Parallel Processing

Unit 2

Dr. Ahmed Sallam

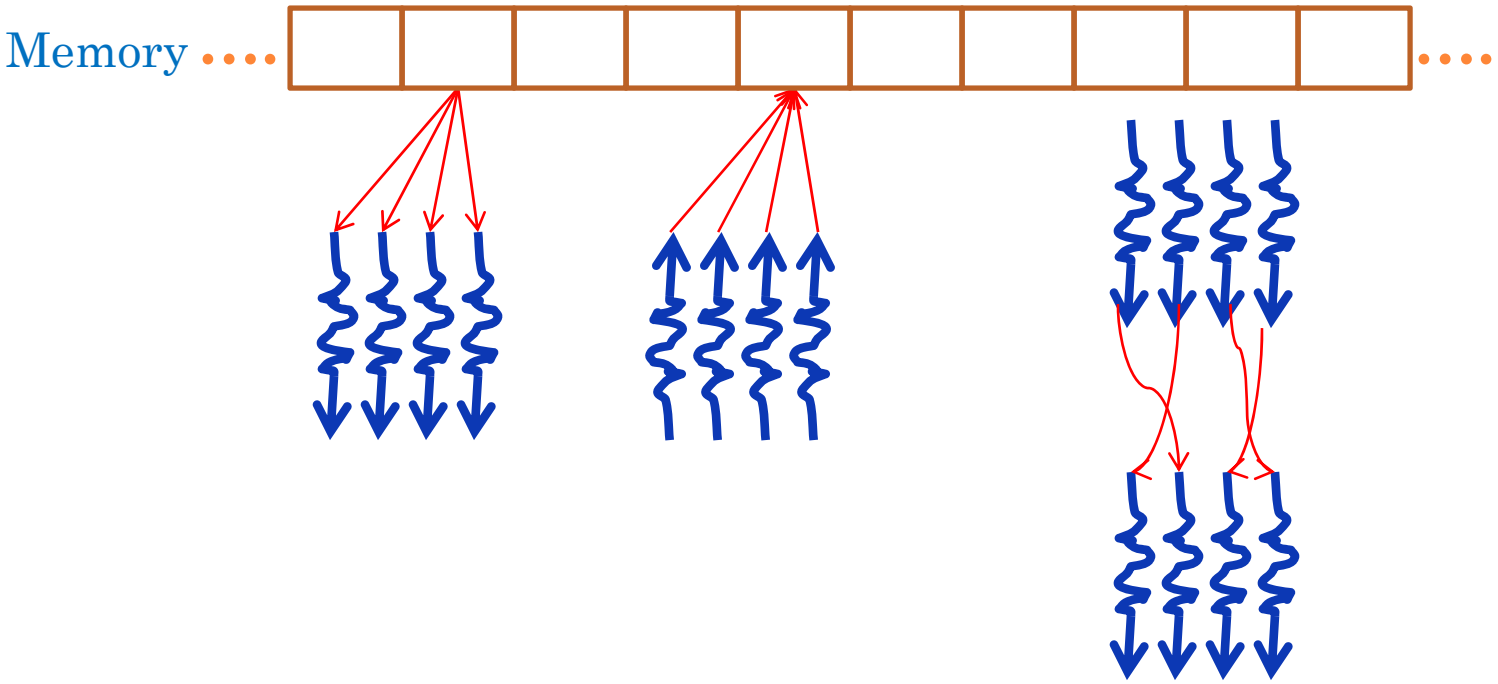
OUTLINES

- Threads communication patterns
- GPU hardware: Processing model
- GPU hardware: Memory model
- Efficient GPU programming

Threads communication patterns

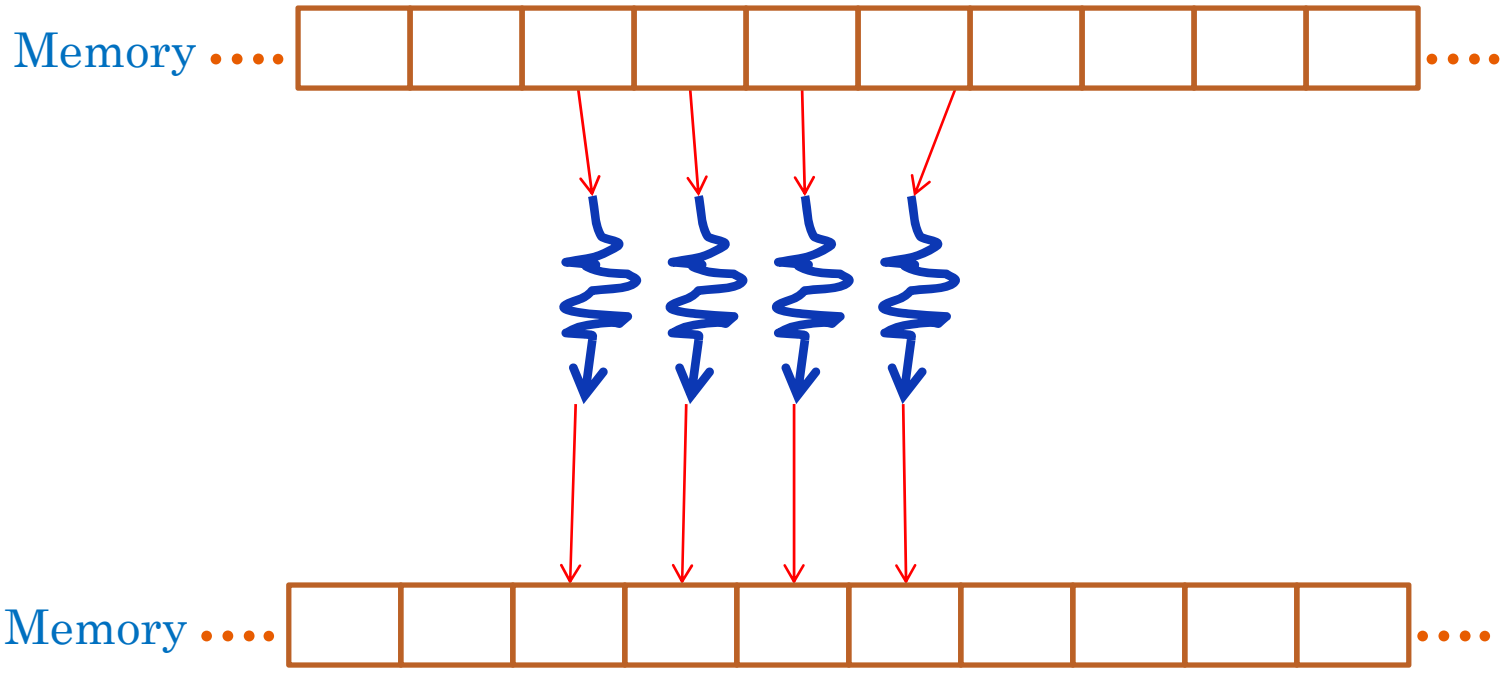
COMMUNICATION PATTERNS

- Parallel Programming: threads solving problem by working together (Communicating through memory)



Map

- Tasks read from and write to specific data elements

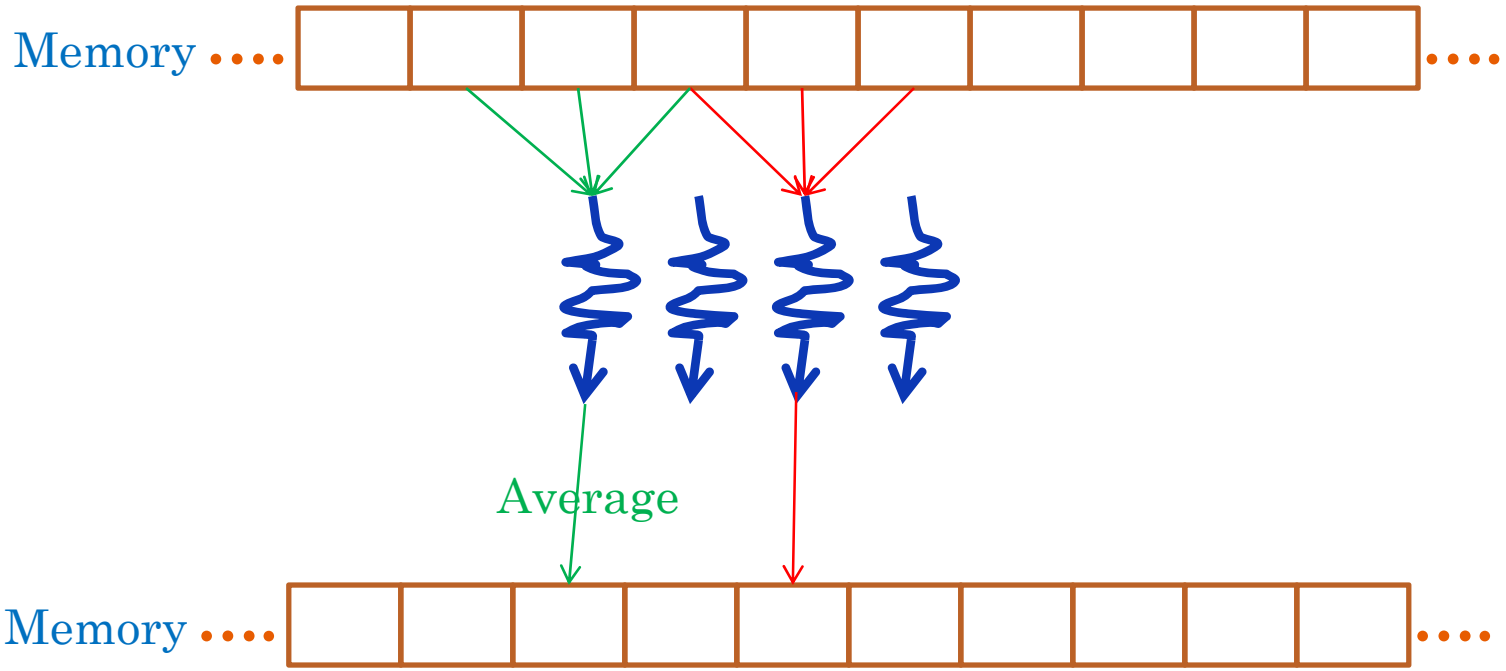


Map Application

- Color image to Gray image

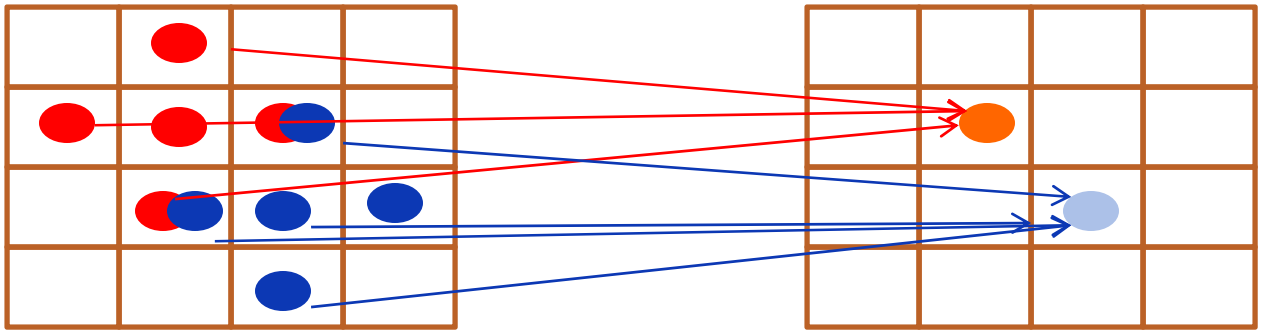
GATHER

- Problem: Many threads read from the same place (Data reuse)



Gather Application

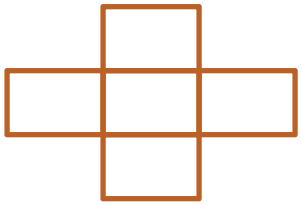
- Image Blurring



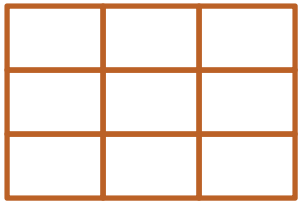
GATHER (STENCIL)

- Tasks gather input from a fixed neighbors.

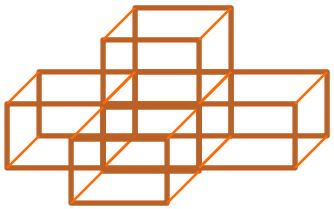
2D Neumann



2D Moore

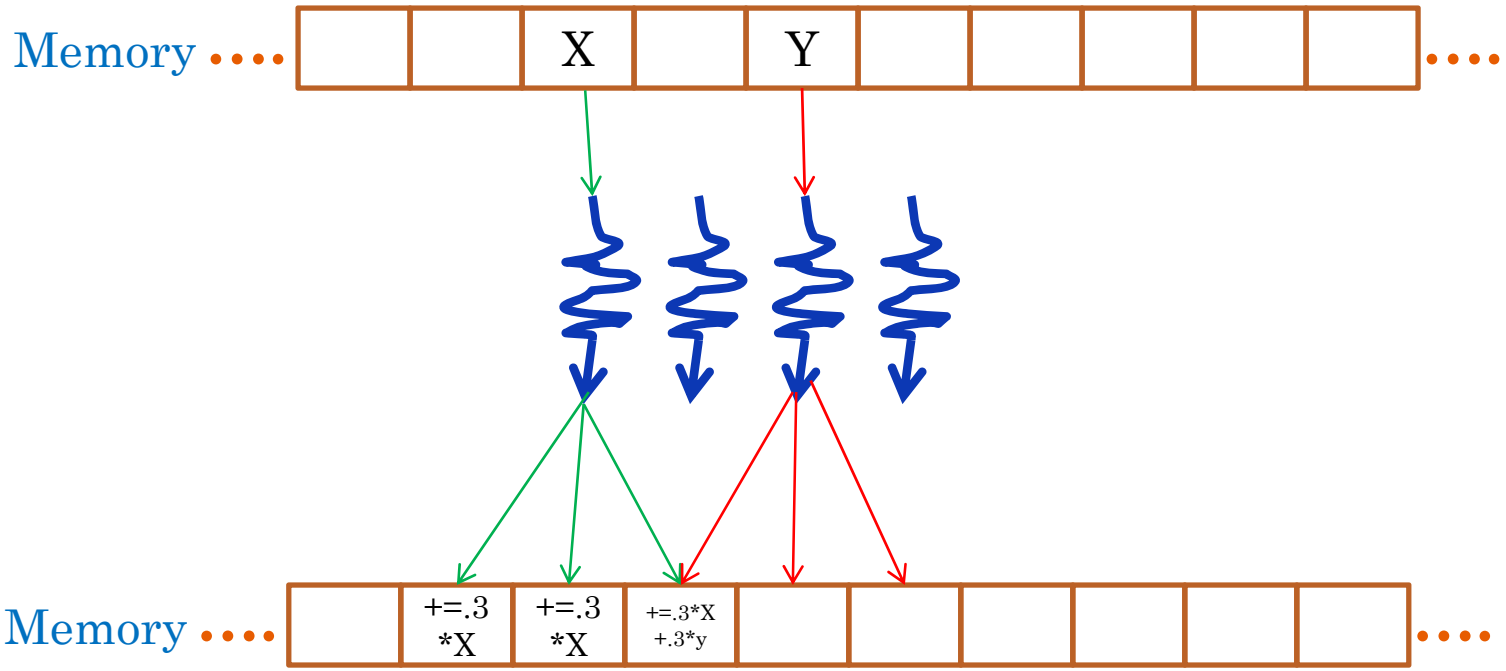


3D Neumann



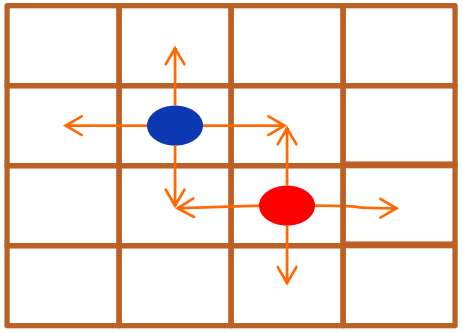
Scatter

- Tasks compute where to write output
- Problem: threads write to the same place



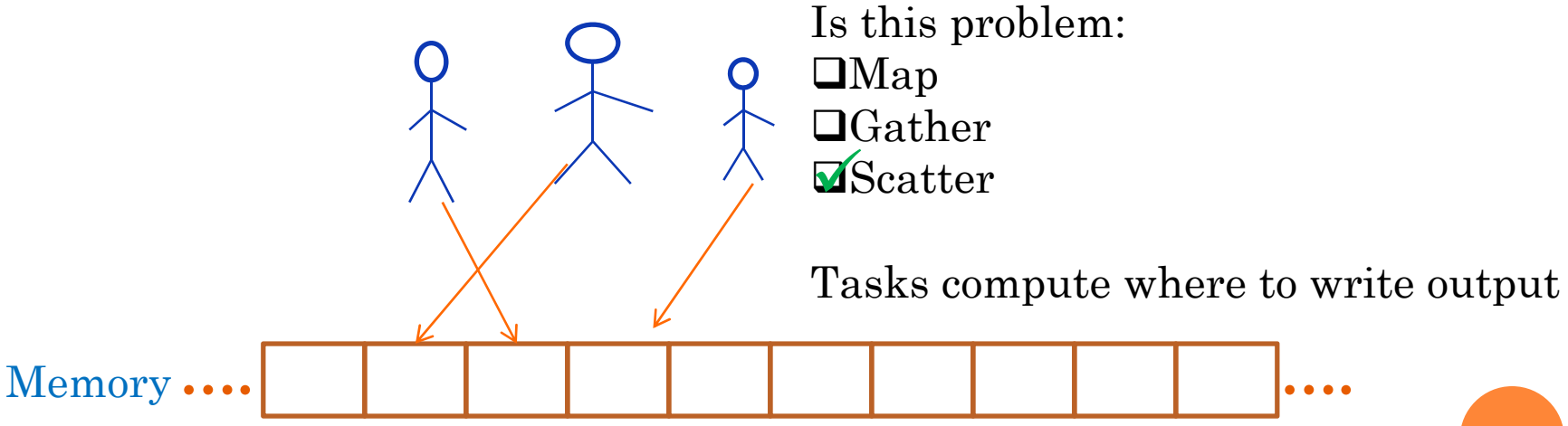
Scatter Application

- Image Blurring



QUIZ

- Given a list of players with the information (name, height, tall-rank)
- Write each player's record in it's location in a sorted list according to the tall-rank



TRANSPOSE

1	2	3	4
5	6	7	8
9	10	11	12

1	5	9
2	6	10
3	7	11
4	8	12

Row major order

Memory .... | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | ....

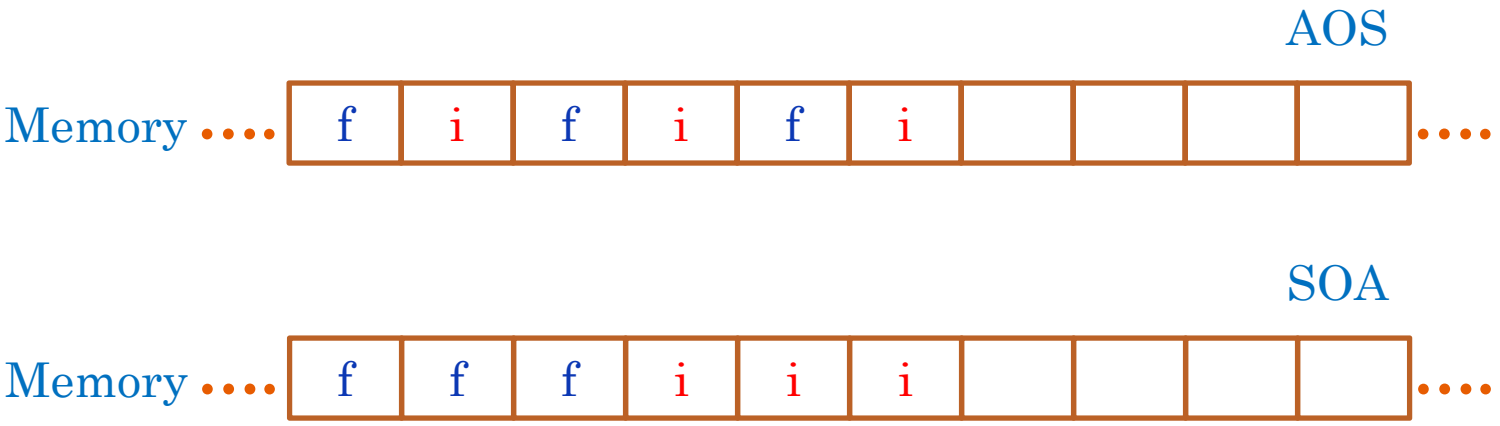
Column major order

Memory .... | 1 | 5 | 9 | 2 | 6 | 10 | 3 | 7 | | | ....

Transpose Applications

- Array
- Matrix
- Image
- Data Structures (AOS to SOA)

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



QUIZ

- Label code snippets by problem:

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

const float pi= 3.1415;
```

1
```
out[i]= pi* in[i];
```

5
```
out[i + j*128]=in[j + i*128];
```

```
if(i%2){
```
3
```
    out[i-1] +=pi * in[i];  out[i+1] += pi * in[i];
```

2
```
    out[i] = (in[i] + in[i-1]+ in[i+1] * pi/3.0f)
}
```

1. Map
2. Gather
3. Scatter
4. Stencil
5. Transpose

Notes

- Map — one-to-one
- Transpose — one-to-one
- Gather — many-to-one
- Scatter — one-to-many
- Stencil — several-to-one

- Reduce — all-to-one
- Scan/sort — all-to-all

NOTES (CONT.)

- How can threads efficiently access memory in concert?
  - How to exploit data reuse?
- How can threads communicate partial results by sharing memory?
  - safely

OUTLINES

- Threads communication patterns
- GPU hardware: Processing model
- GPU hardware: Memory model
- Efficient GPU programming

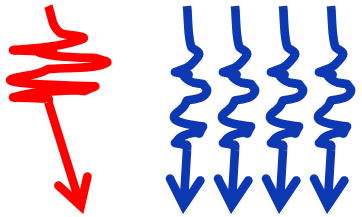
GPU HARDWARE

Processing model

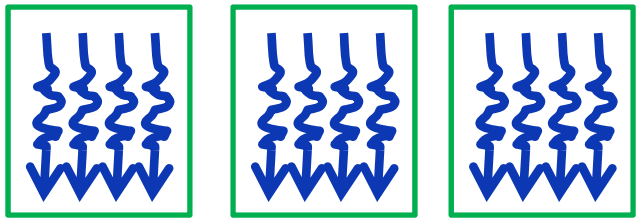
PROGRAMMER VIEW OF THE GPU

- Kernels C/C++ function

- Threads

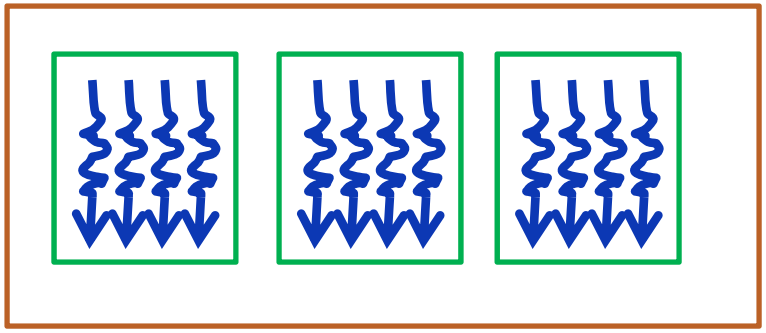


- Blocks: Group of threads that cooperate to solve subproblem.

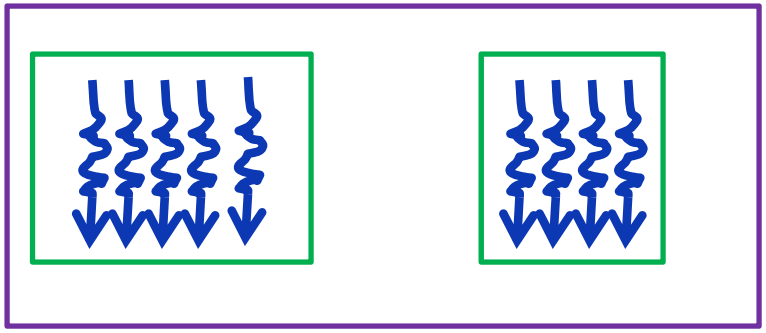


Programmer view of the GPU (cont.)

- Kernel

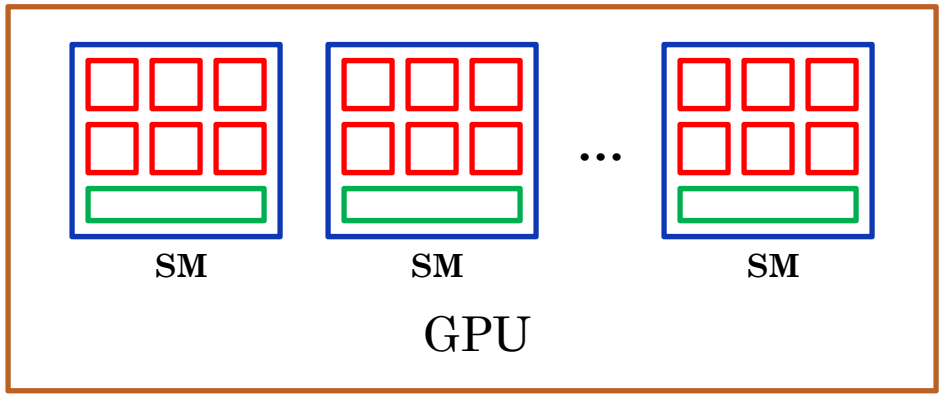


- Another Kernel



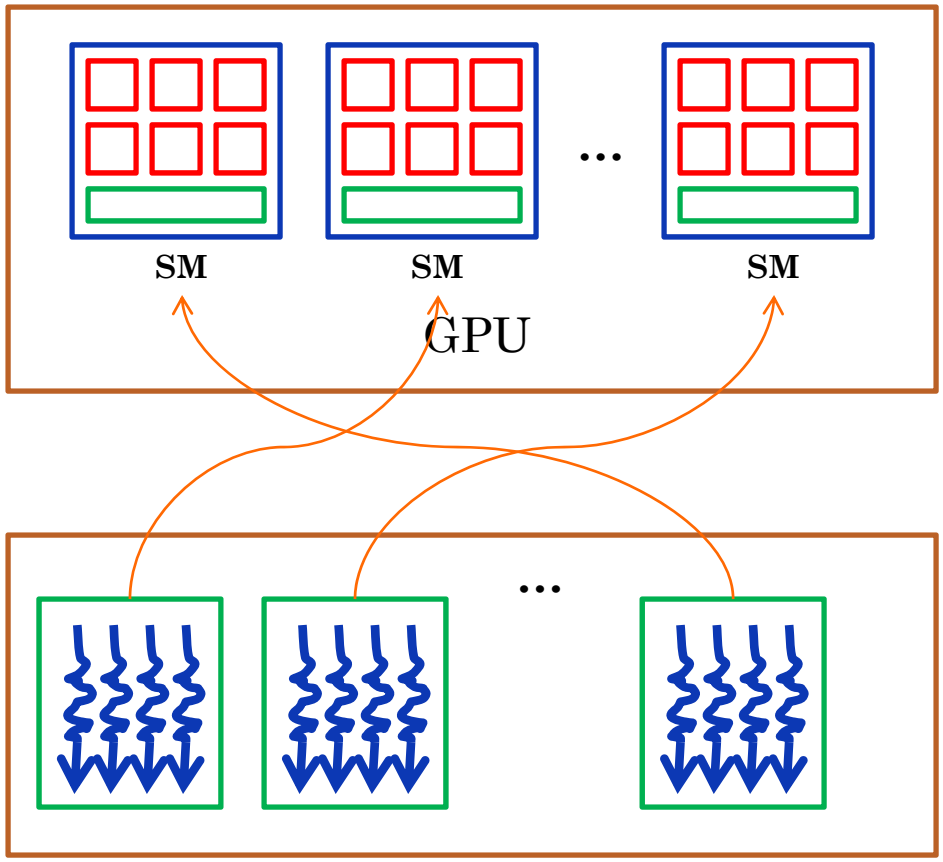
GPU Hardware

- GPU consists of Streaming Multiprocessors (SMs)
- SM consists of many cores each has local memory (Registers) and shared memory (L1,L2 cache)



BLOCKS GPU HARDWARE

- GPU is responsible for allocating blocks to SMs



QUIZ

- ✓ A thread block contains many threads
- ✓ An SM may run more than one block
- A block may run on more than SM
- ✓ All the threads in a thread block may cooperate to solve a subproblem
- All the threads that run on a given SM may cooperate to solve a subproblem

- ✓ The programmer is responsible for defining thread blocks in software.
- The programmer is responsible for allocating thread blocks to hardware streaming multiprocessors (SMs)

QUIZ

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

- That block X will run at the same time as block Y
- That block X will run after block Y
- That block X will run on SM Z

CUDA GUARANTEES

CUDA makes few guarantees:

- All threads in the same block can run on the same SM at the same time
- All blocks in the kernel finish before any blocks in another kernel

Advantages:

- Hardware can run things efficiently (Flexibility)
- No waiting on slow blocks if we have free SMs
- Scalability
  - From cell phones to super computers
  - From current to future GPUs

Consequences :

- No assumptions about what SM run what Block.
- No communication between blocks
  - If Block Y needs result from Block X, we could have a dead lock state because Block X will evaporate after completion.

QUIZ

- How many different outputs can different run of this program produce?

```
#include <stdio.h>

#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1
__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n", blockIdx.x);
}

int main(int argc,char **argv)
{
    // launch the kernel
    hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();

    printf("That's all!\n");

    return 0;
}
```

Is this problem:

- ❑1
- ❑16
- ❑65,536
- ☑21 trillion

=16!

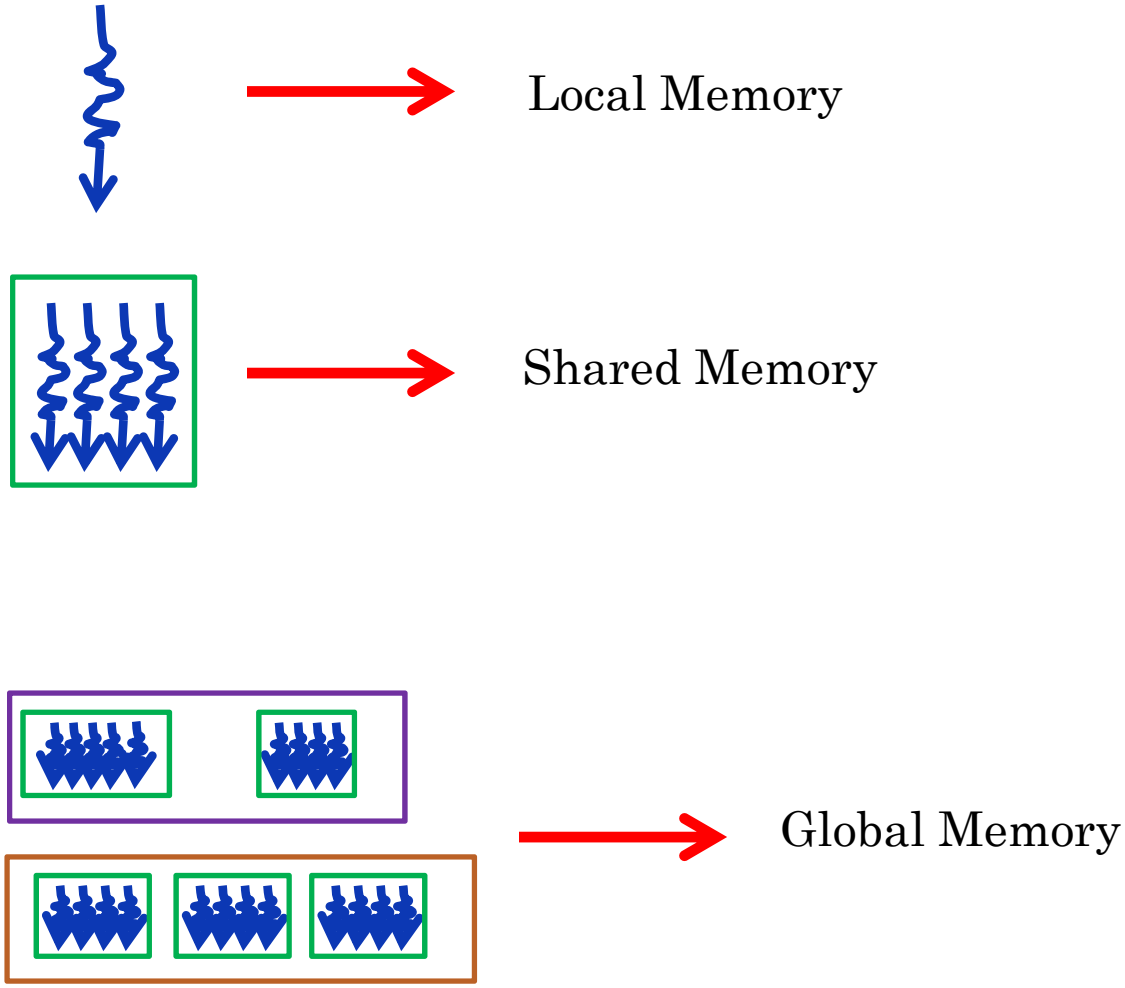
OUTLINES

- Threads communication patterns
- GPU hardware: Processing model
- GPU hardware: Memory model
- Efficient GPU programming

GPU Hardware

Memory model

GPU Memory



GPU Memory (cont.)

```
__global__ void GPU_memory(float *out, float *in )
{

    int i= threadIdx.x;
    int j= threadIdx.y;

    const float pi= 3.1415;

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

    __shared__ float sh_arr[128];


}
```

QUIZ

- All threads from a block can access the same variable in that blocks shared memory
- Threads from two different blocks can access the same variable in global memory
- Threads from different blocks have their own copy of local variables in local memory

- Threads from the same block have their own copy of local variables in local memory

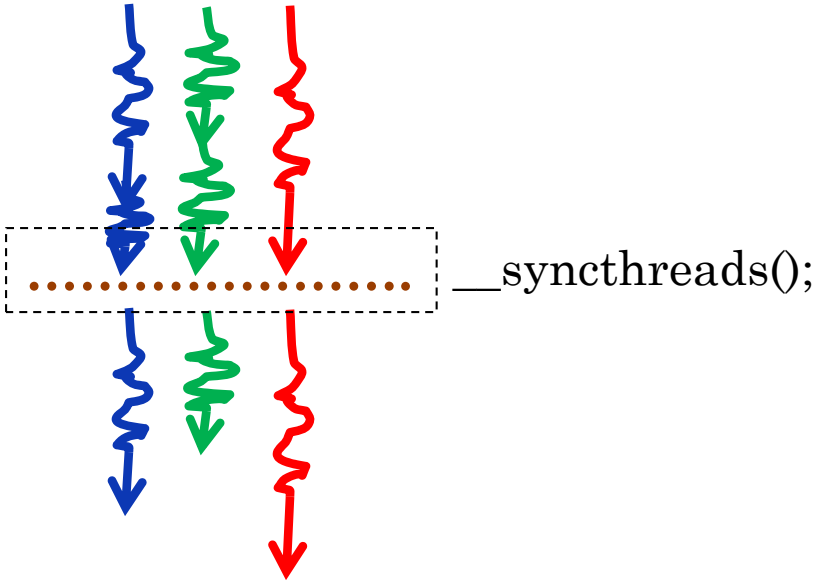
SYNCHRONIZATION

- Treads can access each other's results through shared and global memory. (Work together)

- Problem: what if a thread reads a result before another thread writes it?

Synchronization Barrier

- Barrier: is a point in the program where threads stop and wait when all threads have reached the barrier, then they can proceed.



QUIZ

- Are the following code snippet correct?

```
__global__ void foo( ... )
{
    __shared__ int s[128];
    int i= threadIdx.x;

✘   s[i]=i;
    __syncthreads();

✘   s[i]= s[i+1];   →   temp=s[i];
                        __synctrheads();
                        s[i]=s[i+1];
                        __synctrheads();

    if(i%2)
✓       s[i]=s[i-1];
}
```

OUTLINES

- Threads communication patterns
- GPU hardware: Processing model
- GPU hardware: Memory model
- Efficient GPU programming

Writing Efficient Program

Efficient memory access

Maximize arithmetic intensity

A high-end GPU can do 3 trillion Floating point Operation Per Second (3 FLOPS)

$$\frac{\text{Math}}{\text{memory}}$$

- Maximize compute operations per thread.
- Minimize time spent on memory per thread.
  - (The access time not the amount)

MINIMIZE TIME SPENT ON MEMORY PER THREAD.

- The access time not the amount

Local memory > Shared memory >> Global memory

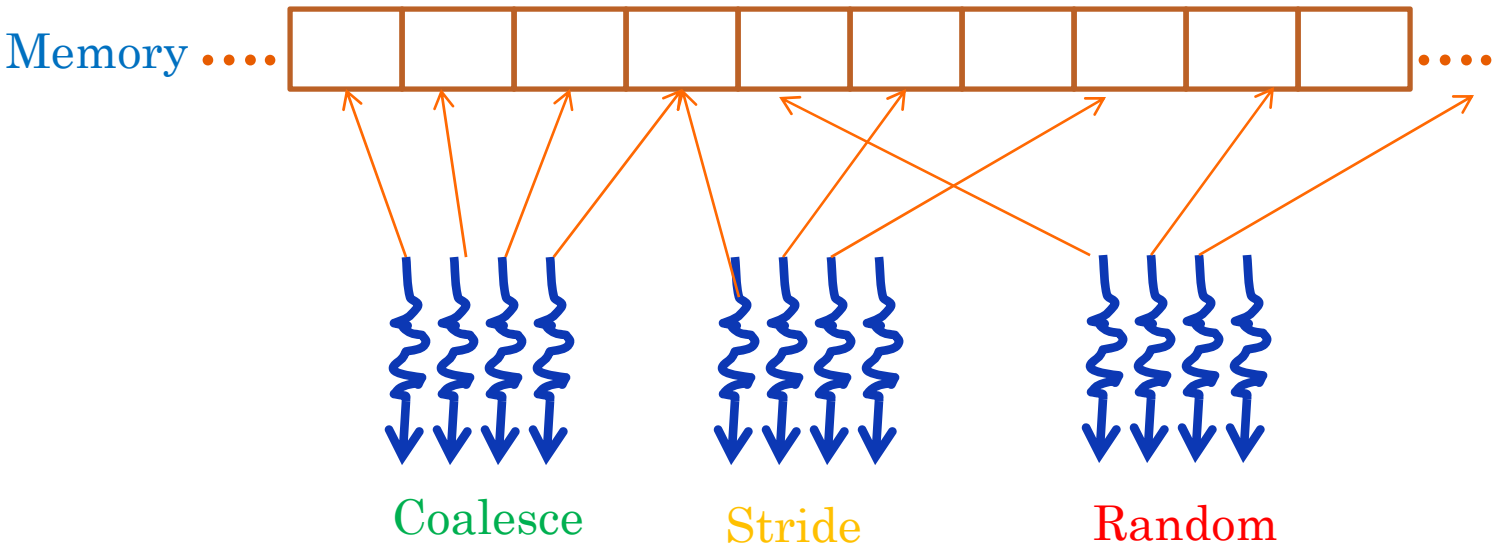
QUIZ

- Sort following variable based on their memory speed

```
__global__ void GPU_memory(float *out, float *in )
{

    int i= threadIdx.x;
    int j= threadIdx.y;

    const float pi= 3.1415;

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

    __shared__ float sh_arr[128];


}
```

USING COALESCED GLOBAL MEMORY ACCESS

- We can read memory by block not by unites



Atomic operation

Atomic operations

- Special hardware to prevent more than one thread access the same memory
- Has specific operations (add, sub, xor) and works with specific data types (int)

Example

Atomic operations limitations

- Only certain operations
  - atomicCAS() has a broader use “Compare and swap”
- Still no ordering constraints
- Serializes access to memory (Slow)

QUIZ

State which sentence give the correct results and sort them based on memory access time :

- ☑ 1 million threads incrementing 1 million elements
- ☑ 1 million threads atomically incrementing 1 million elements
- ☐ 1 million threads incrementing 100 elements

- ☑ 1 million threads atomically incrementing 100 elements
- ☑ 10 million threads atomically incrementing 100 elements

THREAD DIVERGENCE

- Running threads with loops or conditional statement which diverse the time of each thread.

OUTLINES

- Threads communication patterns
- GPU hardware: Processing model
- GPU hardware: Memory model
- Efficient GPU programming