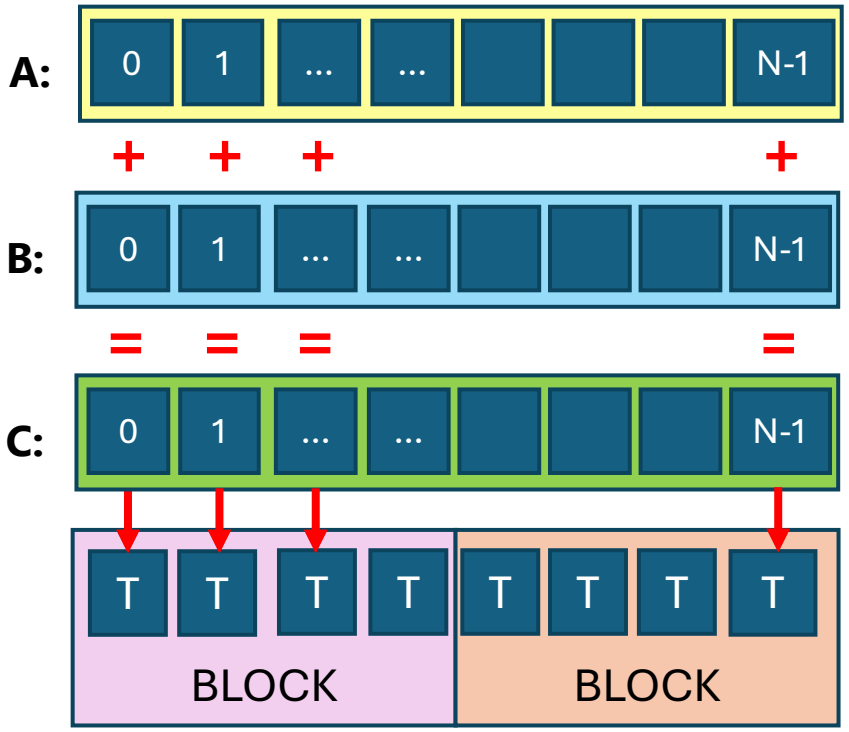
➢ Step 1: create a grid for your task



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

➢ Step 1: create a grid for your task

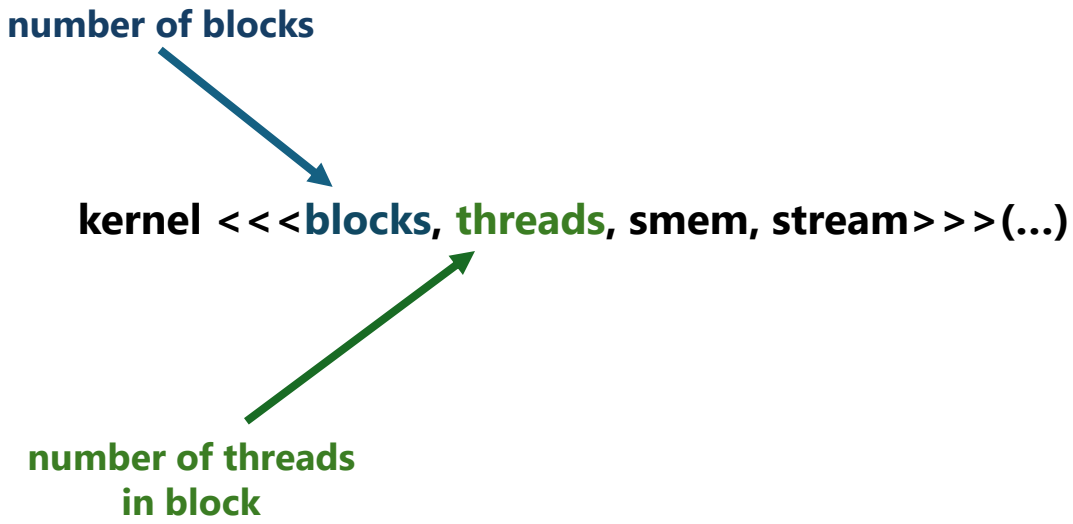


- ✓ Total *num_threads* = cover whole data
- ✓ Select block size
- ✓ *Number of blocks* = num_threads / block_size (round up)

Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

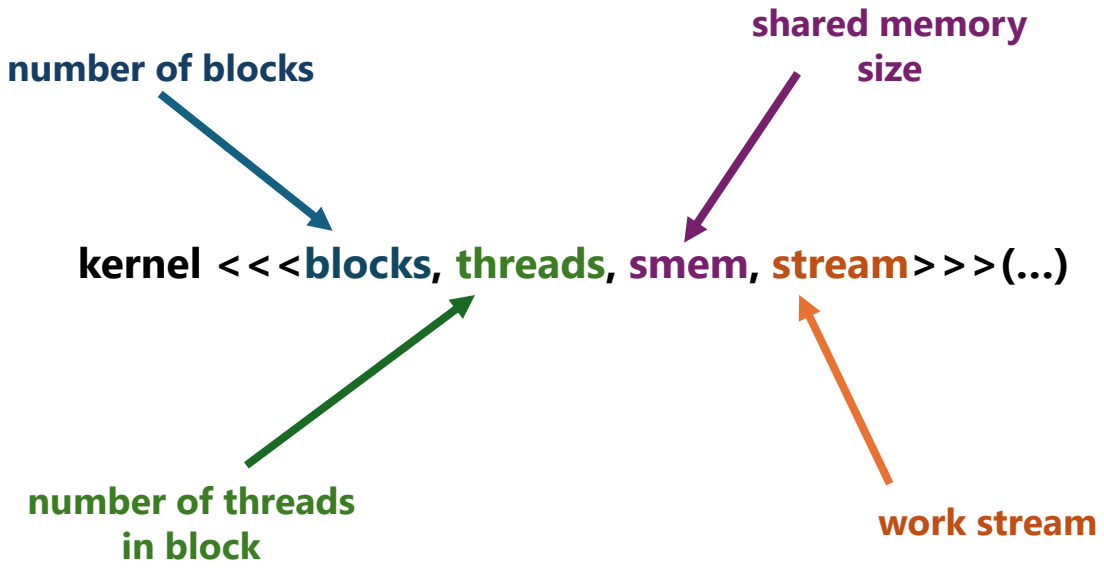
➢ Step 2: launch kernel



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

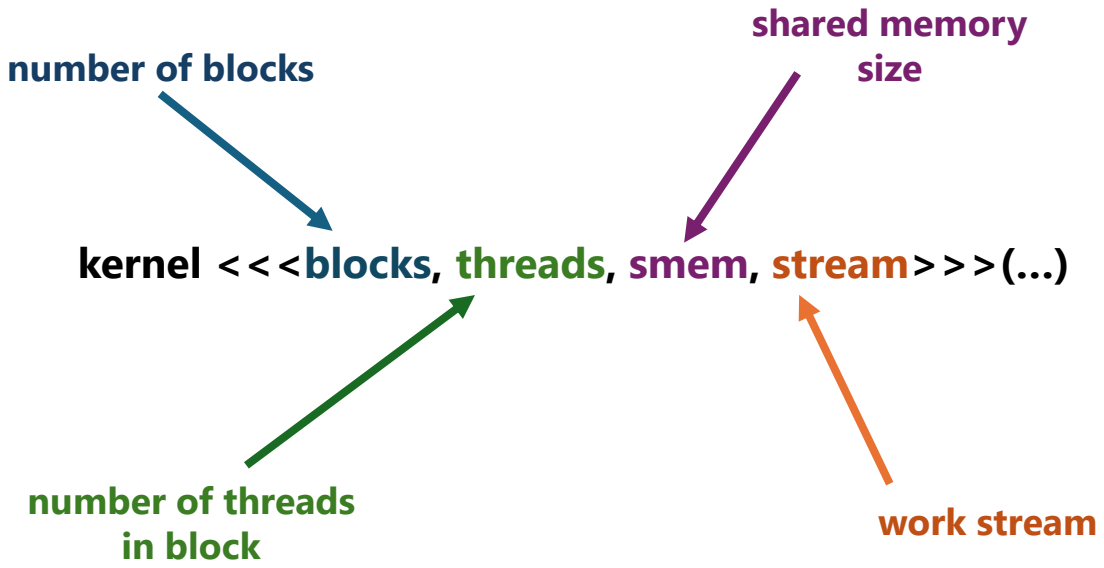
➢ Step 2: launch kernel



Kernel launching

```
const int N = 10000;
float *Ah, *Bh, *Ch, *Cref;
float *Ad, *Bd, *Cd;

Ah = (float *)malloc(N * sizeof(float)); Bh = (float *)malloc(N * sizeof(float));
Ch = (float *)malloc(N * sizeof(float)); Cref = (float *)malloc(N * sizeof(float));

for (int i = 0; i < N; i++) {
  Ah[i] = sin(i) * 2.3;
  Bh[i] = cos(i) * 1.1;
  Cref[i] = Ah[i] + Bh[i];
}

hipMalloc((void **)&Ad, N * sizeof(float));
hipMalloc((void **)&Bd, N * sizeof(float));
hipMalloc((void **)&Cd, N * sizeof(float));

hipMemcpy(Ad, Ah, sizeof(float) * N, hipMemcpyHostToDevice);
hipMemcpy(Bd, Bh, sizeof(float) * N, hipMemcpyHostToDevice);

dim3 blocks, threads;
threads = dim3(256, 1, 1);
blocks = dim3((N + 256 - 1) / 256, 1, 1);

// hipLaunchKernelGGL(vector_add, blocks, threads, 0, 0, Ad, Bd, Cd, N); // or
vector_add<<<blocks, threads, 0, 0>>>(Ad, Bd, Cd, N);

hipMemcpy(Ch, Cd, sizeof(float) * N, hipMemcpyDeviceToHost);

hipFree(Ad); hipFree(Bd); hipFree(Cd);
free(Ah); free(Bh); free(Ch); free(Cref);
```

➢ Step 2: launch kernel



✓ Acceptable launch:

kernel <<<1, 1>>>(...)

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

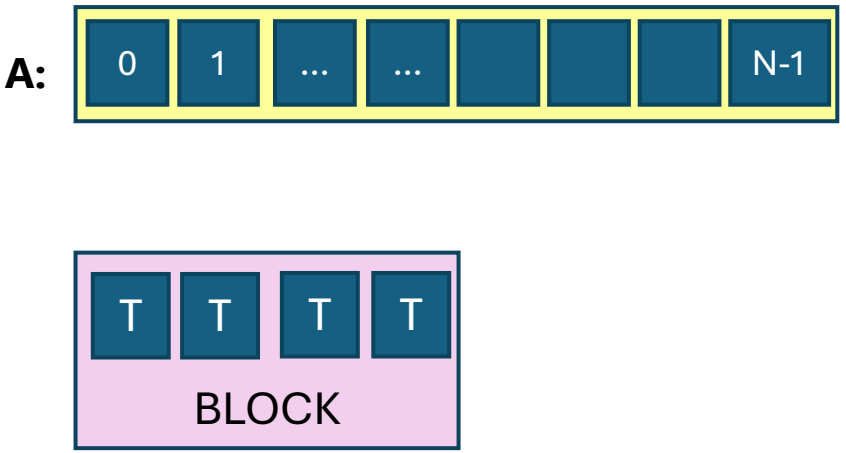
Alternative: number of threads < number of tasks:

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

Alternative: number of threads < number of tasks:

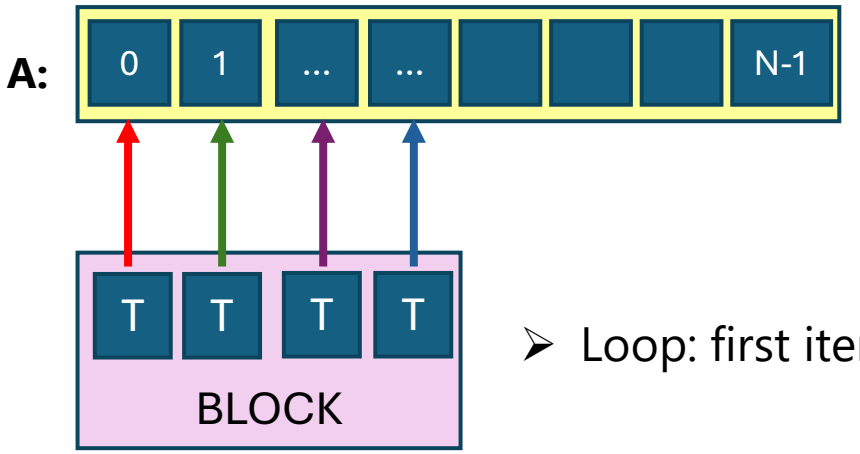


Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

Alternative: number of threads < number of tasks:



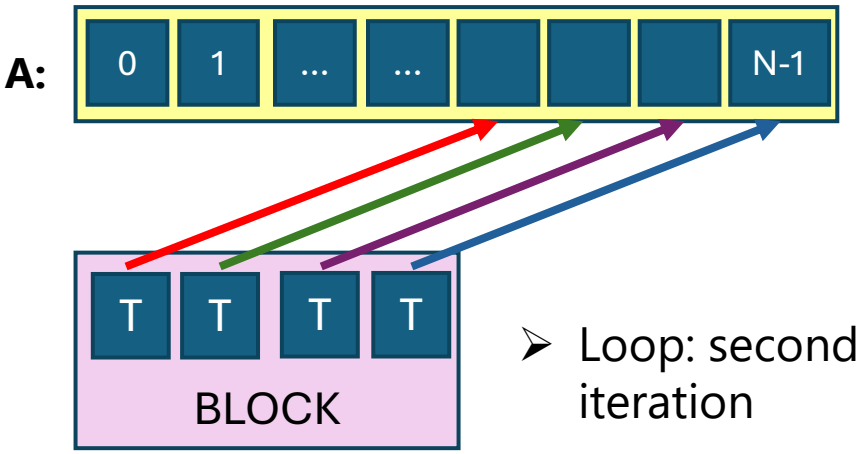
- Loop: first iteration

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

Alternative: number of threads < number of tasks:



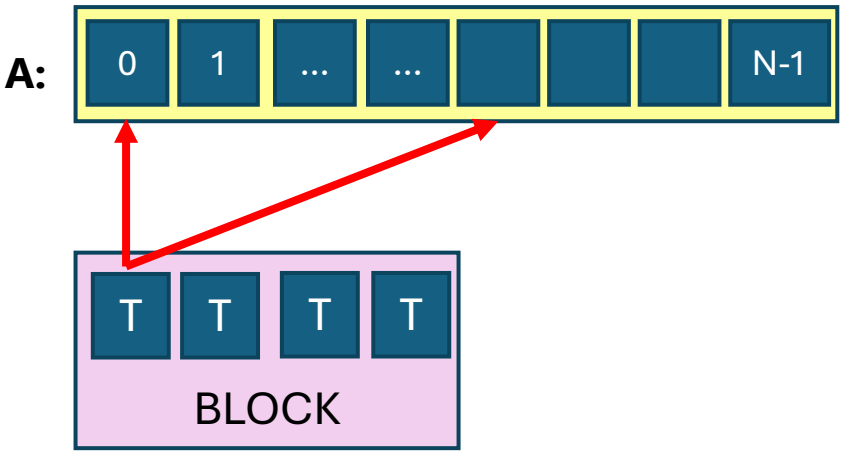
- Loop: second iteration

Kernel itself

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
```

- __global__ : run on D, called from H
  __device__ : run on D, called from D
  __host__ : run on H, called from H
- Global: always **void**
- Find global thread index
- Out of range check
- Each thread calculates one sum

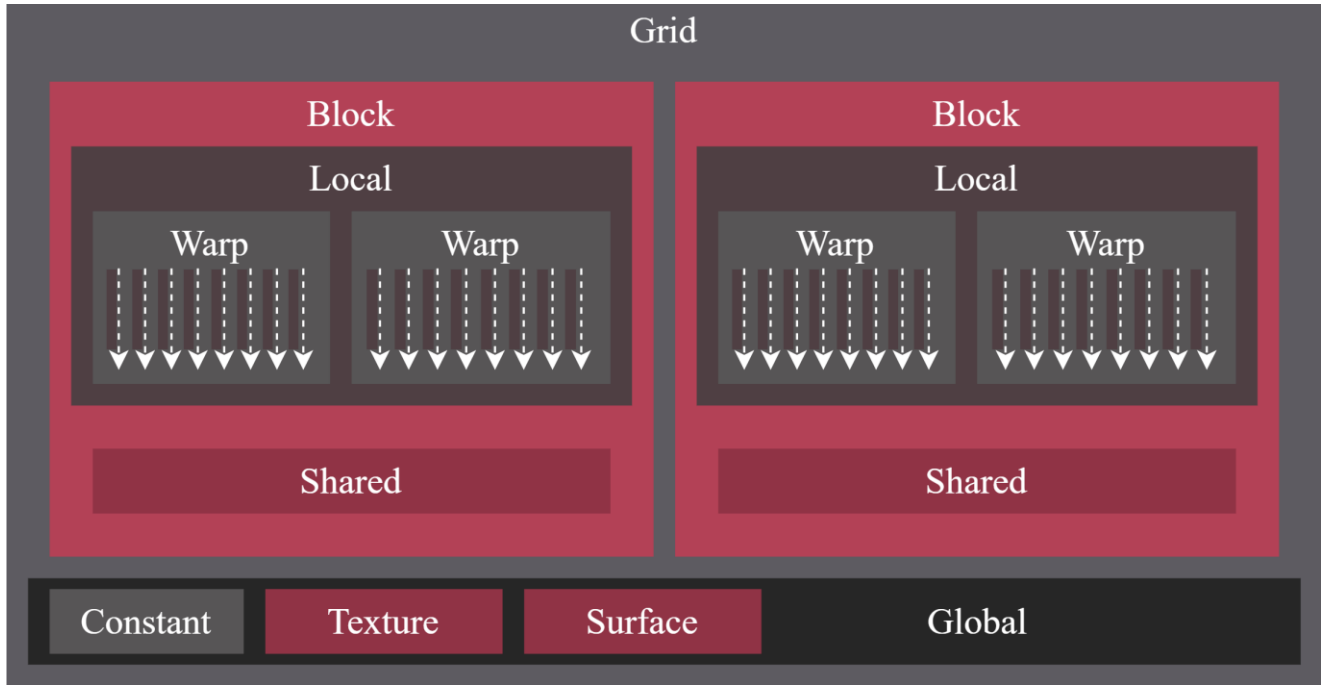
Alternative: number of threads < number of tasks:



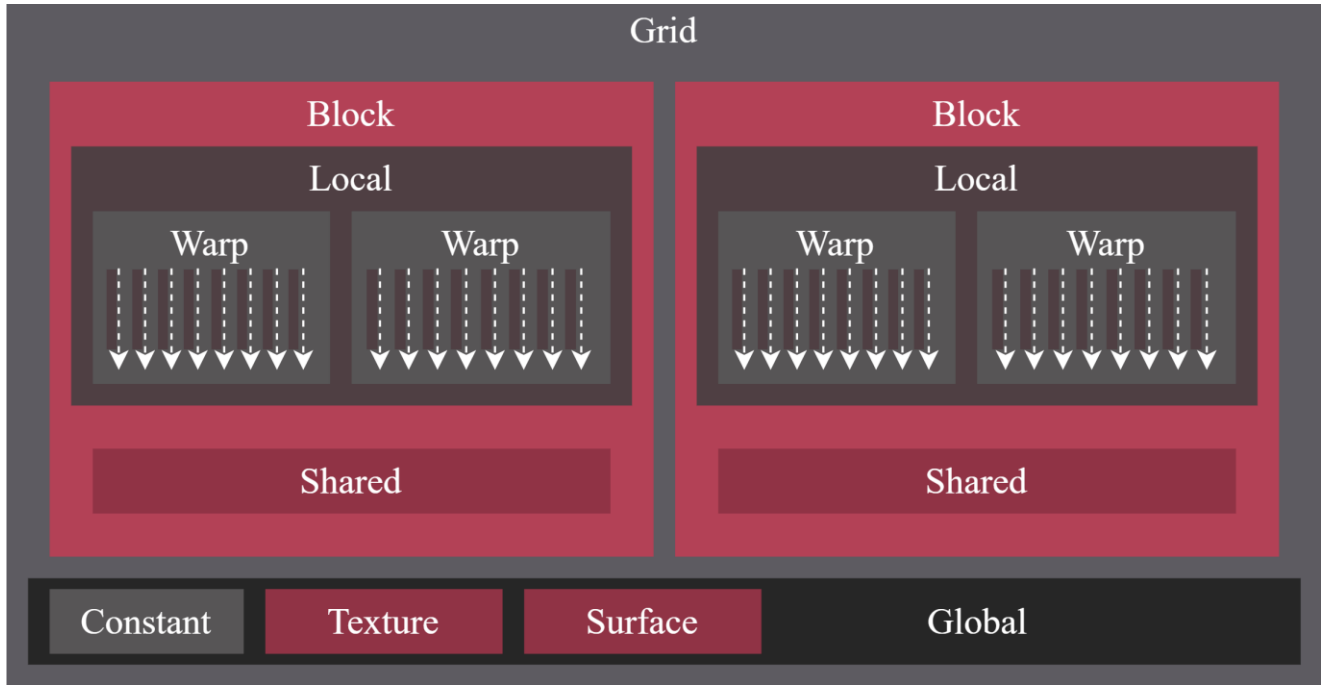

```
__global__ void vector_add(float *A, float *B, float *C, int n) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  while (tid < n) {
    C[tid] = A[tid] + B[tid];
    tid += gridDim.x*blockDim.x;
  }
}
```

- **Thread linear addressing**: grid-size stride between passes through the loop

Shared memory

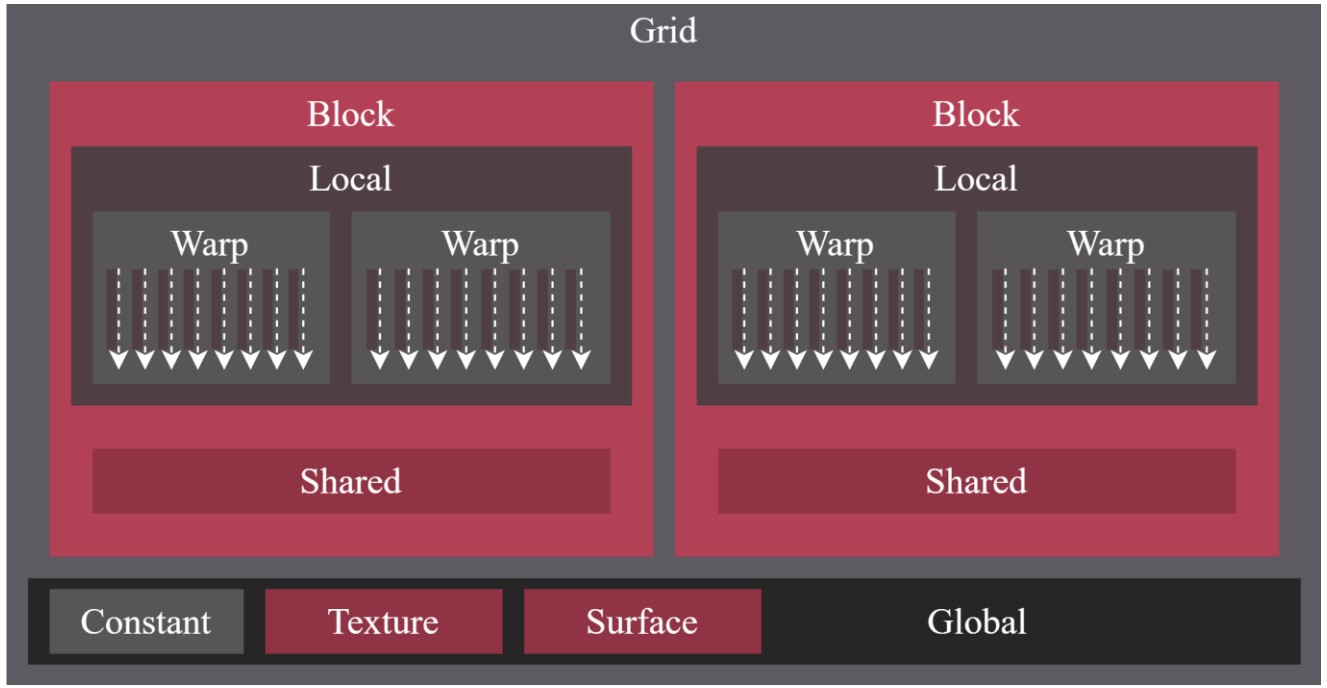


Shared memory



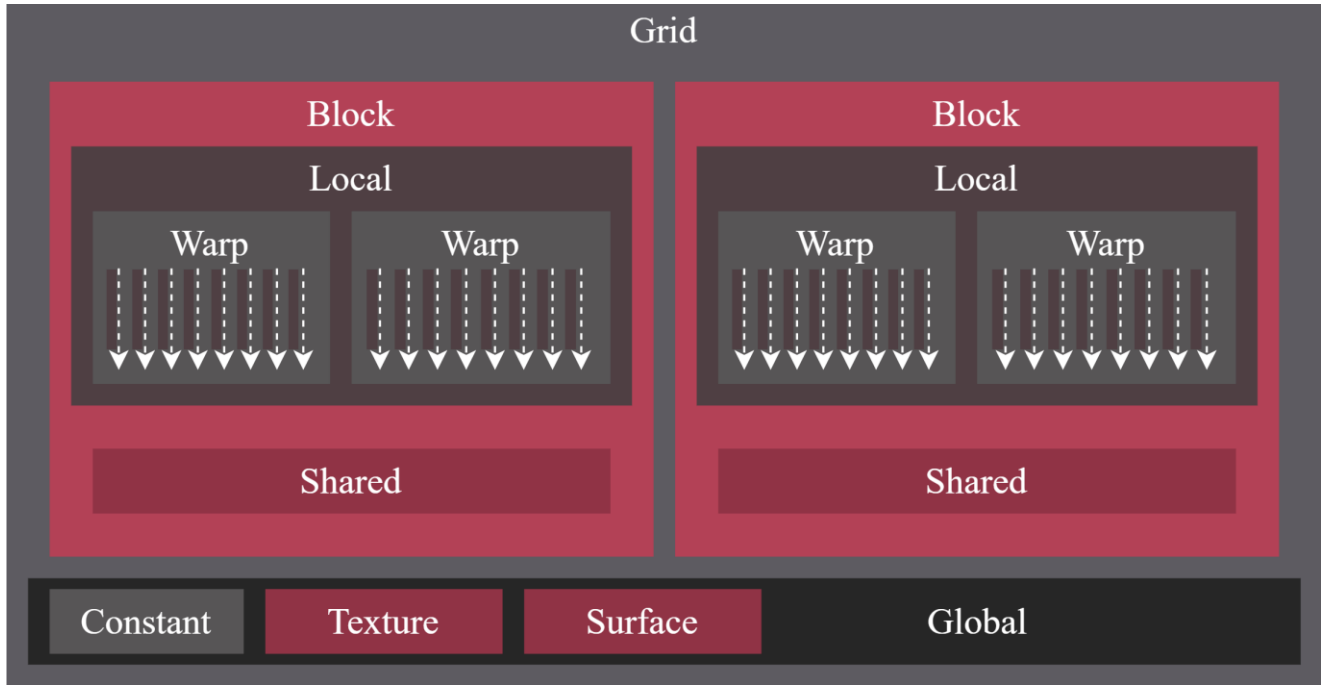
- ✓ **Hardware**: provided by SM, split between blocks on it

Shared memory



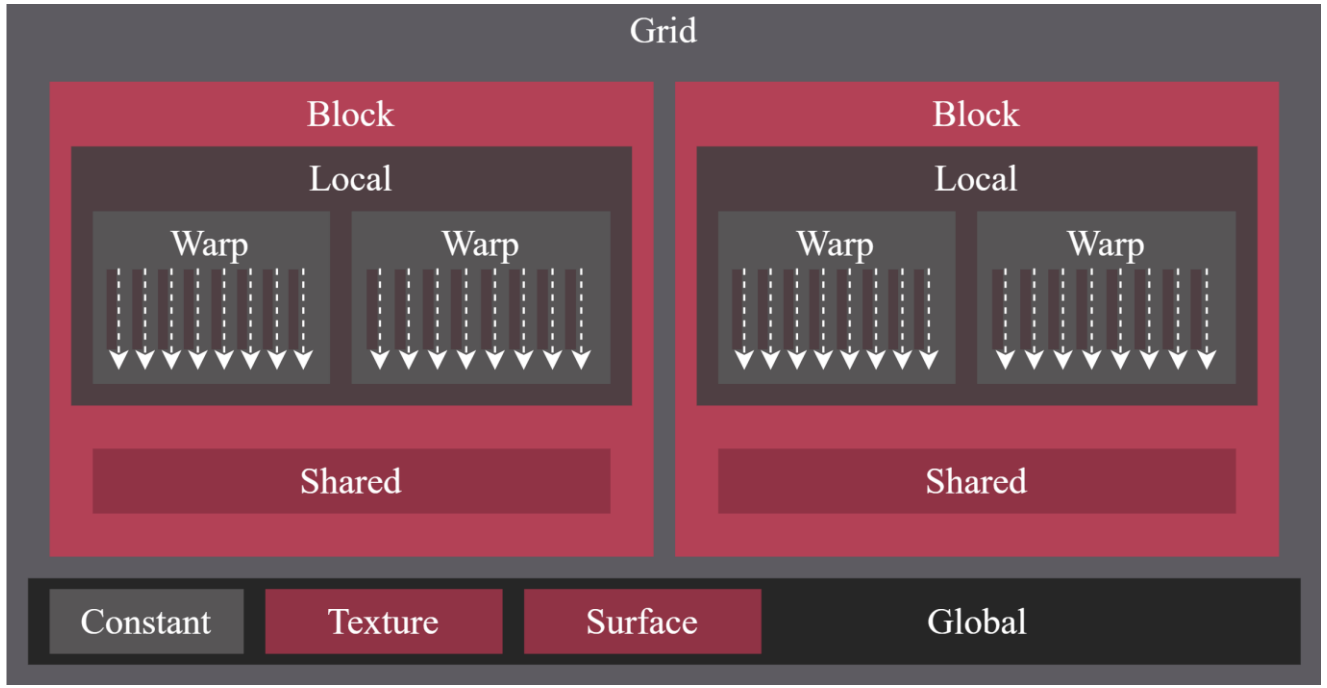
- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster

Shared memory



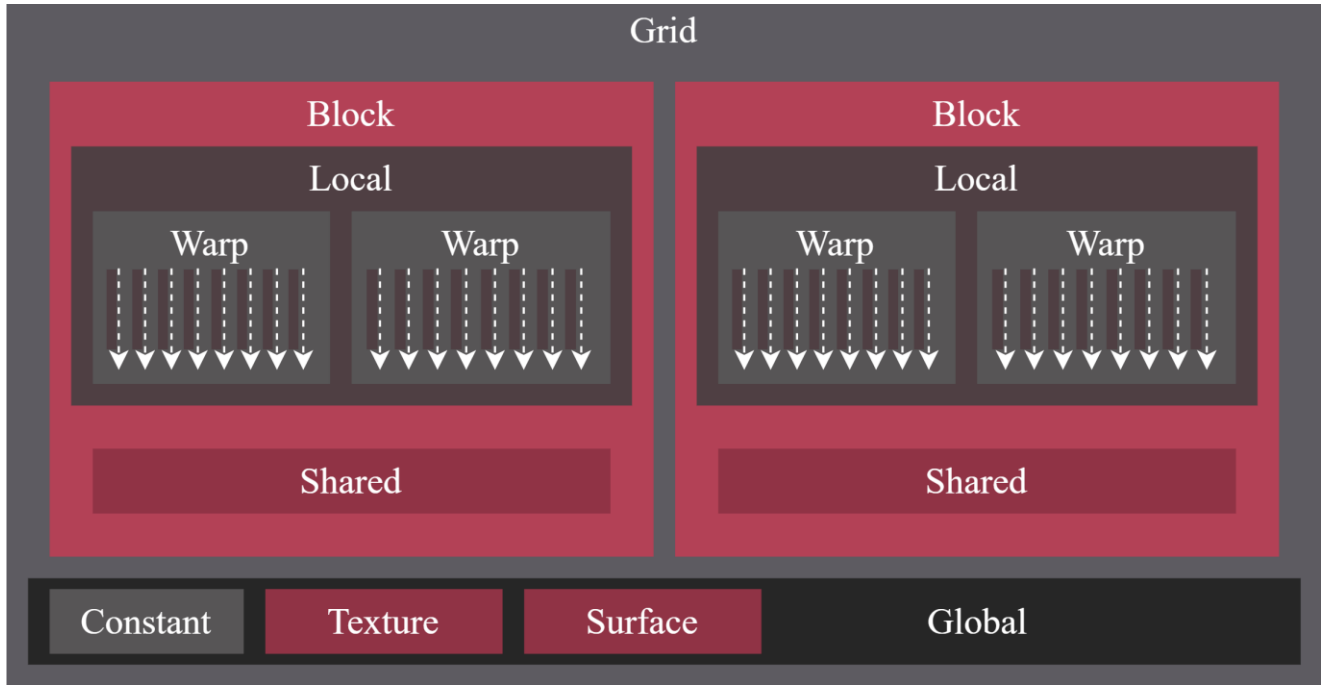
- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster
- ✓ Accessible to all threads in a block

Shared memory



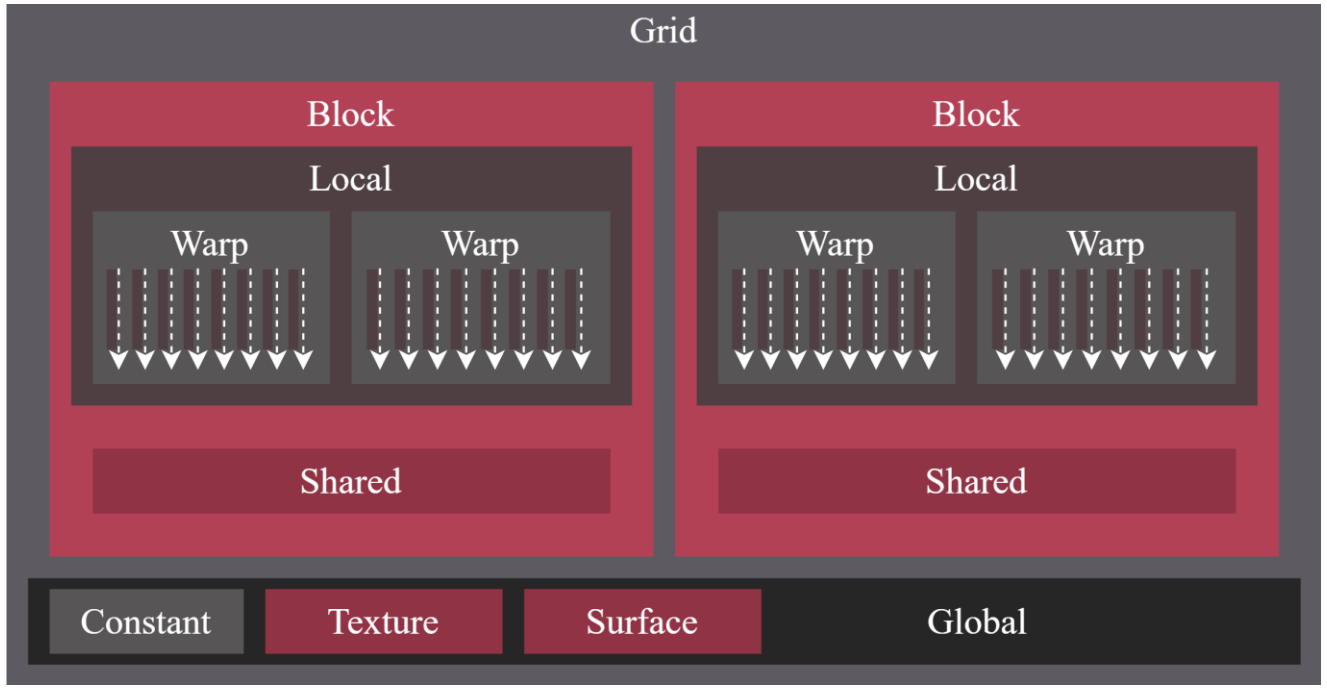
- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster
- ✓ Accessible to all threads in a block
- ✓ Limited size: too much requested -> only one active thread block

Shared memory



- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster
- ✓ Accessible to all threads in a block
- ✓ Limited size: too much requested -> only one active thread block
- ✓ Does not require **coalescing**

Shared memory



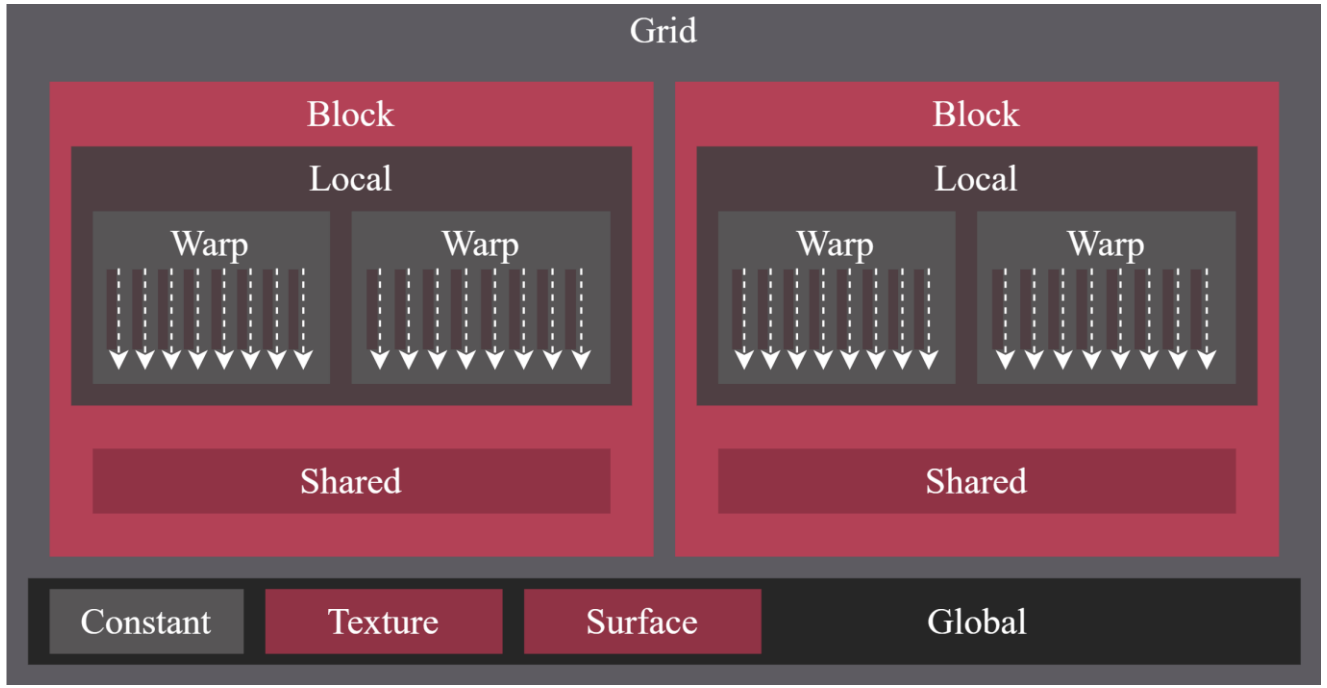
- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster
- ✓ Accessible to all threads in a block
- ✓ Limited size: too much requested -> only one active thread block
- ✓ Does not require **coalescing**

➢ Static

```
__shared__ float data[256];
```

Declared inside kernel;
Size known at compile time

Shared memory



- ✓ **Hardware**: provided by SM, split between blocks on it
- ✓ Faster
- ✓ Accessible to all threads in a block
- ✓ Limited size: too much requested -> only one active thread block
- ✓ Does not require **coalescing**

➢ Static

```
__shared__ float data[256];
```

Declared inside kernel;
Size known at compile time

➢ Dynamic

```
extern __shared__ float data[];
```

Size passed through kernel launch parameter **smem** ;
More flexible, but
Having more than one requires care

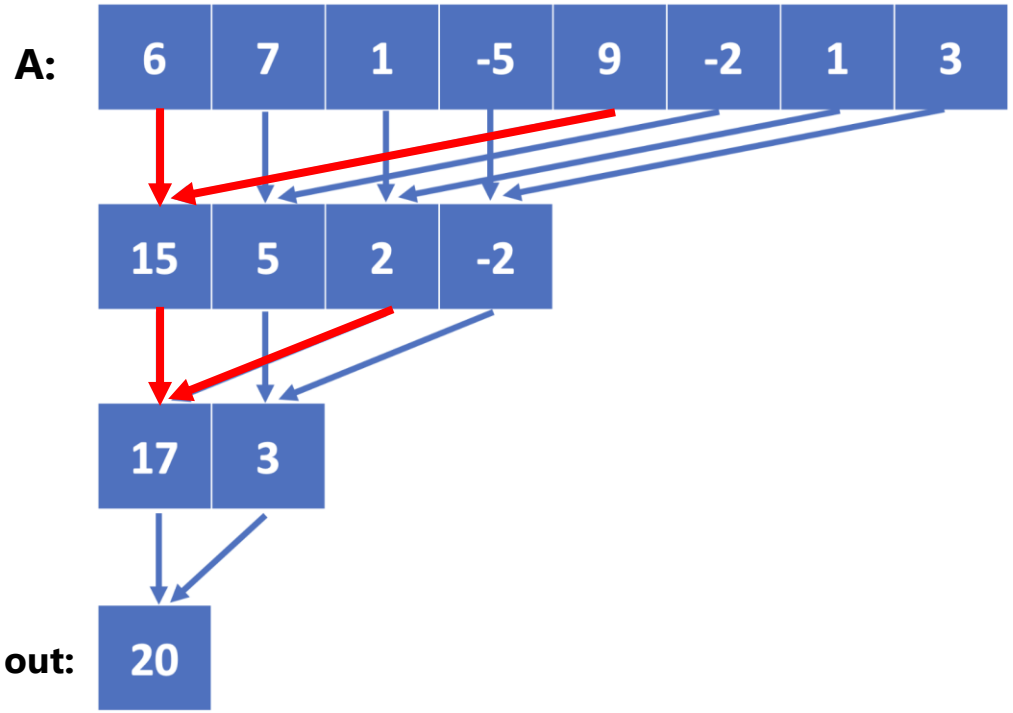
Parallel reduction

- Reductions: **aggregating elements** of an array **into a single value** through operations like summing, finding maximum or minimum, or performing logical operations

Parallel reduction

- Reductions: **aggregating elements** of an array **into a single value** through operations like summing, finding maximum or minimum, or performing logical operations

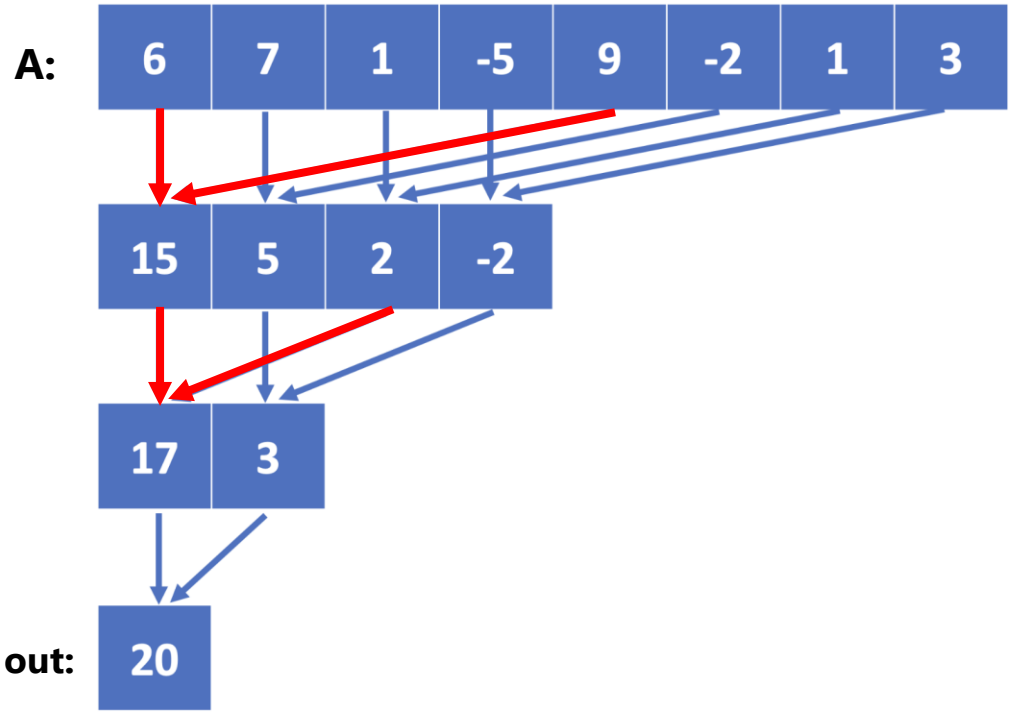
- Example: sum all elements of vector A



Parallel reduction

- Reductions: **aggregating elements** of an array **into a single value** through operations like summing, finding maximum or minimum, or performing logical operations

- Example: sum all N elements of vector A

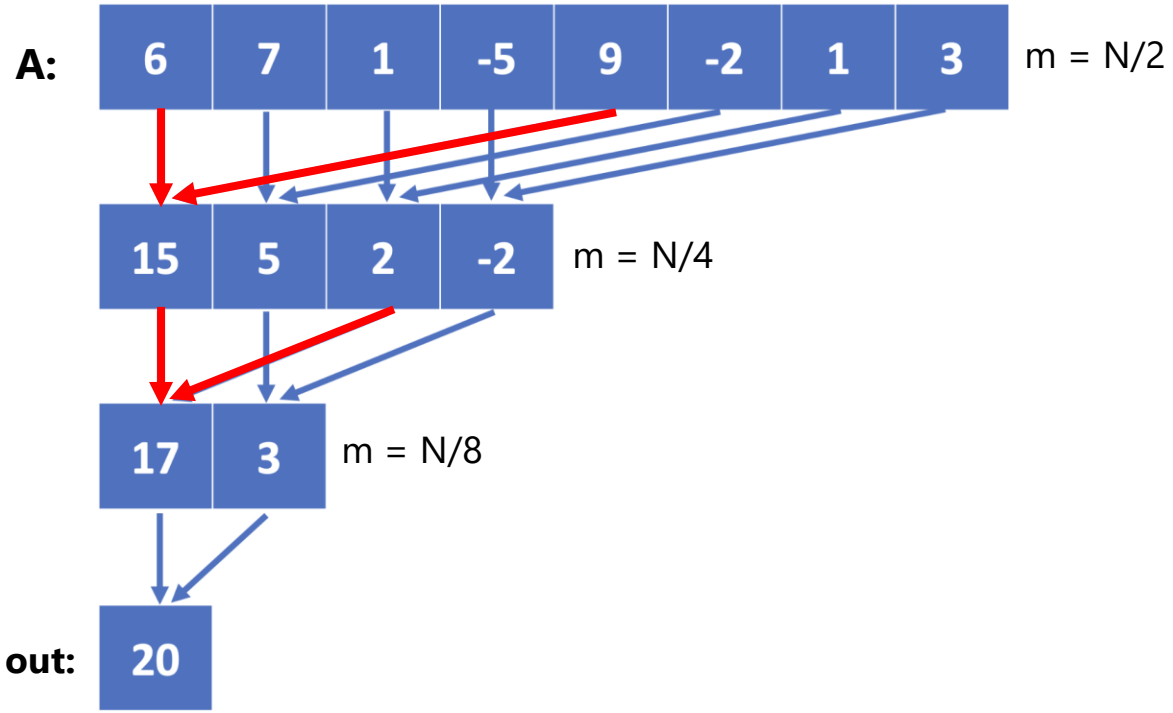


Solution 1: loop through elements
Problem: slow

Parallel reduction

- Reductions: **aggregating elements** of an array **into a single value** through operations like summing, finding maximum or minimum, or performing logical operations

- Example: sum all N elements of vector A



Solution 2: loop through kernel launches halving number of threads

```
for(int m = N/2; m>0; m /= 2) {
    int threads = std::min(256,m);
    int blocks = std::max(m/256,1);
    reduce<<<blocks, threads>>>(A, m);
}
```

where:

```
__global__ void reduce(float *a, int m) {
    int tid = blockDim.x*blockIdx.x+threadIdx.x;
    a[tid] += a[tid+m];
}
```

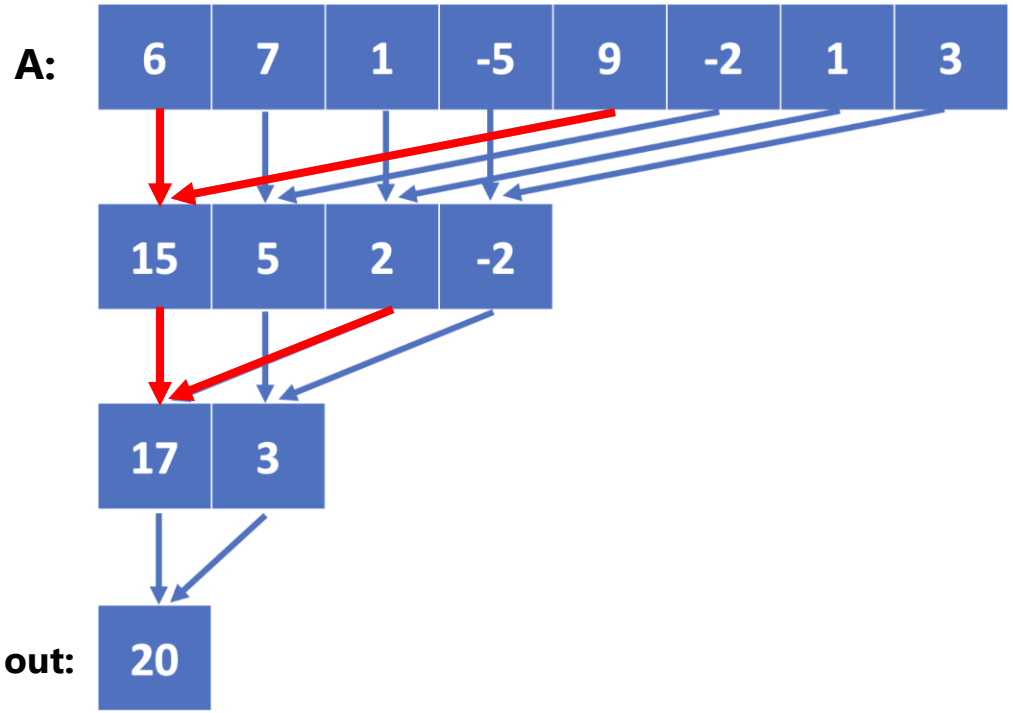
3 calls to global memory, doubled because of iterating

Problem: memory bound, N must be power of 2

Parallel reduction

- Reductions: **aggregating elements** of an array **into a single value** through operations like summing, finding maximum or minimum, or performing logical operations

- Example: sum all N elements of vector A



Solution 3: iterate inside the kernel, accumulate partial sums there; use grid stride inside kernel

```
__global__ void reduce (float *a, int N) {
int tid = blockDim.x*blockIdx.x+threadIdx.x;
float tsum = 0.0f;
for(int k=tid; k<N; k += gridDim.x*blockDim.x) tsum += a[k];
a[tid] = tsum;
}
...

reduce<<< blocks, threads >>>(A,N);
reduce<<< 1, threads >>>(A,blocks*threads);
reduce<<< 1,1 >>>(A, threads);
```

Can we do better? Let's make threads cooperate

Parallel reduction

- Blocks will sum their individual accumulated totals using shared memory

```
__global__ void reduce(float * out, float * a, int N) {
extern __shared__ float tsum[];
int id = threadIdx.x;
int tid = blockDim.x*blockIdx.x+threadIdx.x;
int stride = gridDim.x*blockDim.x;
tsum[id] = 0.0;
for(int k=tid;k<N;k+=stride) tsum[id] += a[k];
__syncthreads();

if(id<256 && id+256 < blockDim.x) tsum[id] += tsum[id+256]; __syncthreads();
if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
if(id< 8) tsum[id] += tsum[id+ 8]; __syncwarp();
if(id< 4) tsum[id] += tsum[id+ 4]; __syncwarp();
if(id< 2) tsum[id] += tsum[id+ 2]; __syncwarp();
if(id==0) out[blockIdx.x] = tsum[0]+tsum[1];
}
```

To save all partial sums in the corresponding part of shared memory

Parallel reduction

- Blocks will sum their individual accumulated totals using shared memory

```
__global__ void reduce(float * out, float * a, int N) {
extern __shared__ float tsum[];
int id = threadIdx.x;
int tid = blockDim.x*blockIdx.x+threadIdx.x;
int stride = gridDim.x*blockDim.x;
tsum[id] = 0.0;
for(int k=tid;k<N;k+=stride) tsum[id] += a[k];
__syncthreads();

if(id<256 && id+256 < blockDim.x) tsum[id] += tsum[id+256]; __syncthreads();
if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
if(id< 8) tsum[id] += tsum[id+ 8]; __syncwarp();
if(id< 4) tsum[id] += tsum[id+ 4]; __syncwarp();
if(id< 2) tsum[id] += tsum[id+ 2]; __syncwarp();
if(id==0) out[blockIdx.x] = tsum[0]+tsum[1];
}
```

Actually save all partial sums in the corresponding part of shared memory

Parallel reduction

- Blocks will sum their individual accumulated totals using shared memory

```
__global__ void reduce(float * out, float * a, int N) {
extern __shared__ float tsum[];
int id = threadIdx.x;
int tid = blockDim.x*blockIdx.x+threadIdx.x;
int stride = gridDim.x*blockDim.x;
tsum[id] = 0.0;
for(int k=tid;k<N;k+=stride) tsum[id] += a[k];
__syncthreads();

if(id<256 && id+256 < blockDim.x) tsum[id] += tsum[id+256]; __syncthreads();
if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
if(id< 8) tsum[id] += tsum[id+ 8]; __syncwarp();
if(id< 4) tsum[id] += tsum[id+ 4]; __syncwarp();
if(id< 2) tsum[id] += tsum[id+ 2]; __syncwarp();
if(id==0) out[blockIdx.x] = tsum[0]+tsum[1];
}
```

Do the reduction INSIDE shared memory (threads = 512 assumed)

Parallel reduction

- Blocks will sum their individual accumulated totals using shared memory

```
__global__ void reduce(float * out, float * a, int N) {
extern __shared__ float tsum[];
int id = threadIdx.x;
int tid = blockDim.x*blockIdx.x+threadIdx.x;
int stride = gridDim.x*blockDim.x;
tsum[id] = 0.0;
for(int k=tid;k<N;k+=stride) tsum[id] += a[k];
__syncthreads();

if(id<256 && id+256 < blockDim.x) tsum[id] += tsum[id+256]; __syncthreads();
if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
if(id< 8) tsum[id] += tsum[id+ 8]; __syncwarp();
if(id< 4) tsum[id] += tsum[id+ 4]; __syncwarp();
if(id< 2) tsum[id] += tsum[id+ 2]; __syncwarp();
if(id==0) out[blockIdx.x] = tsum[0]+tsum[1];
}
```

Still reduction in shared memory, but making use of warps (for AMD we would start _syncwarp the line above)

Parallel reduction

- Blocks will sum their individual accumulated totals using shared memory

```
__global__ void reduce(float * out, float * a, int N) {
extern __shared__ float tsum[];
int id = threadIdx.x;
int tid = blockDim.x*blockIdx.x+threadIdx.x;
int stride = gridDim.x*blockDim.x;
tsum[id] = 0.0;
for(int k=tid;k<N;k+=stride) tsum[id] += a[k];
__syncthreads();

if(id<256 && id+256 < blockDim.x) tsum[id] += tsum[id+256]; __syncthreads();
if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
if(id< 8) tsum[id] += tsum[id+ 8]; __syncwarp();
if(id< 4) tsum[id] += tsum[id+ 4]; __syncwarp();
if(id< 2) tsum[id] += tsum[id+ 2]; __syncwarp();
if(id==0) out[blockIdx.x] = tsum[0]+tsum[1];
}
```

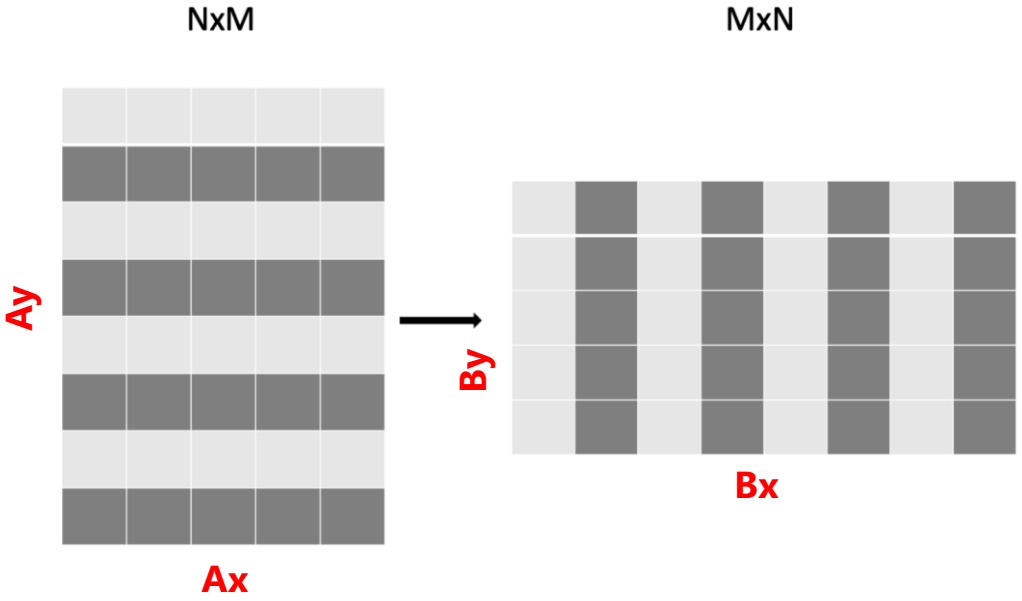
Further improvements:

- ✓ Make threads a template
- ✓ Use cooperative groups
- ✓ Use warp reduction
- ✓ Halve the number of blocks and “double load”

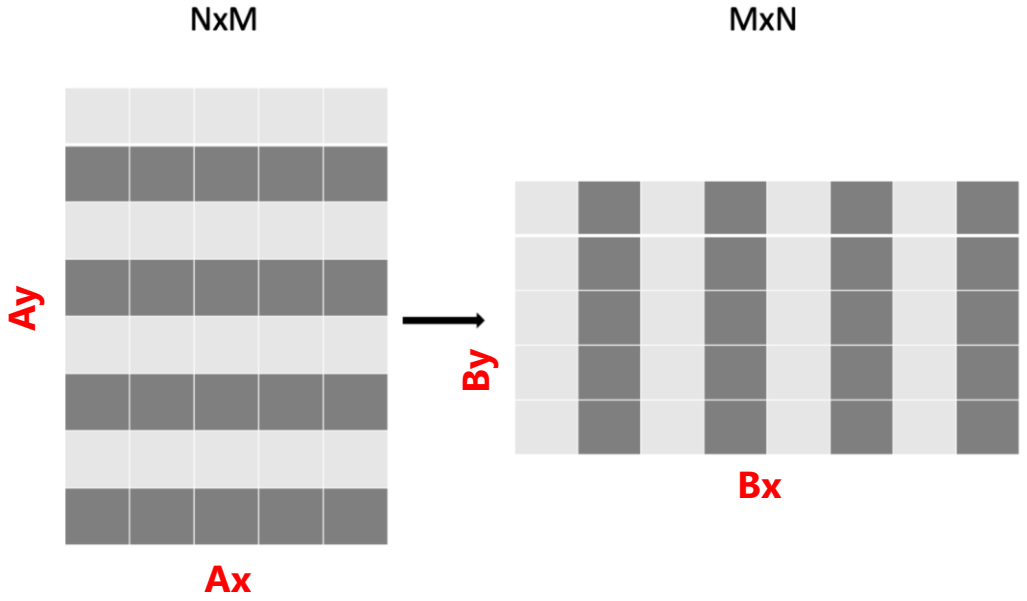
Some good sources:

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
https://github.com/NVIDIA/cuda-samples/blob/master/Samples/2_Concepts_and_Techniques/reduction/
https://developer.nvidia.com/blog/faster-parallel-reductions-kepler

Matrix transpose



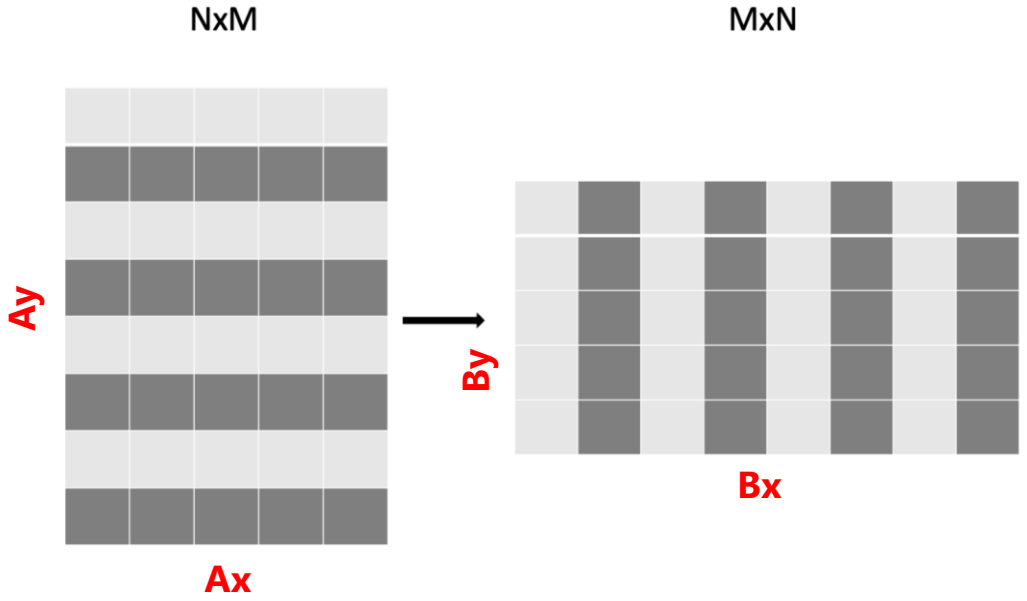
Matrix transpose



- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

Matrix transpose



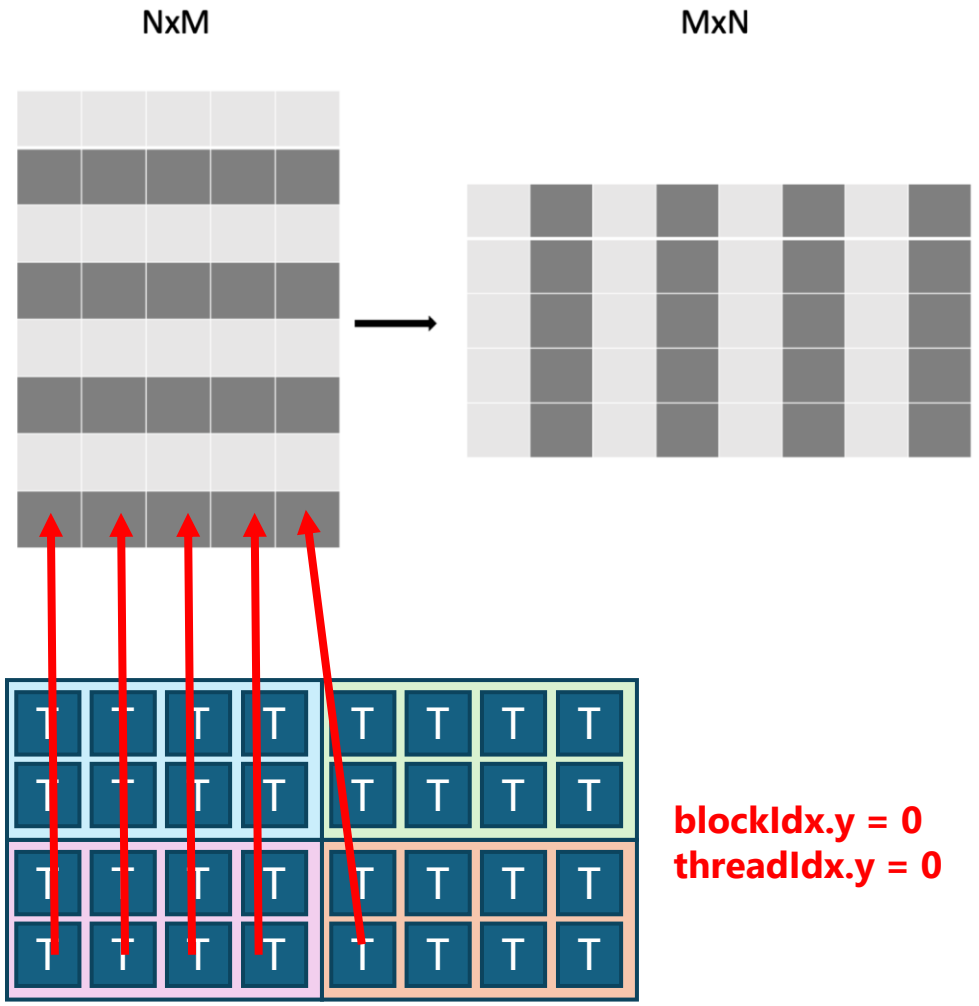
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

Matrix transpose



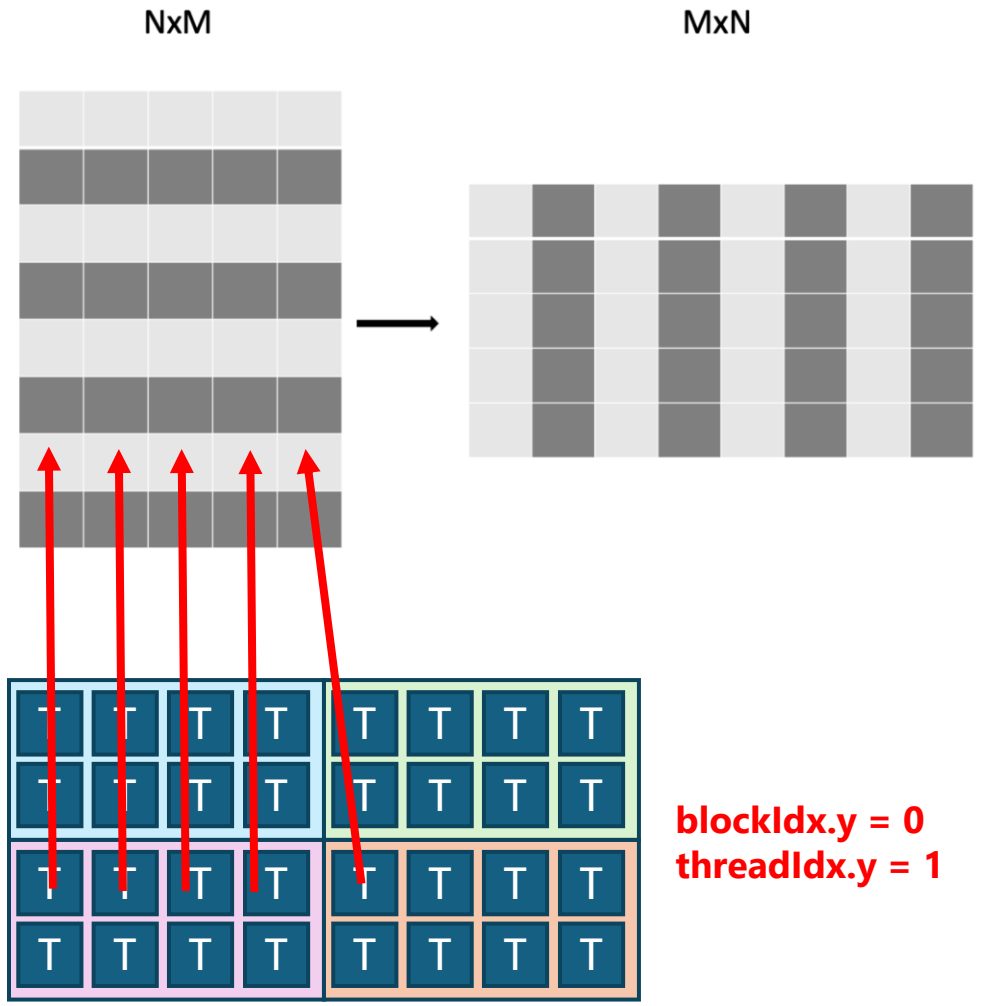
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

Matrix transpose



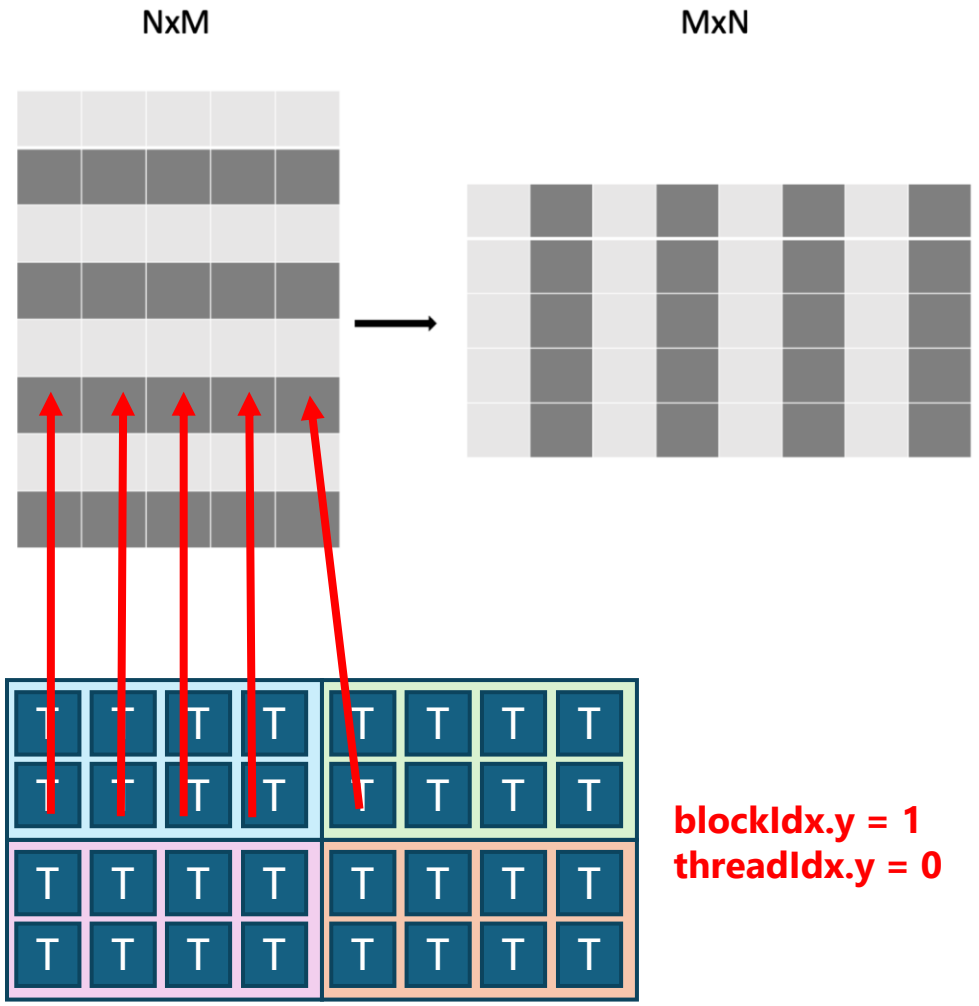
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

Matrix transpose



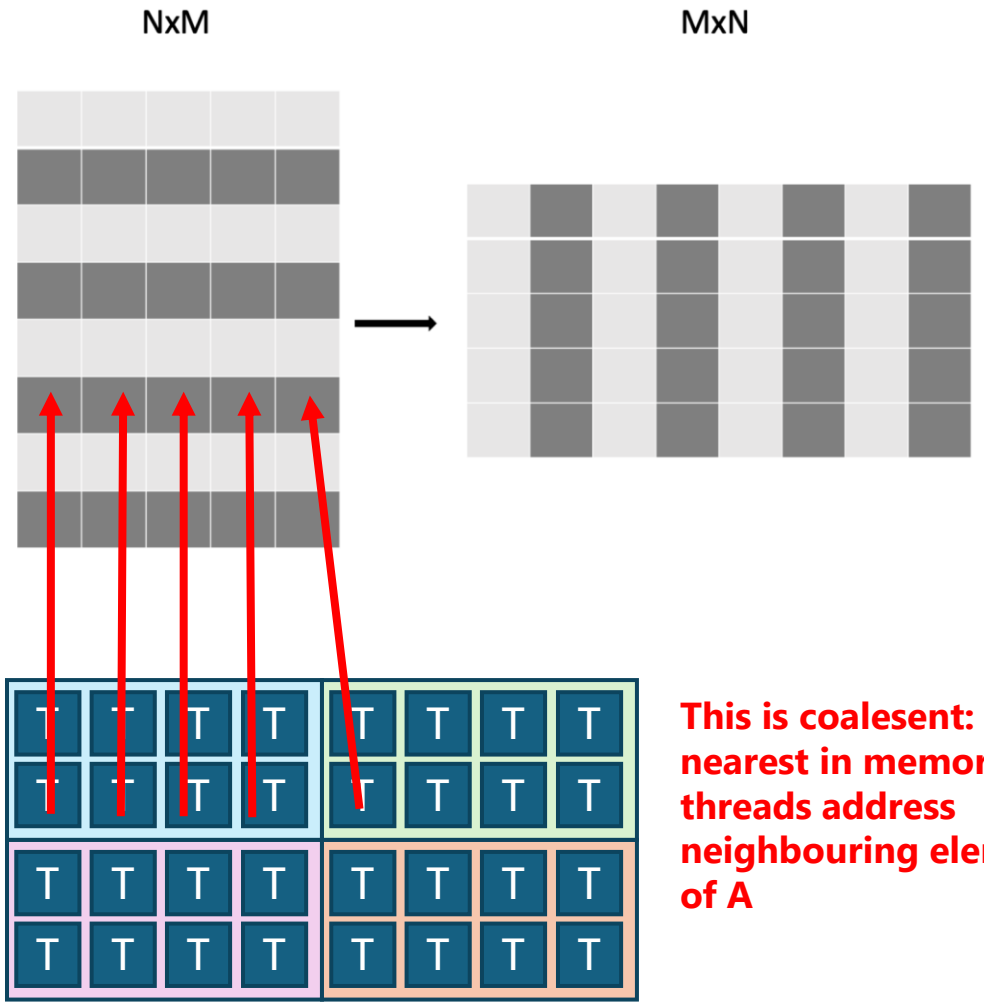
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

Matrix transpose



This is coalesent: nearest in memory threads address neighbouring elements of A

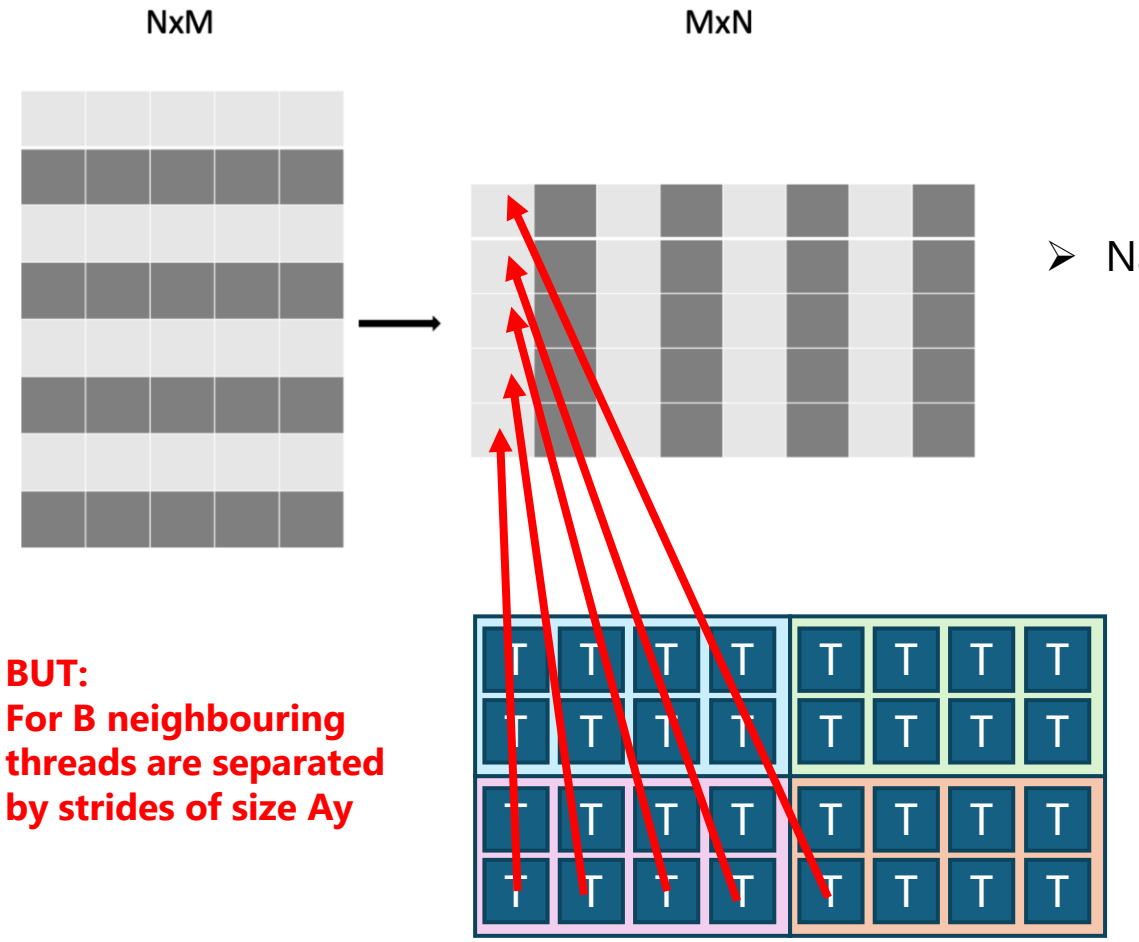
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

Matrix transpose



BUT:
For B neighbouring threads are separated by strides of size Ay

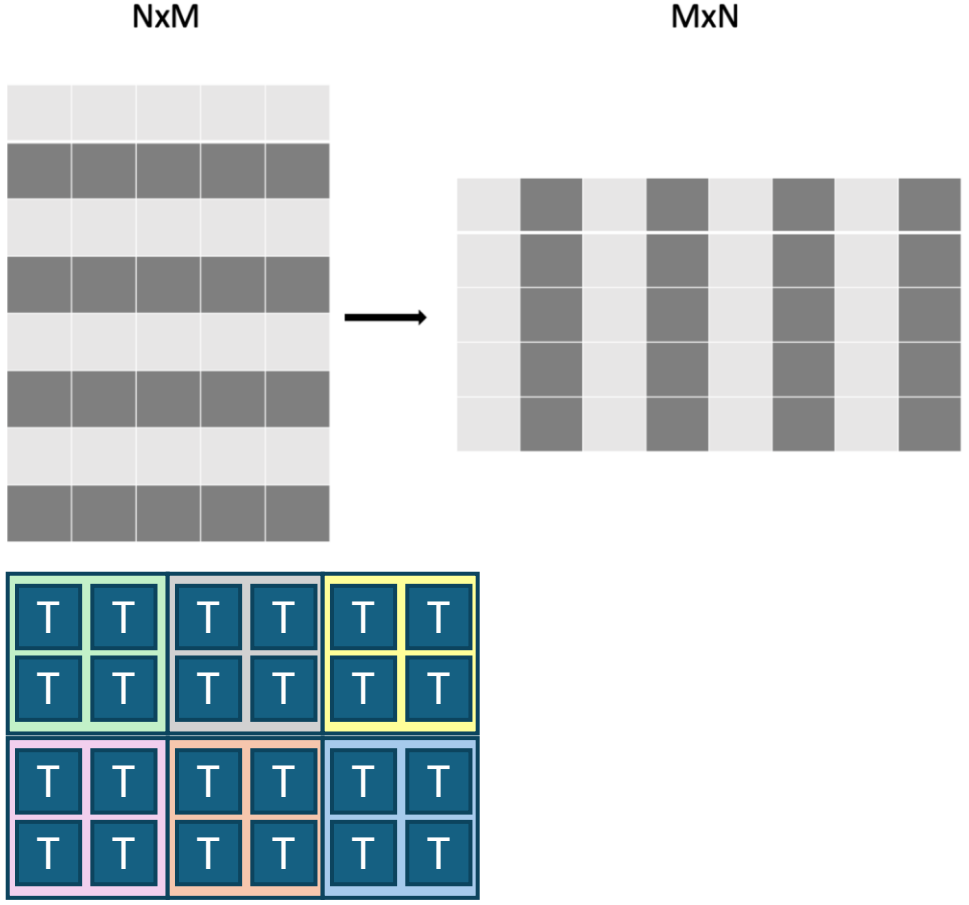
- CPU:

```
//Bx = Ay; By = Ax;
for(int iy=0; iy<Ay; iy++){
    for(int ix=0; ix<Ax; ix++){
        B[ix*Ay+iy] = A[iy*Ax+ix];
    }
}
```

- Naïve version on GPU:

```
__global__ void gpu_transpose(const float * A, float * B, int Ax, int Ay){
int ix = blockIdx.x*blockDim.x + threadIdx.x;
int iy = blockIdx.y*blockDim.y + threadIdx.y;
if(iy >= Ay || ix >= Ax) return;

B[ix * Ay + iy] = A[iy * Ax + ix];
}
...
dim3 threads ={threadx, thready, 1};
dim3 blocks ={(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose<<<blocks,threads>>>(A, B, Ax, Ay);

//Ax = 5; Ay = 8;
//threadx = 4; thready = 2;
//blockx = 2; blocky =  4;
```

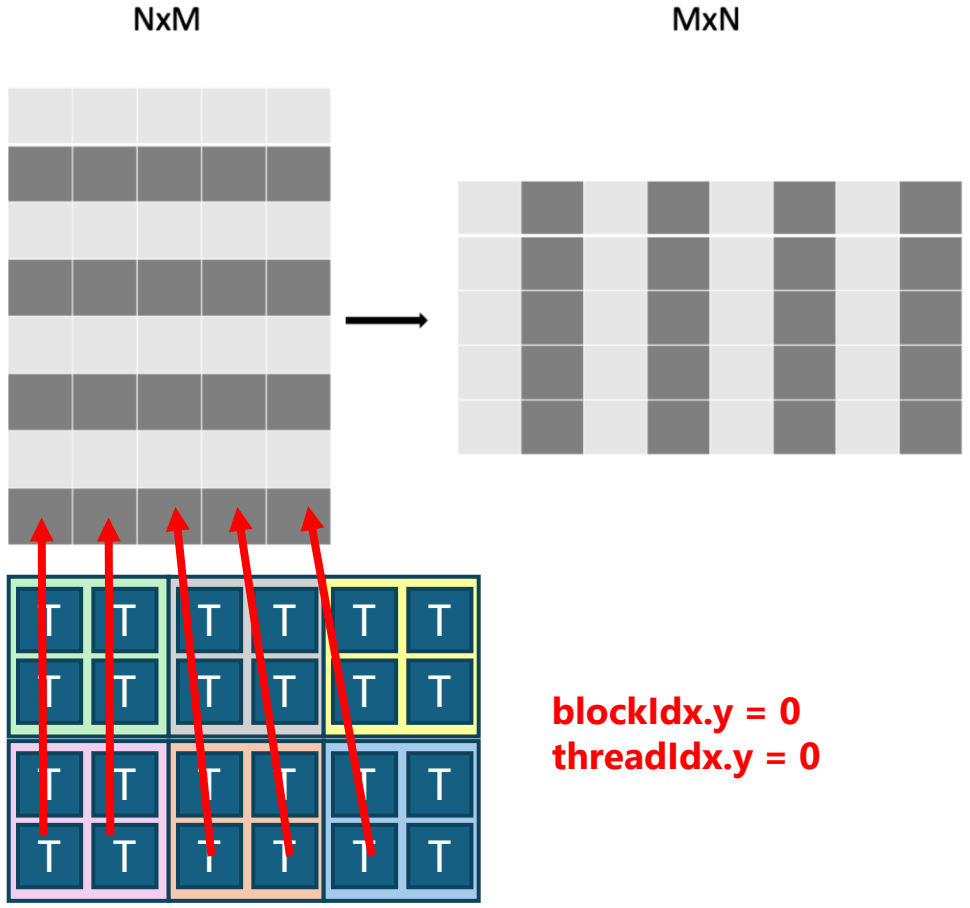
Matrix transpose



- Improved version on GPU – using shared memory:

```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}

...

dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

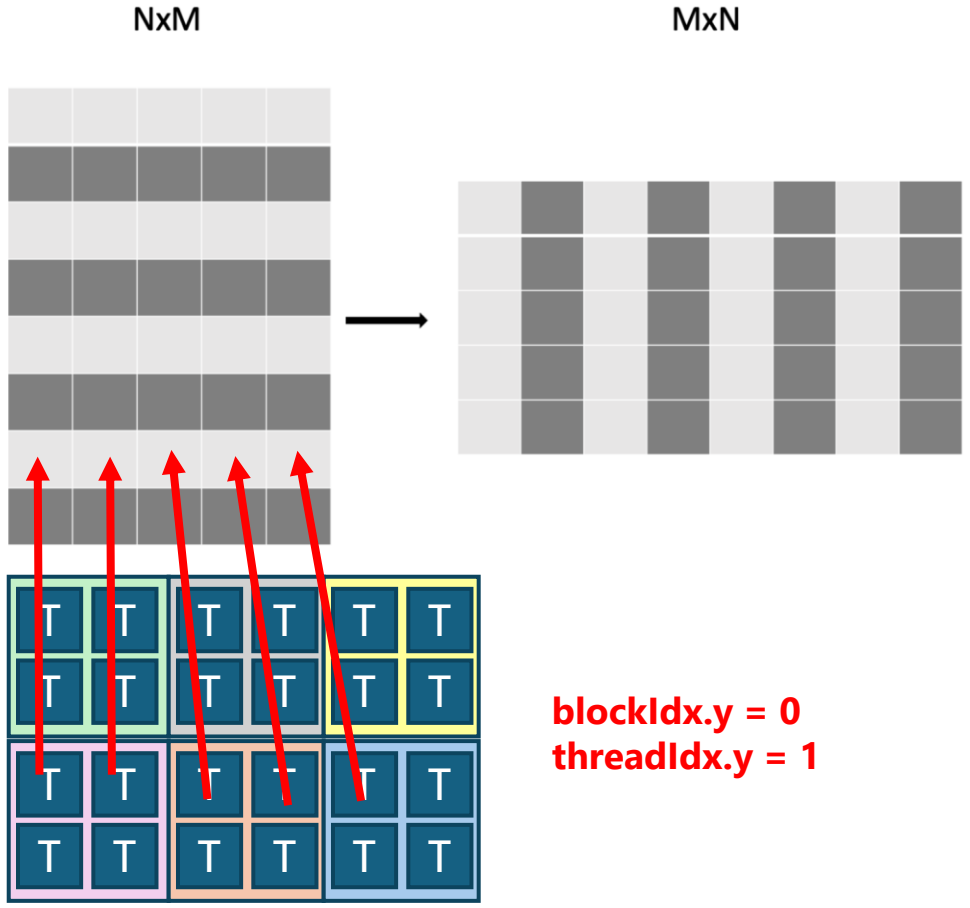
Matrix transpose



- Improved version on GPU – using shared memory:

```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}

...

dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

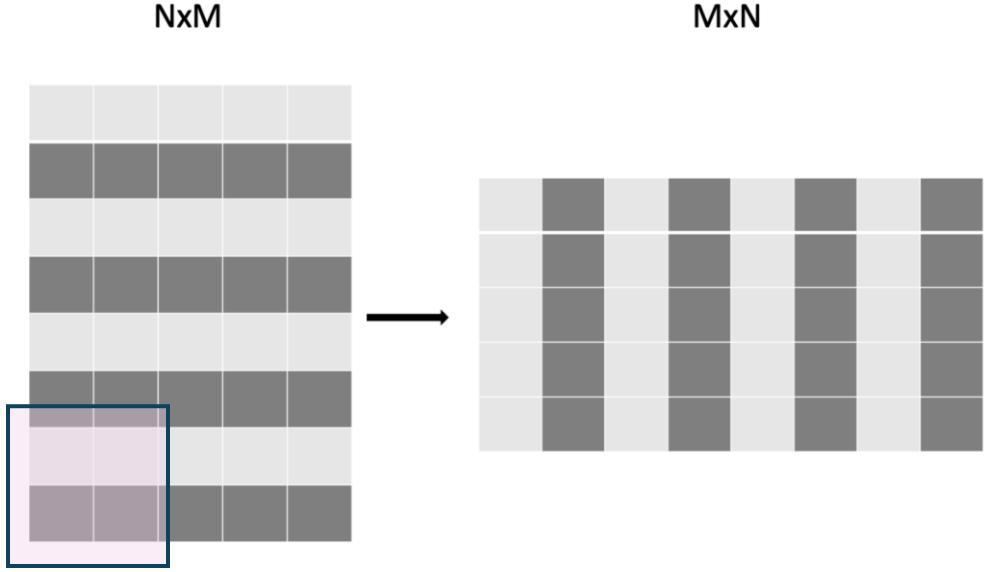
Matrix transpose

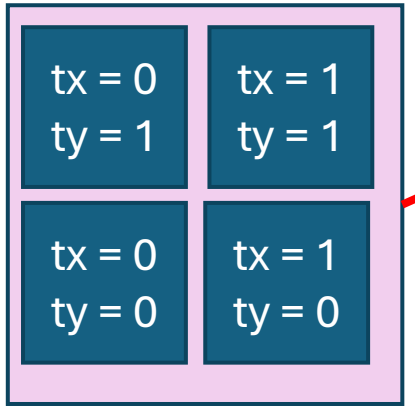


- Improved version on GPU – using shared memory:

```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}

...

dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

Matrix transpose



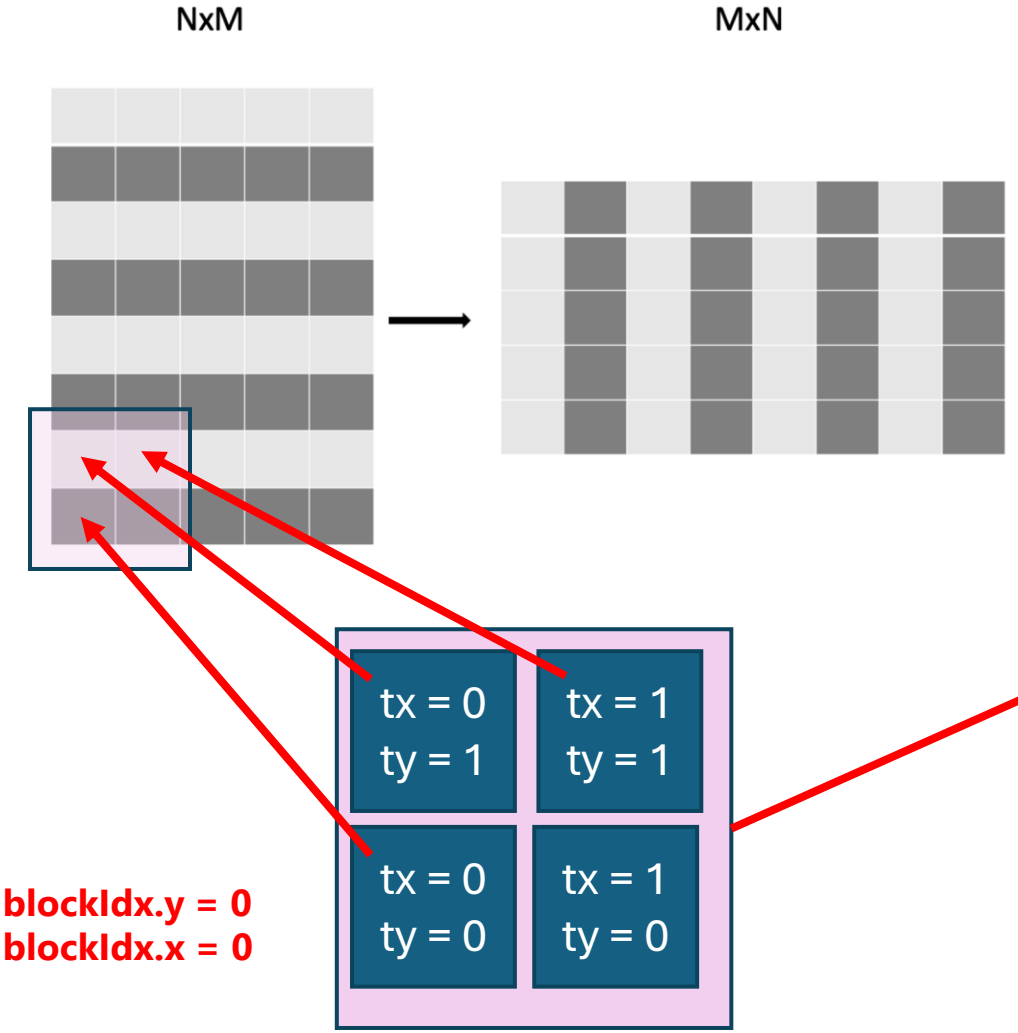


blockIdx.y = 0
blockIdx.x = 0

- Improved version on GPU – using shared memory:

```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}

...

dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

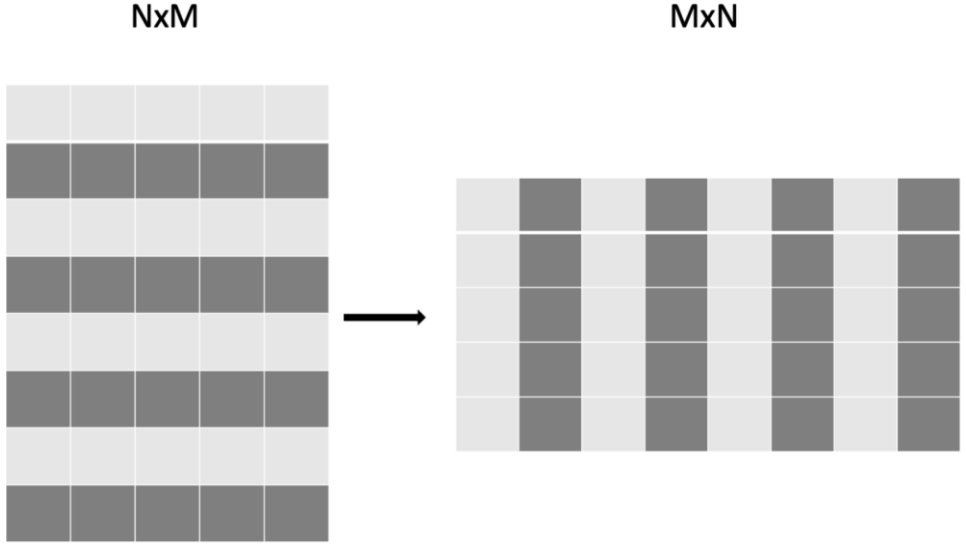
Matrix transpose



- Improved version on GPU – using shared memory:

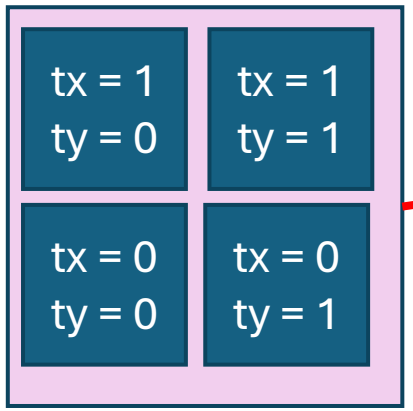
```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}

...

dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

Matrix transpose



- Improved version on GPU – using shared memory:

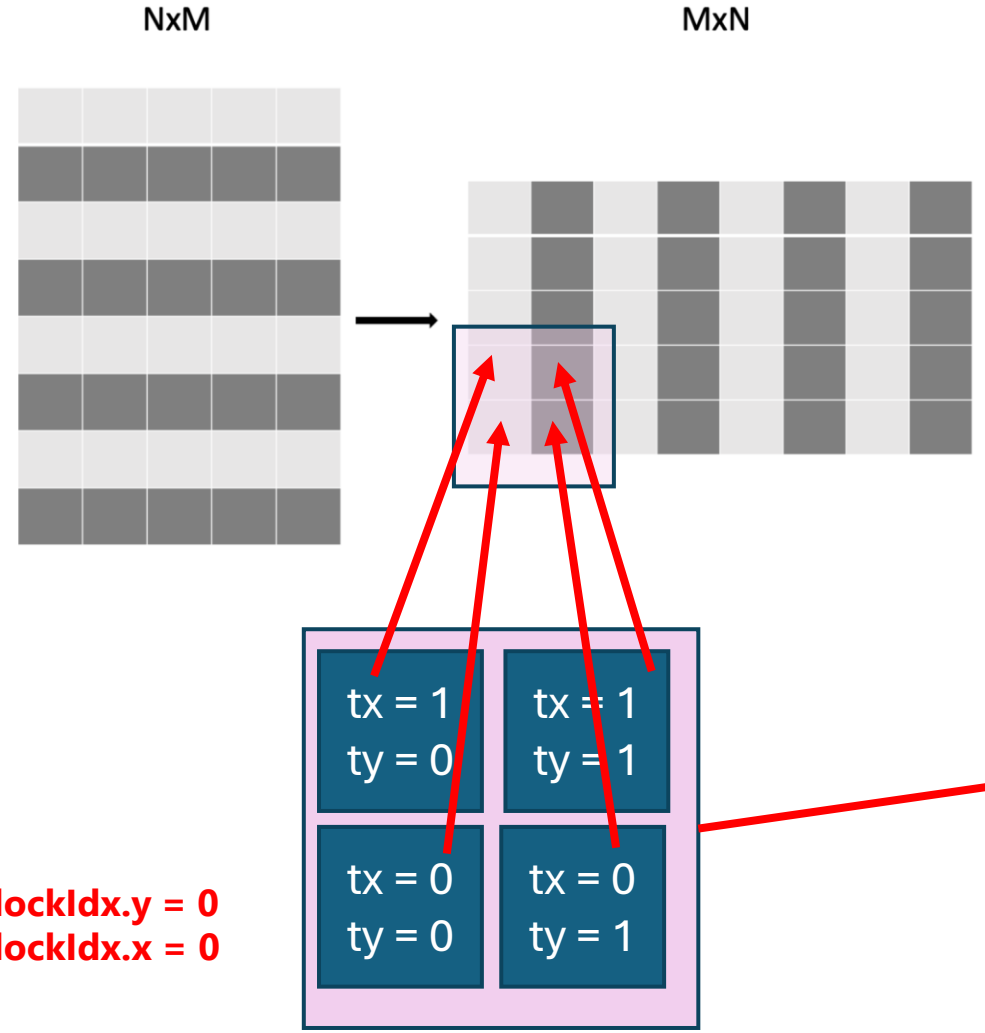
```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}
```



blockIdx.y = 0
blockIdx.x = 0

```
dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

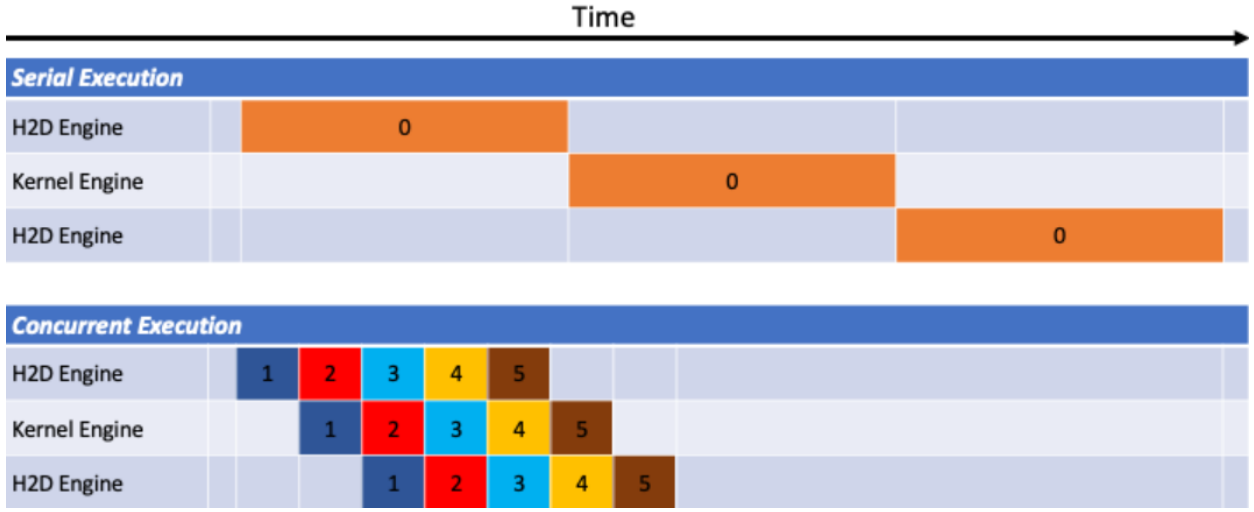
Matrix transpose



- Improved version on GPU – using shared memory:

```
template<int tile_dim>
__global__ void gpu_transpose_SM(const float* A, float* B, int Ax, int Ay){
    __shared__ float tile[tile_dim][tile_dim];

    int x = blockIdx.x * tile_dim + threadIdx.x;   // column in A
    int y = blockIdx.y * tile_dim + threadIdx.y;   // row in A
    int tx = threadIdx.x;   // column in tile
    int ty = threadIdx.y;   // row in tile

    if (x < Ax && y < Ay){ tile[ty][tx] = A[y * Ax + x];}
    __syncthreads();

    int bx = blockIdx.y * TILE_DIM + threadIdx.x;  //column in B
    int by = blockIdx.x * TILE_DIM + threadIdx.y;  //row in B

    if (bx < Ay && by < Ax){ B[by * Ay + bx] = tile[tx][ty];}
}


dim3 thread = {tile_dim, tile_dim, 1};
dim3 blocks = {(Ax+threads.x-1)/threads.x, (Ay+threads.y-1)/threads.y, 1};
gpu_transpose_S<tile_dim><<<blocks,threads>>>(A, B, Ax, Ay);
```

Streams



- Allow to split data into pieces that can be processed and copied **simultaneously**
- Set with “stream” parameter inside kernel call
- Have to be accompanied with *hipMemcpyAsync*
- Default stream is 0

Useful for:

- Large “frame by frame” data processing
- Simultaneously running memory-bound and computationally-bound kernels

Data transfer options

- Zero-copy memory: pinned memory on CPU can be modified and accessed by GPU directly; device pointer is still needed, referring to the same memory
- Unified memory: same pointer is used for CPU and GPU, memory transfers are “hidden”

✓ Extra care with synchronization: hipDeviceSynchronize and implicit memory copying

✓ UM can be slow on windows

✓ Zero-copy becomes slow upon repeated GPU read

… and more

- Cooperative groups: convenient functionality to get bid, tid etc; allows grouping threads in tiles
- Thrust library: Adds std::vector to GPU; has some useful features like complex numbers
- Multi-GPU: cudaSetDevice, cudaGetDeviceCount etc.; Unified Virtual Addressing – treats memory of devices as a unit
- Warp-level programming: Parallel reduction as an example; speed-up
- Atomic operations: Queue threads' requests to modify memory simultaneously

Look out for

- Coalescent memory access
- Avoid thread divergence within warp – will make them run in sets
- Avoid deadlocks – thread synchronisations in *if* statements
- Don’t allow threads to read/write in the same place in memory – race conditions
- Check if you are trying to access out of bound memory
- Use error checks:

```
#define ASSERT_CUDA_SUCCESS(cudaCall)                                                   \
   {                                                                                    \
      cudaError_t error = cudaCall;                                                     \
      if(error != cudaSuccess) {                                                        \
         std::fprintf(                                                                  \
             stderr, "Error on line %i, file %s: %s\n", __LINE__, __FILE__, cudaGetErrorString(error)); \
         std::exit(EXIT_FAILURE);                                                       \
      }                                                                                 \
   }
```

To take home message

- Learn about the hardware
- Keep track of memory allocations and flow
- Track data/threads indexing correspondence and coalescence
- Don’t trust ChatGPT too much
- If in doubt, opt for an easier solution and make GPU communicate back to you

https://github.com/ENCCS/gpu-programming