High Performance Computing

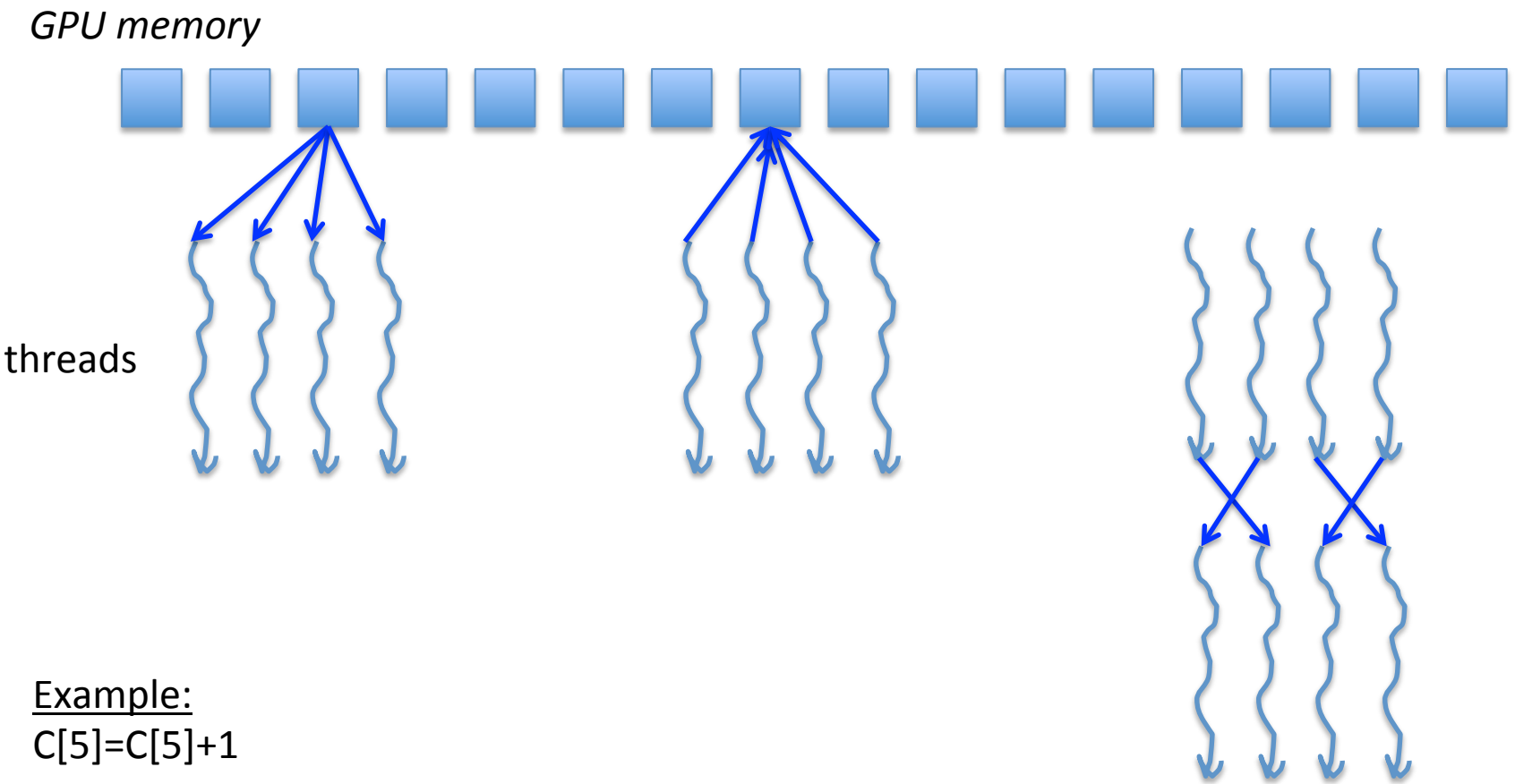
Term 4 2018/2019

Lecture 8

Last time we started with CUDA

CUDA: multiple threads working together

Communication is key!

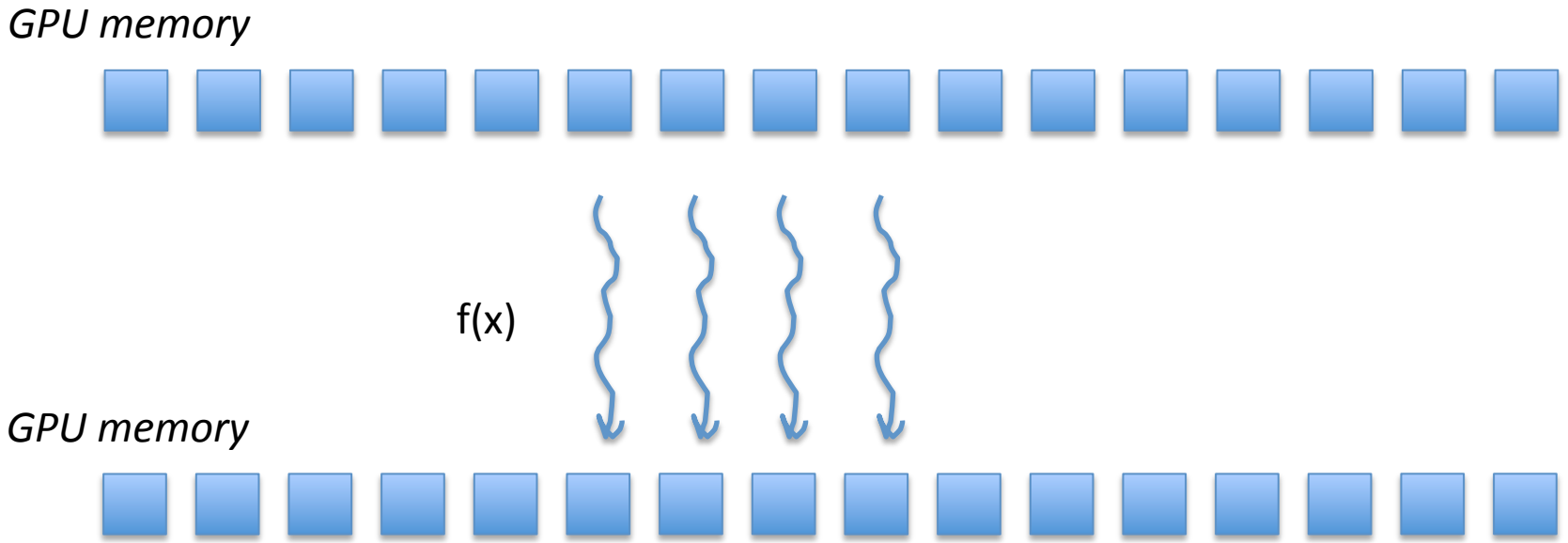


Example:
C[5]=C[5]+1

Parallel communication patterns

We have already learned and used parallel communication patterns in MPI.

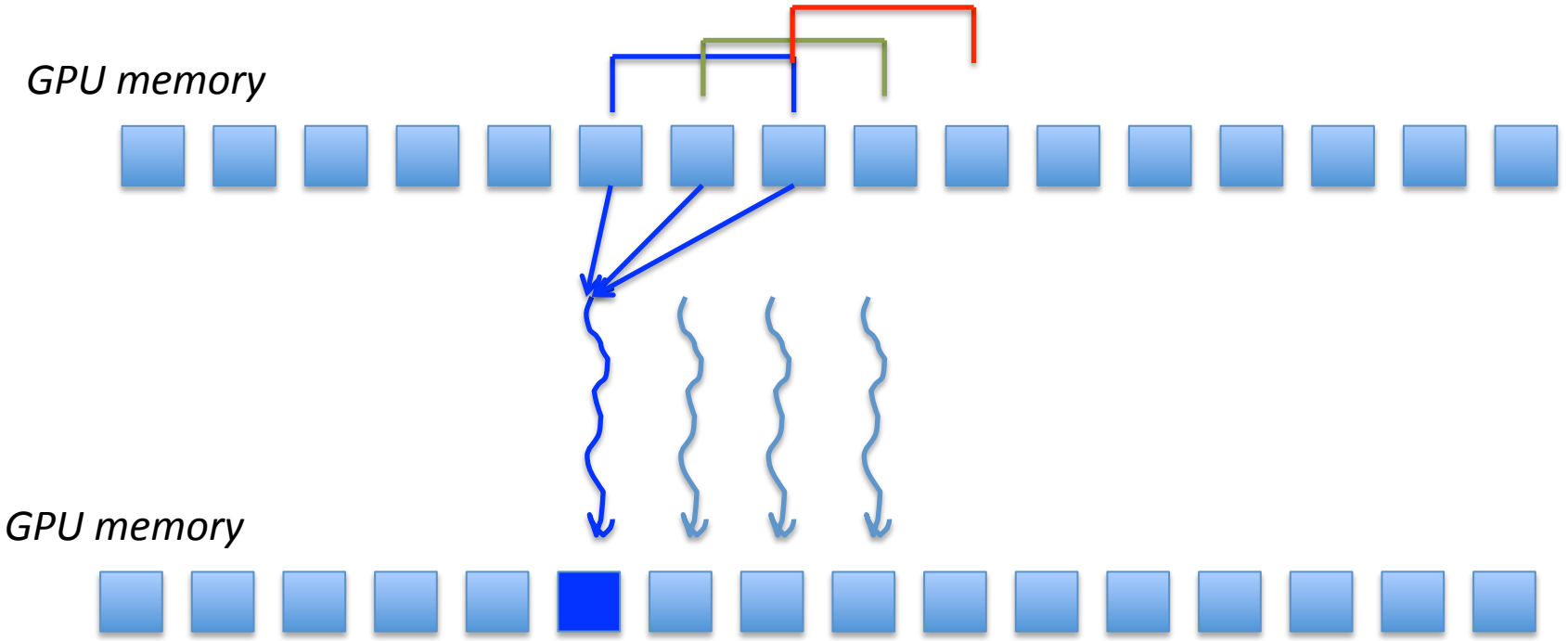
MAP



GPUs are really good at MAP

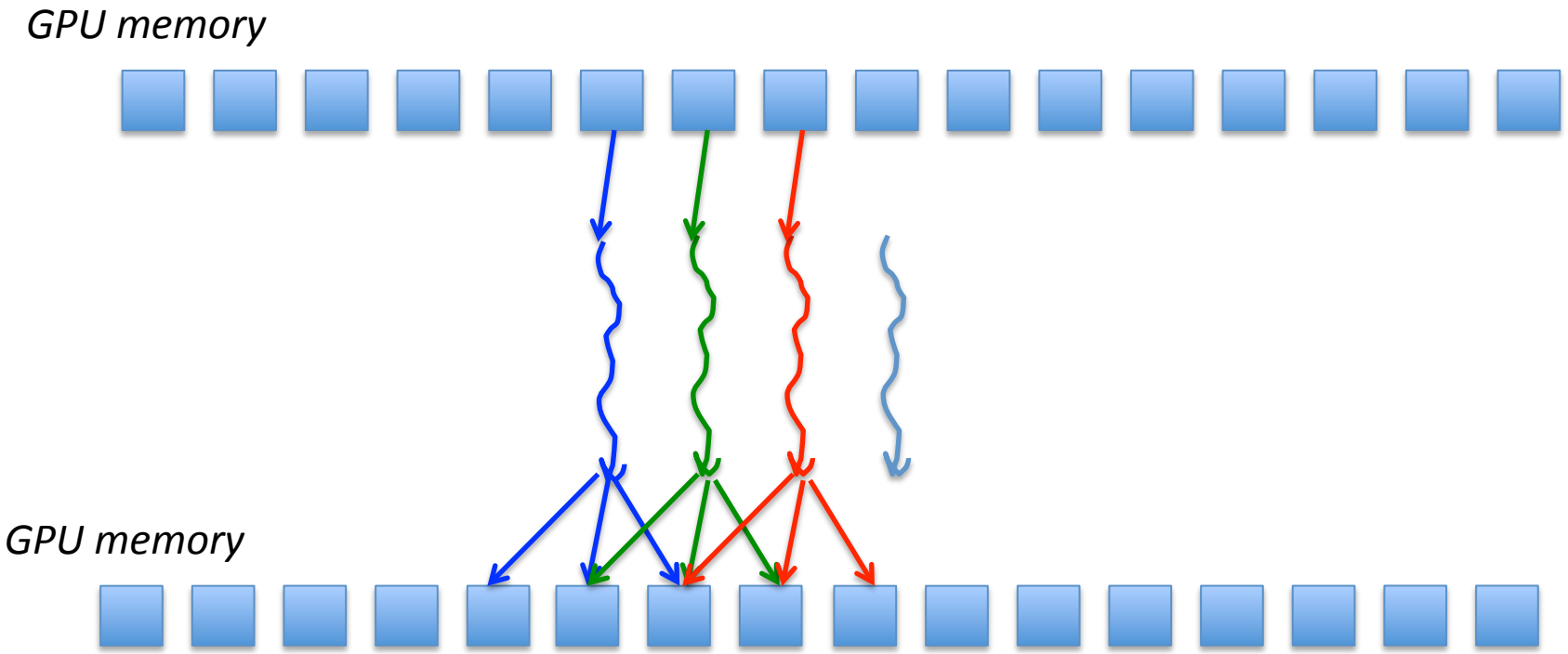
Parallel communication patterns

GATHER



Parallel communication patterns

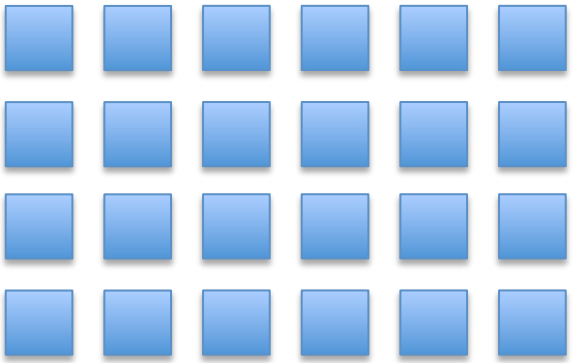
SCATTER

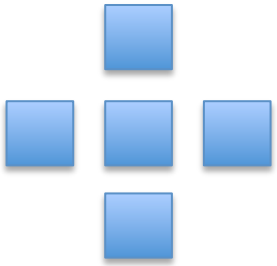


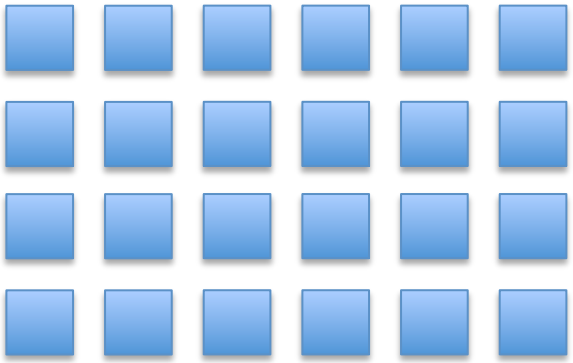
Parallel communication patterns

STENCIL

GPU memory



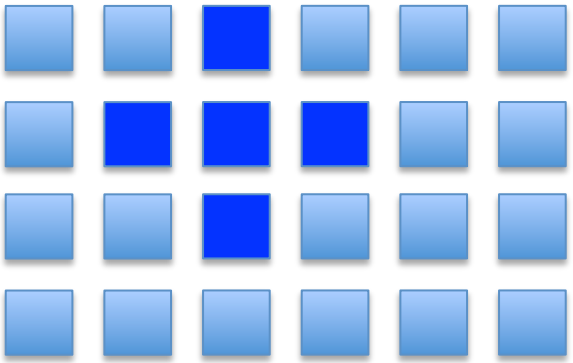


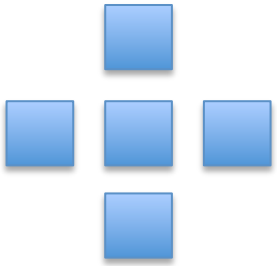


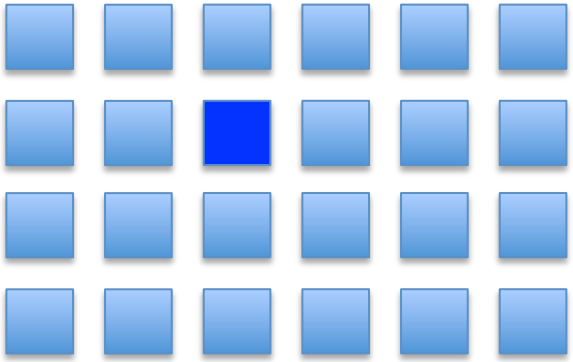
Parallel communication patterns

STENCIL

GPU memory



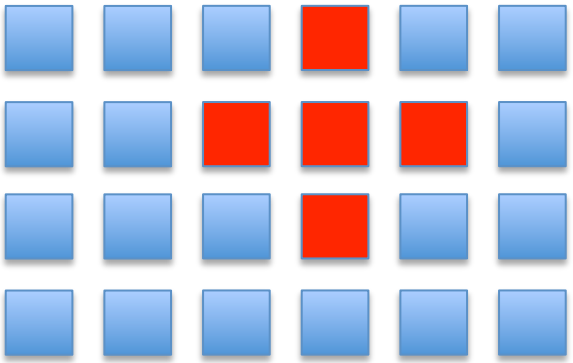


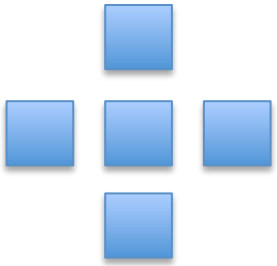


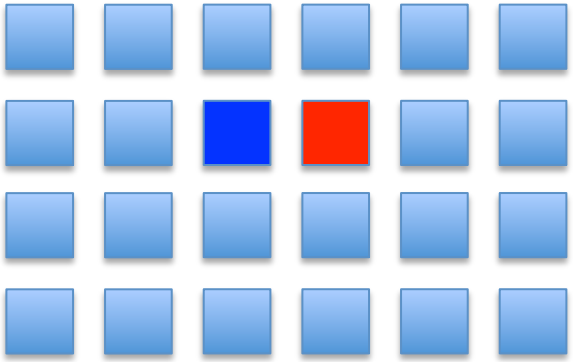
Parallel communication patterns

STENCIL

GPU memory





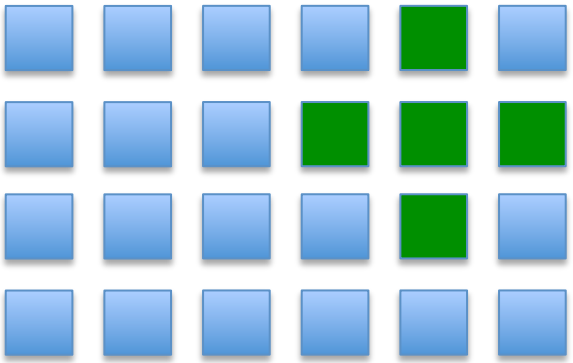


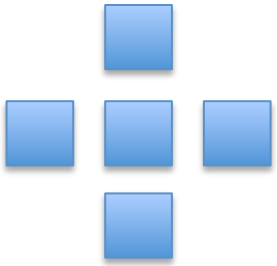
Parallel communication patterns

STENCIL

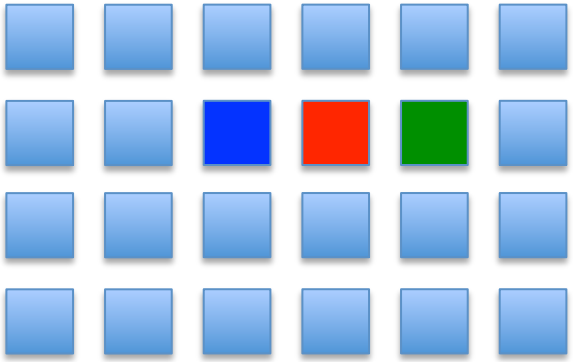
Data reuse!

GPU memory





2D von Neumann

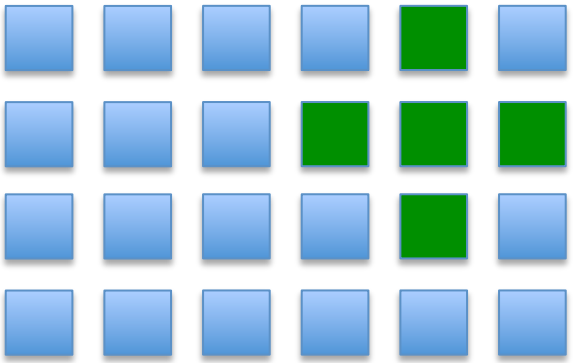


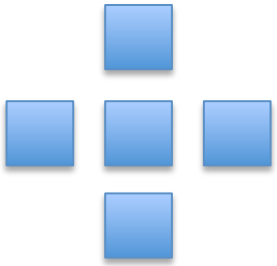
Parallel communication patterns

STENCIL

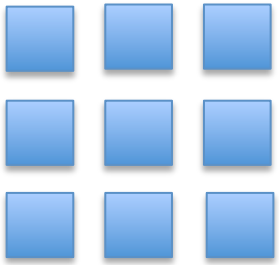
Data reuse!

GPU memory

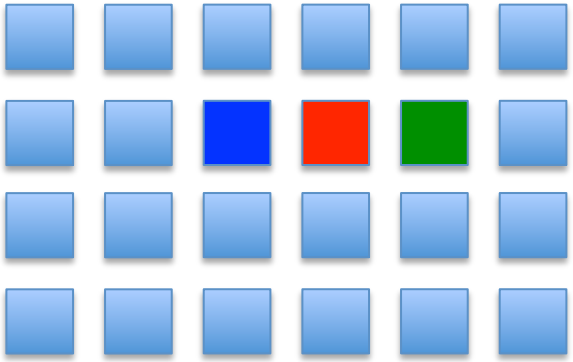




2D von Neumann

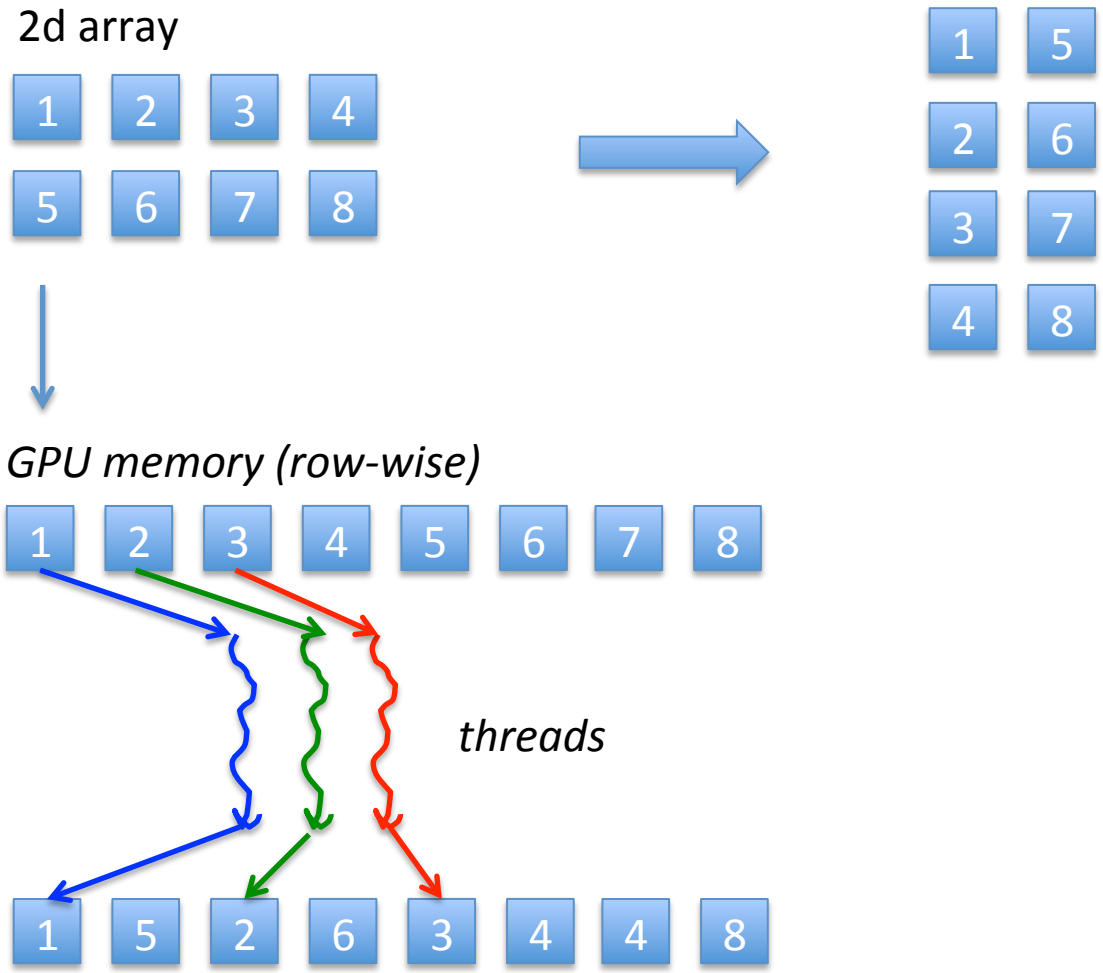


2D Moore



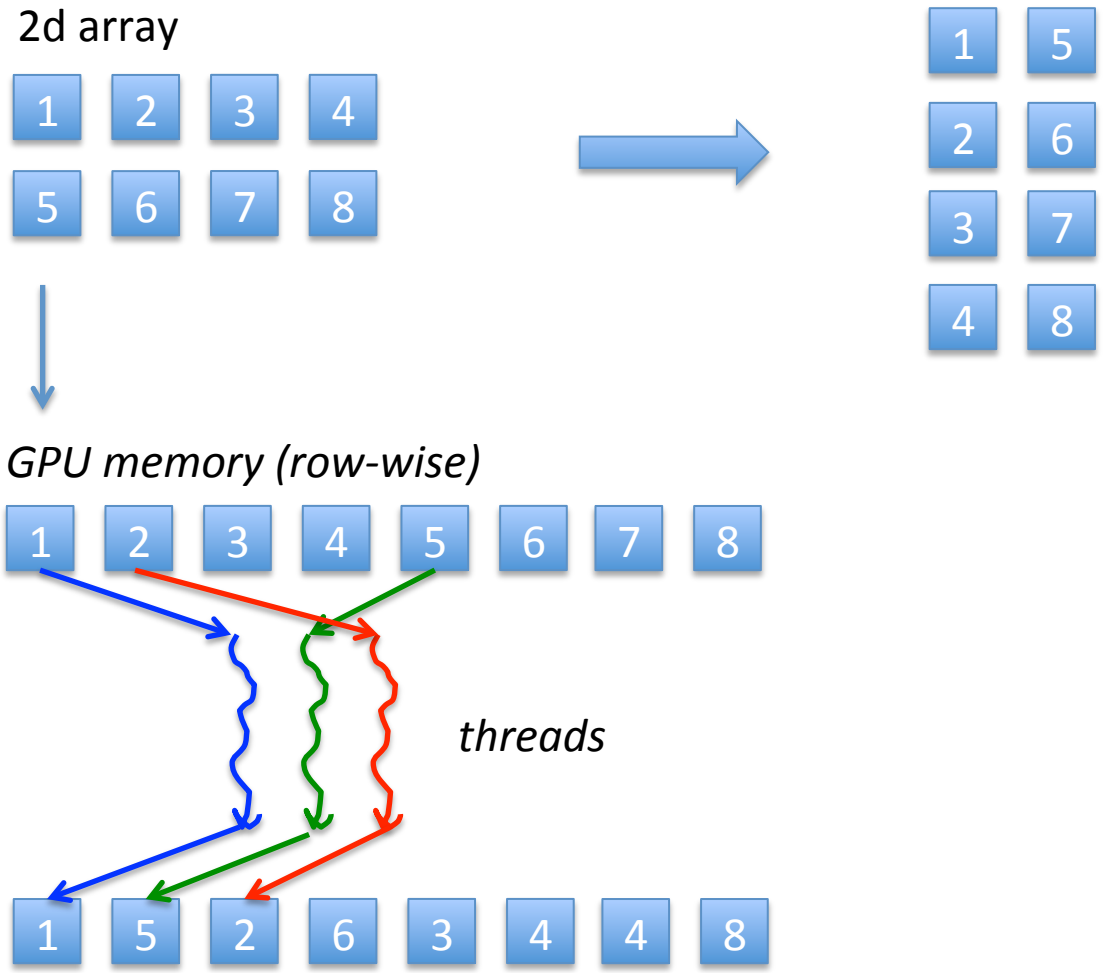
Parallel communication patterns

TRANSPOSE



Parallel communication patterns

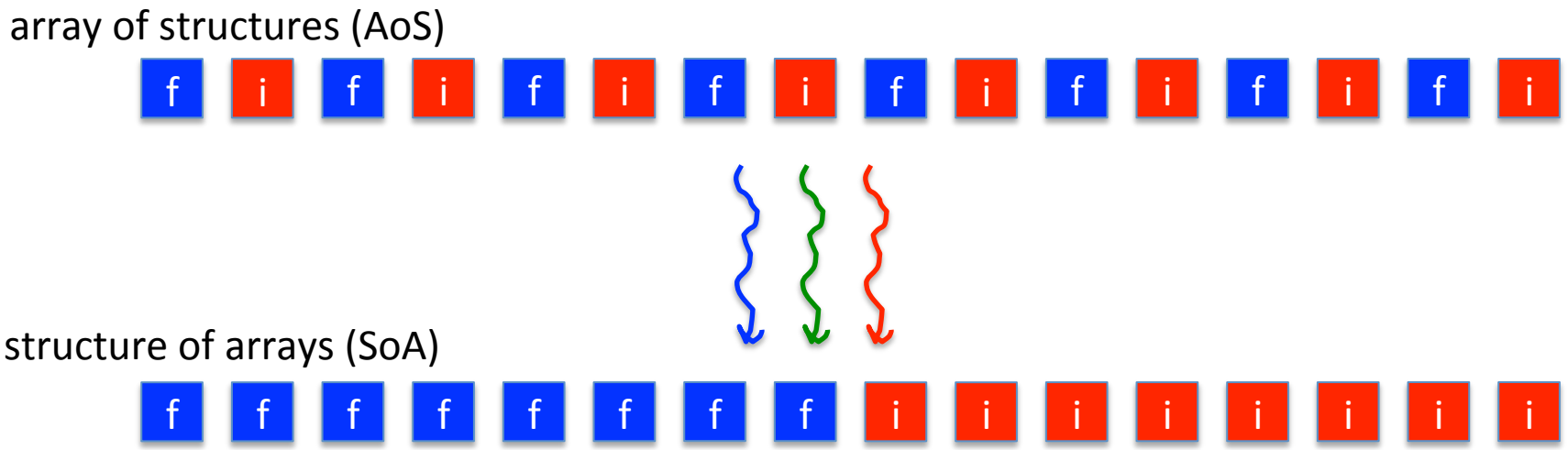
TRANSPOSE



Parallel communication patterns

TRANSPOSE

```
struct foo
{
    float f;
    int i;
}


foo array[1000];
```



Parallel communication patterns

```
float out[], in[];
int i = threadIdx.x;
int j = threadIdx.y;

const float pi = 3.1415;

out[i] = pi * in[i];

out[i + j*128] = in[j + i*128];

if (i % 2) {
    out[i-1] += pi * in[i]; out[i+1] += pi * in[i];

    out[i] = (in[i] + in[i-1] + in[i+1]) * pi / 3.0f;
}
```

?

?

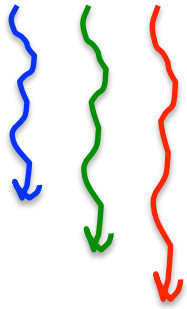
Parallel communication patterns

MAP **TRANSPOSE**	**one-to-one operations**
GATHER	**many-to-one**
SCATTER	**one-to-many**
STENCIL	**several-to-one**
REDUCE	**all-to-one**
SCAN/SORT	**all-to-all**

1. **How can threads efficiently access memory in concert?**
2. **How to exploit data reuse?**
3. **How can threads communicate partial results?**

Summary of GPU programming model

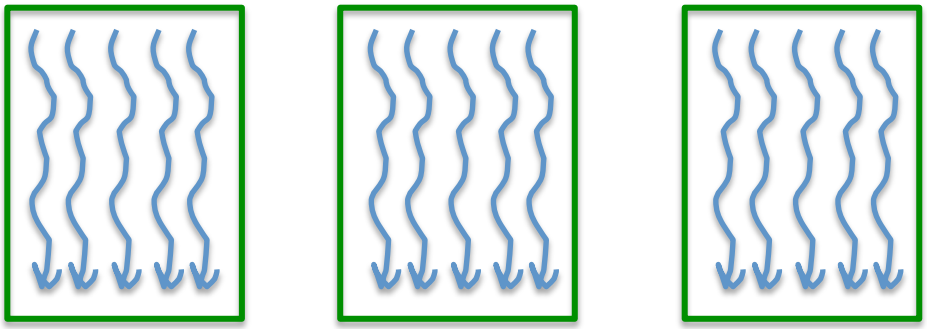
kernels - *small parts of the code that run on GPU (C/C++)*



threads

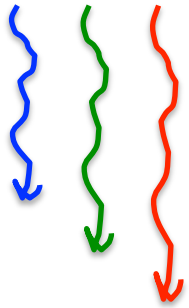
each thread might take a different path (if clauses, for loops,etc)

The key about threads is that they come in thread blocks



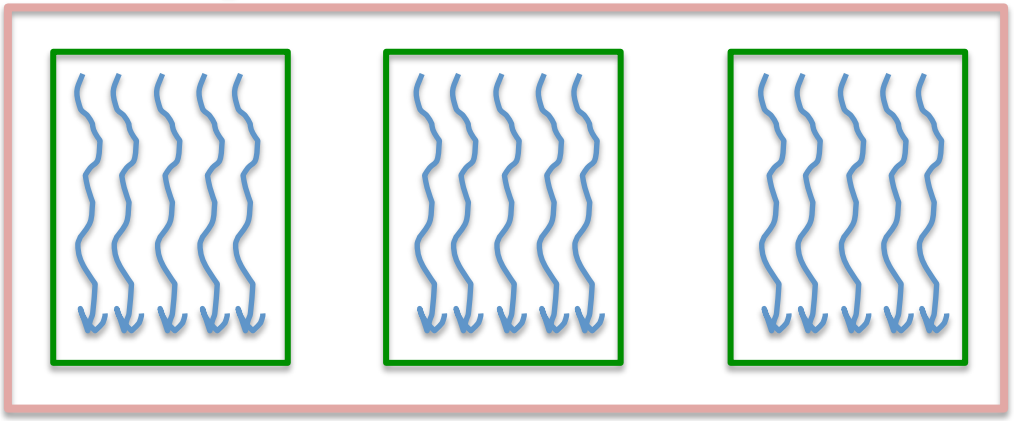
thread blocks: group of threads that cooperate to solve a certain (sub) problem

Summary of GPU programming model

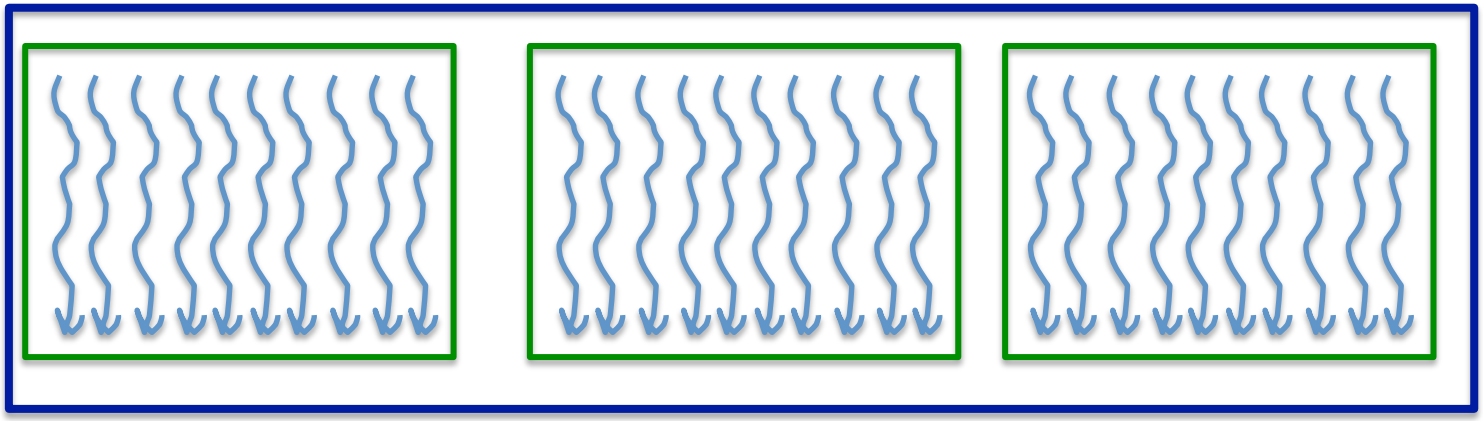


threads

thread blocks: group of threads that cooperate to solve a certain (sub) problem



kernel *square*



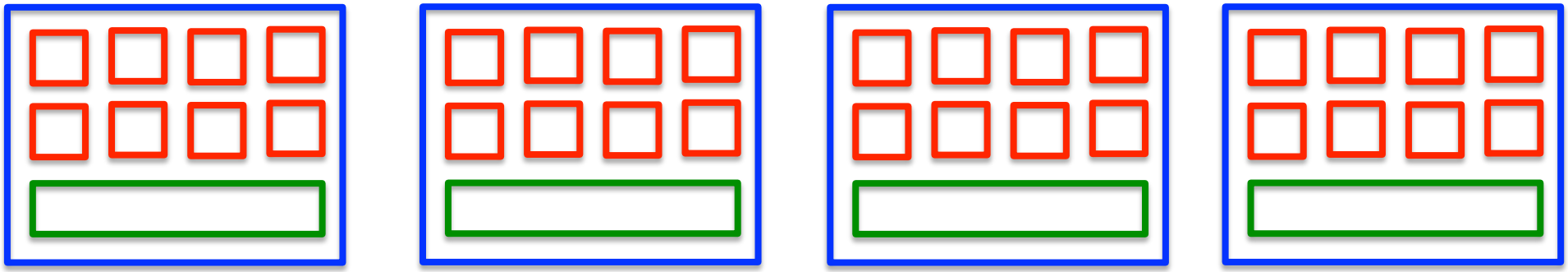
kernel *sum*

GPU Hardware

Questions to be answered:

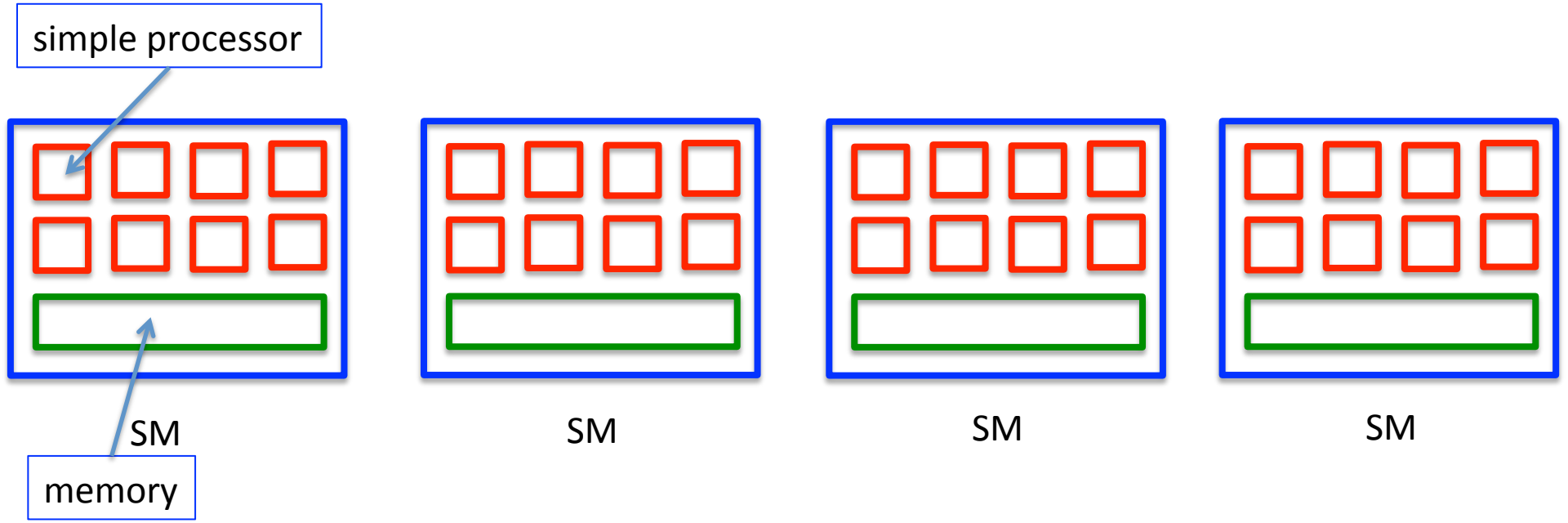
1. Why do we divide problem into blocks?
2. When are blocks run?
3. In what order do they run?
4. How can threads cooperate?
5. With what limitations do threads cooperate?

GPU Hardware



SM = streaming multiprocessor (number of those depends on a GPU)
Zhores cluster has Nvidia V100 installed; Nvidia V100 has 84 SMs

GPU Hardware



SM = streaming multiprocessor (number of those depends on a GPU)
Zhores cluster has Nvidia V100 installed; Nvidia V100 has 84 SMs

You have CUDA program with a bunch of threads organized in thread blocks:
GPU is responsible for allocating blocks to SMs!
Thread block may not run on more than 1 SM!

All the SMs run in parallel and independently

GPU Hardware

Question:

Given a single kernel that is launched on many thread blocks including X and Y, the programmer can specify:

1. That block X will run at the same time as block Y
2. That block X will run after block Y
3. That block X will run on SM 10

GPU Hardware

CUDA makes a few guarantees about when and where thread blocks can run

Advantages:

- efficiency
- scalability

Consequences:

- no assumptions on what blocks run on a given SM
- no explicit communication between blocks

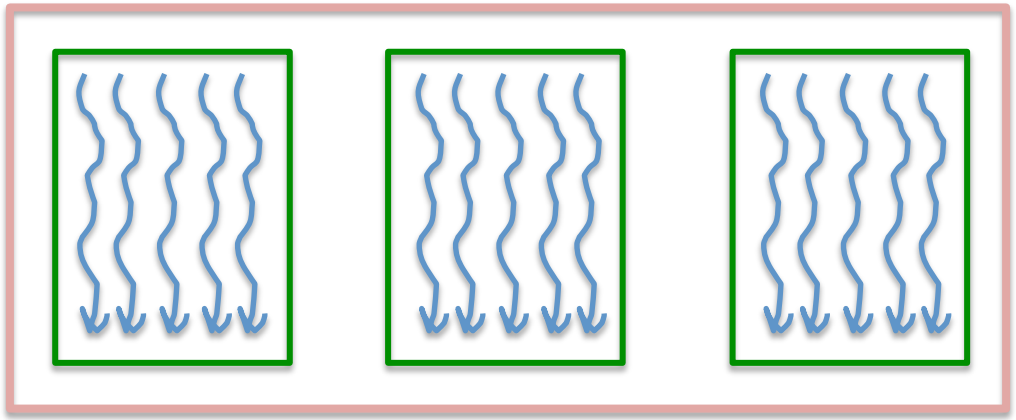
GPU Hardware

```
#include <stdio.h>

__global__ void hello()
{
        printf("Hello world from thread in a block %d\n", blockIdx.x);
}

int main(int argc, char **argv)
{
        hello<<<8, 1>>>();

        cudaDeviceSynchronize();
        printf("End!\n");

        return 0;
}
```

GPU Hardware

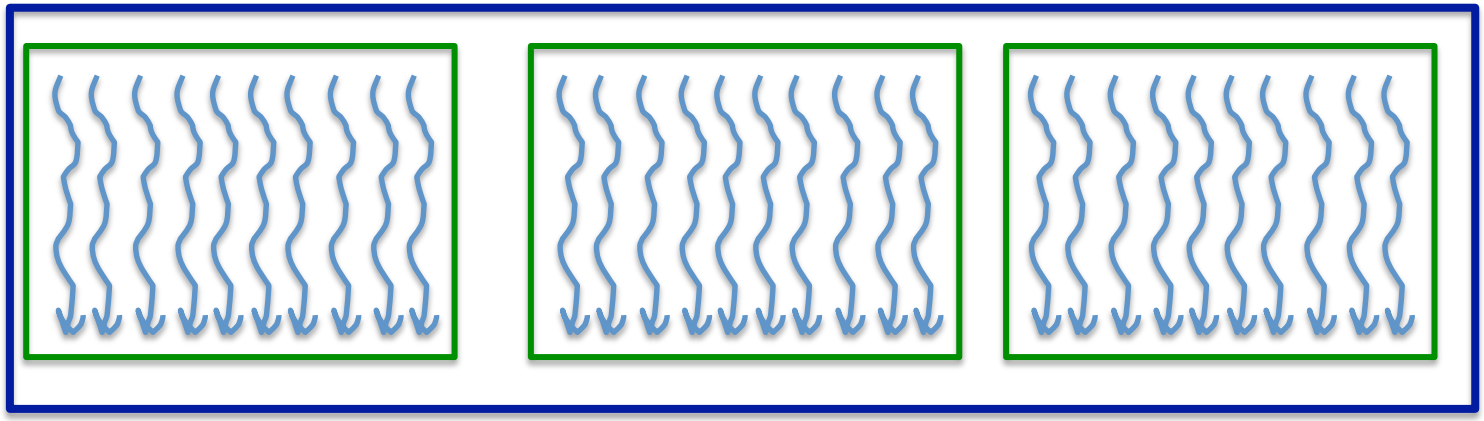
CUDA guarantees:

1. **All threads in a block run on the same SM at the same time.**
2. **All blocks in a kernel finish before any blocks from the next kernel run.**



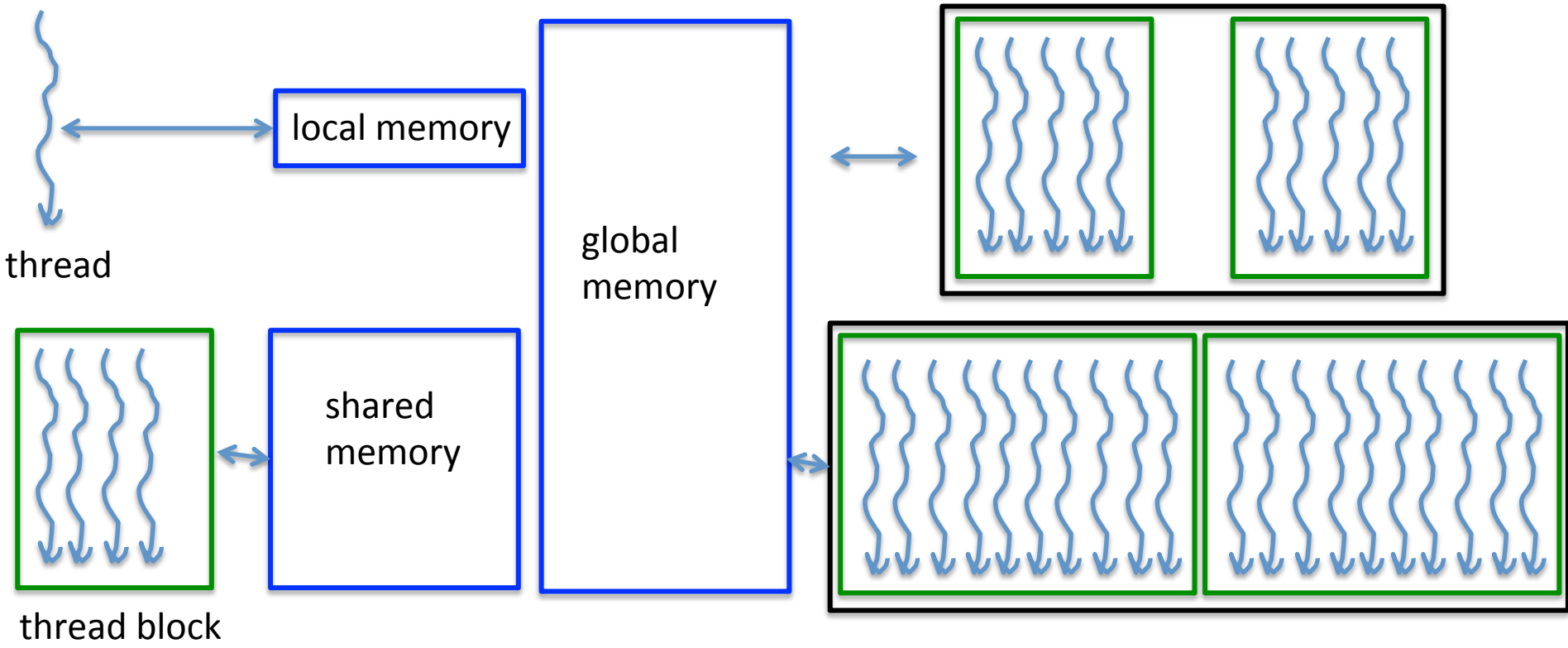
kernel *square*

intrinsic barrier

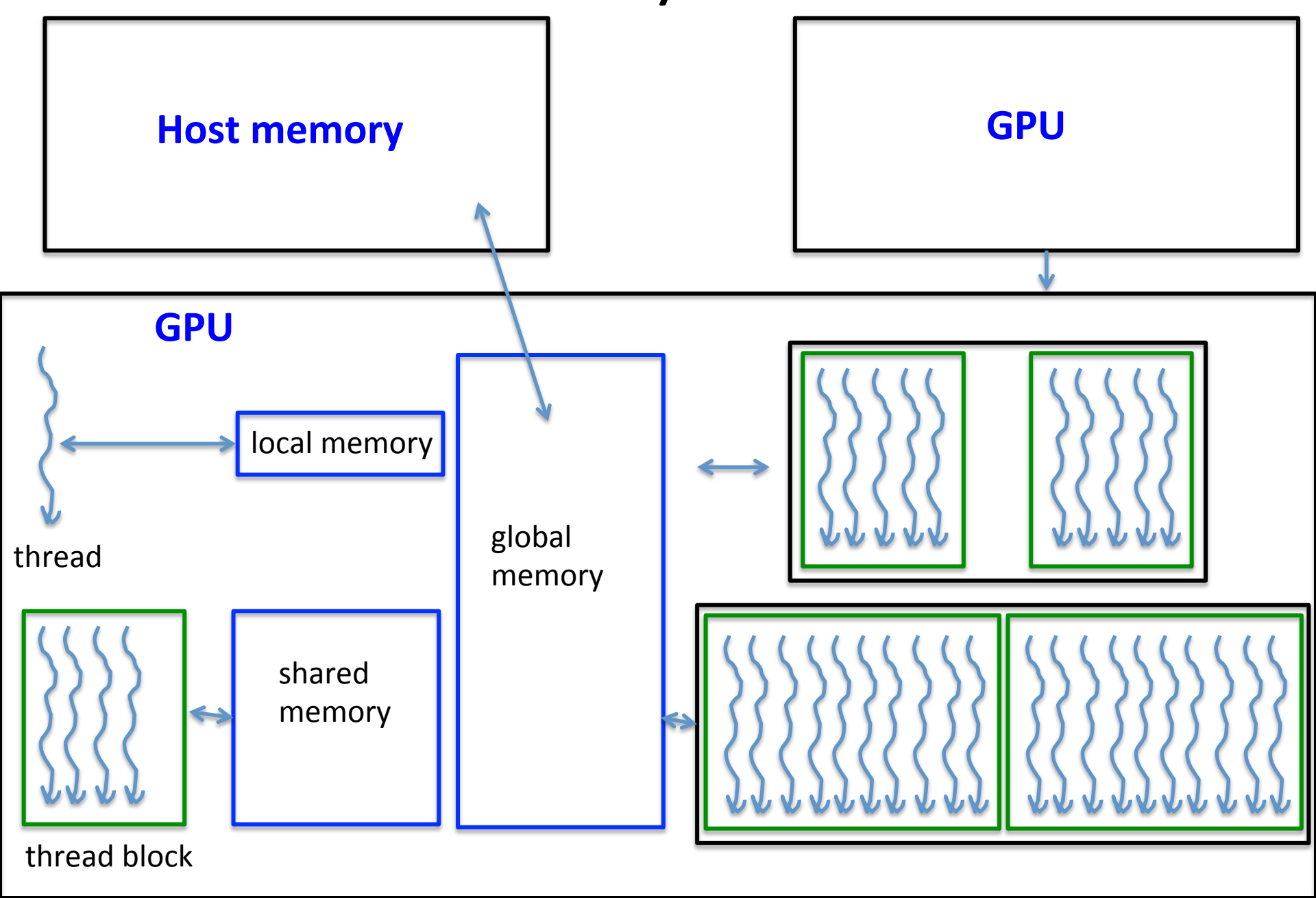


kernel *sum*

Memory model



Memory model



GPU Hardware

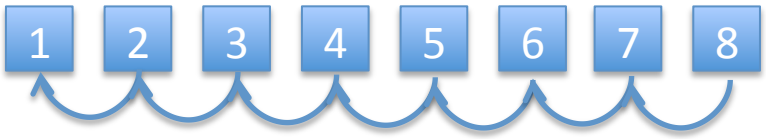
Questions:

1. All threads from a block can access the same variable in that blocks shared memory
2. Threads from 2 different blocks can access the same variable in global memory
3. Threads from different blocks have their own copy of local variables in local memory
4. Threads from the same block have their own copy of local variables in local memory

Synchronization

Threads can access each other's results through global and shared memory

DANGER: What if one thread tries to read the result before the other one writes it?

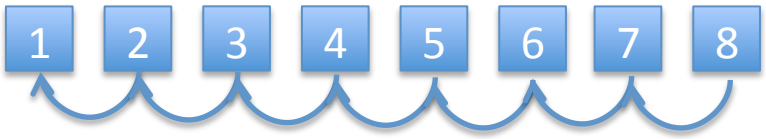



```
…
int idx=threadIdx.x;
__shared__ int array[128];

array[idx]=idx;

if (idx<127){
    array[idx]=array[idx+1];
}
...
```

Synchronization

Threads can access each other‘s results through global and shared memory

DANGER: What if one thread tries to read the result before the other one writes it?

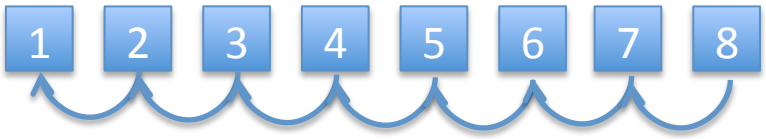



```
…
int idx=threadIdx.x;
__shared__ int array[128];

array[idx]=idx;
__syncthreads();

if (idx<127){
    array[idx]=array[idx+1];
}
...
```

Synchronization

Threads can access each other's results through global and shared memory

DANGER: What if one thread tries to read the result before the other one writes it?

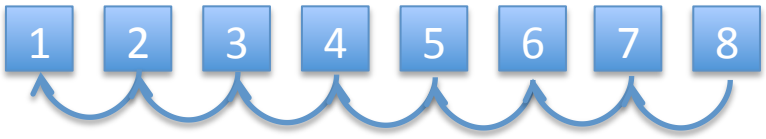



```
…
int idx=threadIdx.x;
__shared__ int array[128];

array[idx]=idx;
__syncthreads();

if (idx<127){
    int temp = array[idx+1];
    __syncthreads();
    array[idx]=temp;
}
...
```

Synchronization

Threads can access each other's results through global and shared memory

DANGER: What if one thread tries to read the result before the other one writes it?




```
…
int idx=threadIdx.x;
__shared__ int array[128];

array[idx]=idx;
__syncthreads();

if (idx<127){
    int temp = array[idx+1];
    __syncthreads();
    array[idx]=temp;
}
...
```

Writing efficient programs

High-level strategies:

1. **Maximize arithmetic intensity: math/memory**
    1. **maximize compute operations per thread**
    2. **minimize memory access time**

Minimize time spent on memory

1. **Move frequently accessed data to fast memory**

local faster than **shared** much faster than **global** much faster than **CPU RAM**

code snippet

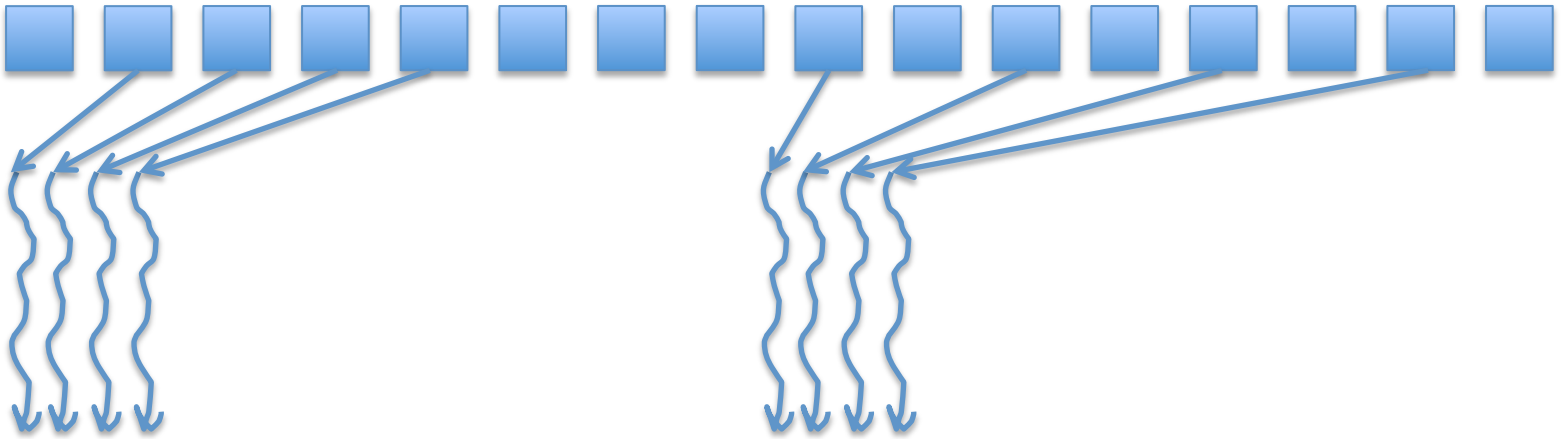
Minimize time spent on memory

1. **Move frequently accessed data to fast memory**

local faster than **shared** much faster than **global** much faster than **CPU RAM**

code snippet

2. **Coalesce global memory access**



coalesced
good

strided
no so good

random
bad

Minimize time spent on memory

```
__global__ void foo(float *g)
{
    float a = 3.14;
    int i = threadIdx.x;

    g[i] = a;

    g[i*2] = a;

    a = g[i];

    a = g[BLOCK_WIDTH/2 + i];

    g[i] = a * g[BLOCK_WIDTH/2 + i];

    g[BLOCK_WIDTH-1 - i] = a;
}
```

Writing efficient programs

High-level strategies:

1. **Maximize arithmetic intensity: math/memory**
    1. **maximize compute operations per thread**
    2. **minimize memory access time**
2. **Be careful with atomics**
3. **Avoid thread divergence**