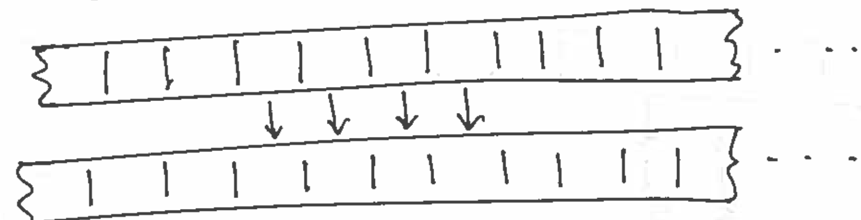
Unit-02

Parallel Communication Patterns

Parallel communication patterns are really all about how to map tasks and memory together. Mapping the tasks, which are threads in CUDA, to the memory they are communicating through.

(1) **Map:** This communication pattern is a one-to-one mapping, which is very efficient on GPUs.

memory



One-to-one mapping

(2) **Gather:** Each calculation gathers input data elements together from different places to compute an output result.

moving average

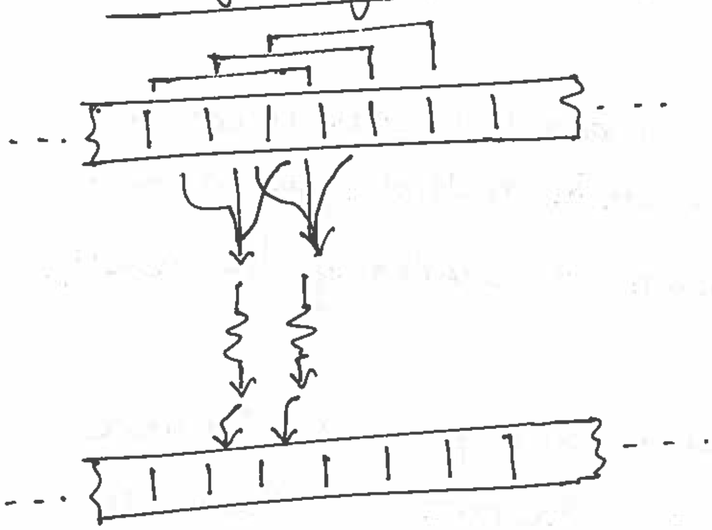
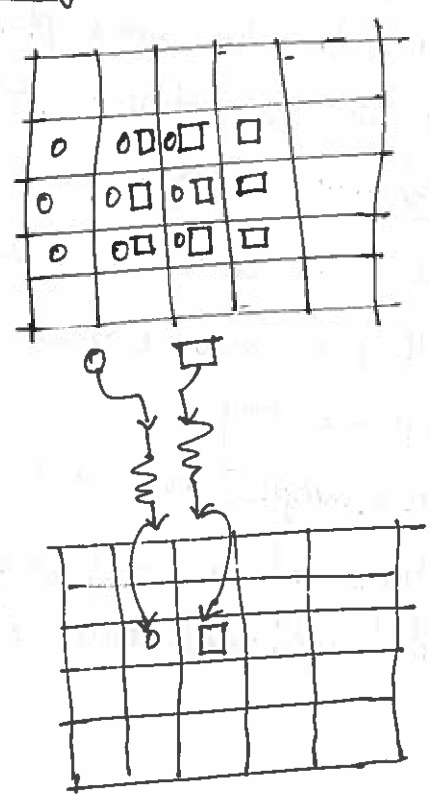
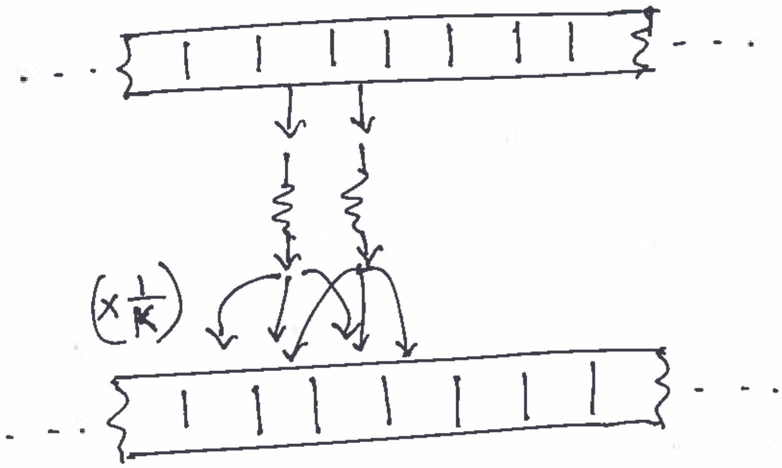
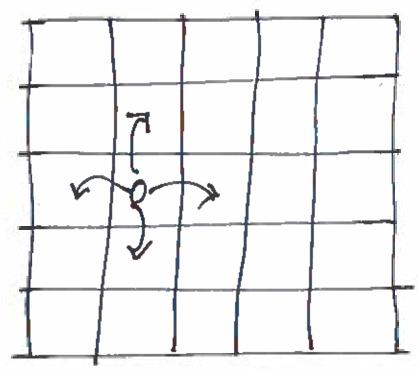


image blurring



(03) **Scatter:** Unlike the example of moving average on image blurring under the "Gather" operation, where each thread read k neighboring elements and sum them up, we can have each thread reading a single input result and add (1/k) of its elements value to the k neighboring elements. In that case, each of the writes would really be an increment operation. The same thing applies to the 2D image blurring example.





The problem with scatter is that several threads attempt to write to the same place more or less at the same time. We will talk about this later.

Definition of ~~ea~~ scatter: When each parallel task needs to write its result in a different place or in multiple places, we call this scatter because the threads are scattering the results over memory.

For example, you have a list of basketball players. You have a bunch of records and each one has the name of the player, and height and rank of the player. For simplicity, suppose the rank is based on the height of the players. And say that the goal is to write the records into a sorted list based on the rank. So, if we implement this in CUDA by having each thread read a record, look at the rank, and we that rank to determine where to write into the array, then this is a scatter operation. Each thread is computing where to write its result.

04 <u>Stencil:</u> tasks read input from a fixed neighborhood in an array.

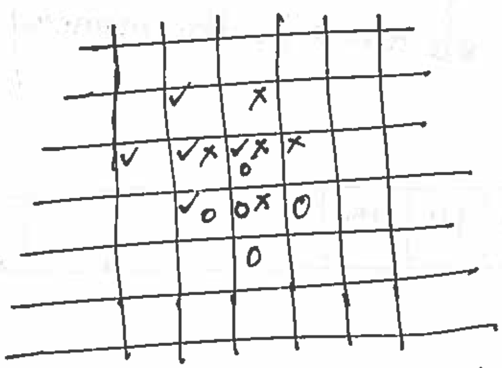
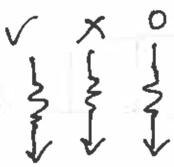
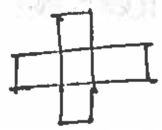


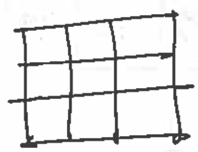
Image blurring example
2D von Neumann stencil for 3 threads



<u>stencil patterns</u>



2D von Neumann



2D Moore

Also, 3D von Neumann, 3D Moore etc.

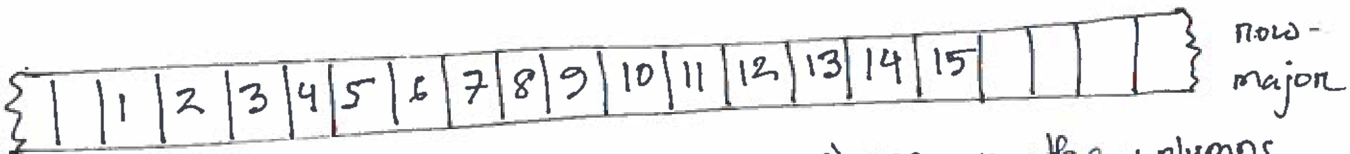
So, there is a lot of data reuse going on. Many threads are accessing and computing from the same data. ~~At~~ And, exploiting that data reuse is something we are going to do later on to speed up the process.

05 **Transpose:** Suppose, you have an array (an image), laid out in a row-major order (one row at a time).

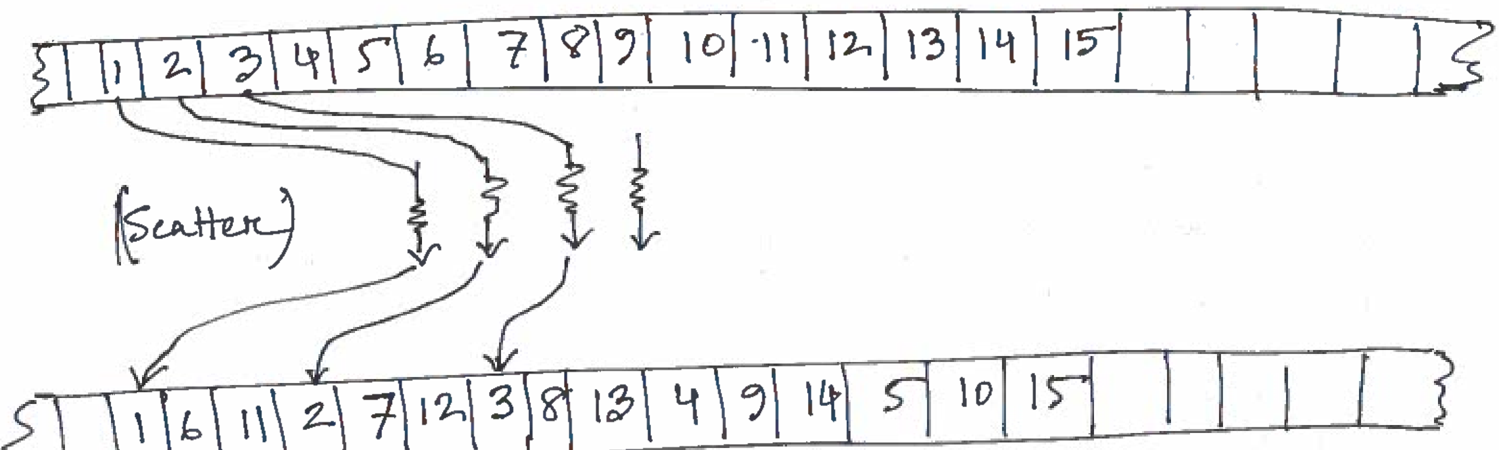
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

$\Longrightarrow$

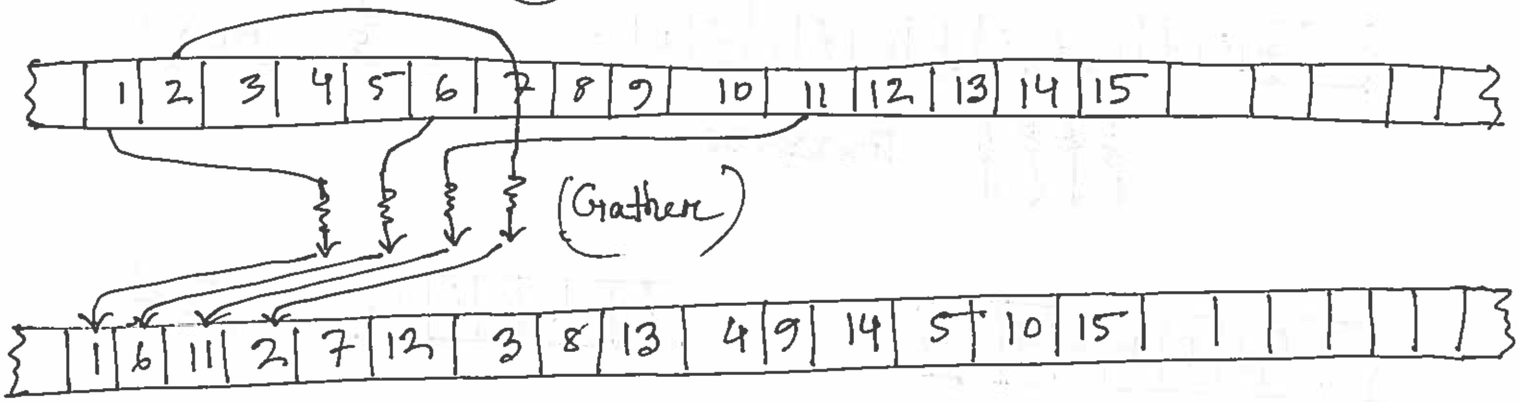
1	6	11
2	7	12
3	8	13
4	9	14
5	10	15



But you might want to do some processing on the columns of the image. And so, you'd want a layout like this column-major order. So, you need to reorder the elements in the memory.



It is possible to use either Scatter or Gather operations to do this. In case of Scatter, each thread is reading from an adjacent element in the array, but is writing to some place scattered in memory, according to the stride of this row-column transpose.



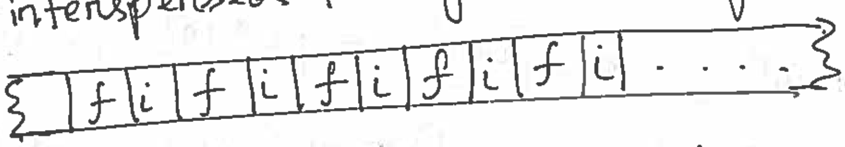
So, you can see that a transpose operation might come up when you are doing array operations, matrix operations, image operations. But the concept is generally applicable to all kinds of data structures. Here is an example:

Say, you have a structure named foo. and you have an array of thousand of these.

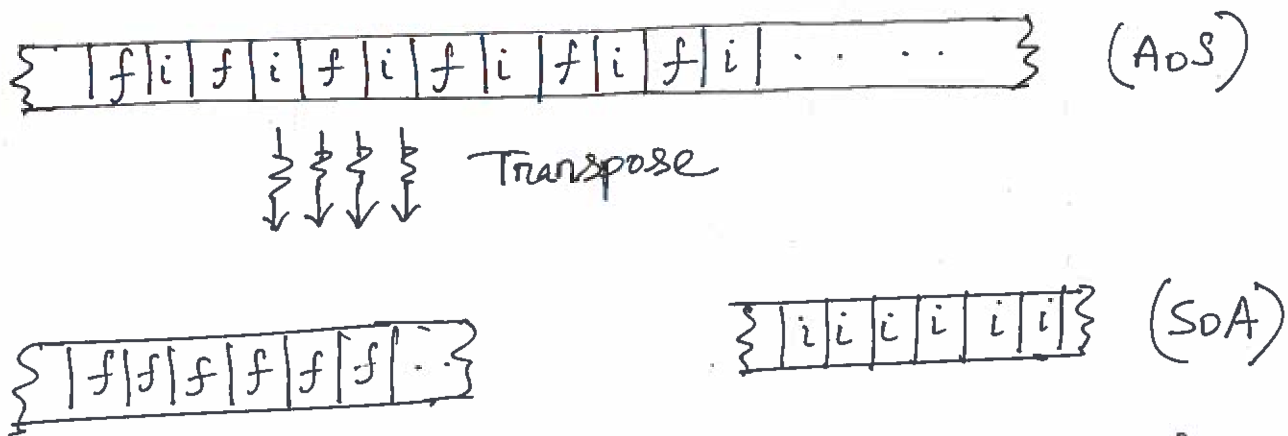
```
struct foo {
    float f;
    int i;
};
```

```
foo array[1000];
```

The memory footprint of this array is that floats and ints are interspersed throughout memory.



Now, if you are going to do a lot of processing on the floats, it can be more efficient to access all the floats contiguously. So, you want to turn your array of structures (AoS) representation into a structure of arrays (SoA). And, this operation is actually a transpose.



So, in general, the transpose operation is the task of reordering the data elements in the memory.

Example of communication patterns (Code Snippet)

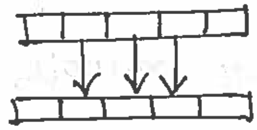
```
float out[ ], in[ ];
int i = threadIdx.x;
int j = threadIdx.y;
const float pi = 3.1415;

out[i] = pi * in[i];  (Map)
out[i+j*128] = in[j + i*128];  (Transpose)
if (i%2) {
    out[i-1] += pi * in[i];   out[i+1] += pi * in[i];  (Scatter)
    out[i] = (in[i] + in[i-1] + in[i+1]) * pi/3.0f;  (Gather)
}
```

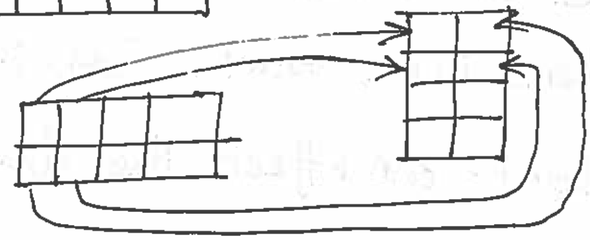
Not Stencil because, it is not writing to every location because of the modulus guard.

Parallel Communication Patterns

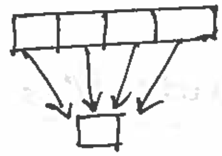
(1) Map — one-to-one



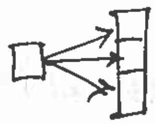
(2) Transpose — one-to-one



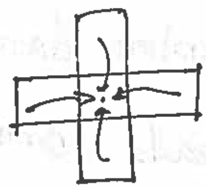
(3) Gather — many-to-one



(4) Scatter — one-to-many

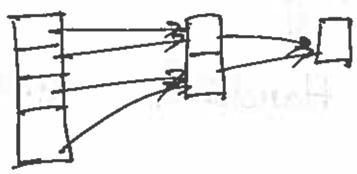


(5) Stencil — several-to-one

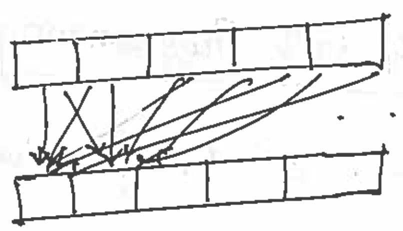


---

(6) Reduce — all-to-one



(7) Scan/Sort — all-to-all



Map/Transpose are one-to-one because each input maps to a single unique output. Gather is many-to-one because many possible inputs can be chosen to compute an output. Scatter is one-to-many because ~~man~~ each thread chooses from many possible output destinations. Stencil can be seen as a specialized gather that pulls output from a few selected inputs in a given neighborhood of the output. Reduce could be turned all-to-one. For example, if you are adding up the numbers in an array. Scan/Sort is all-to-all because all of the input can affect the destination of the resulting output.

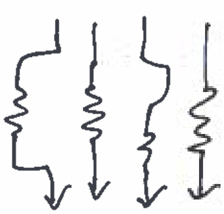
Deeper Questions About PCPs

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

$\Rightarrow$ GPU Hardware knowledge is required.

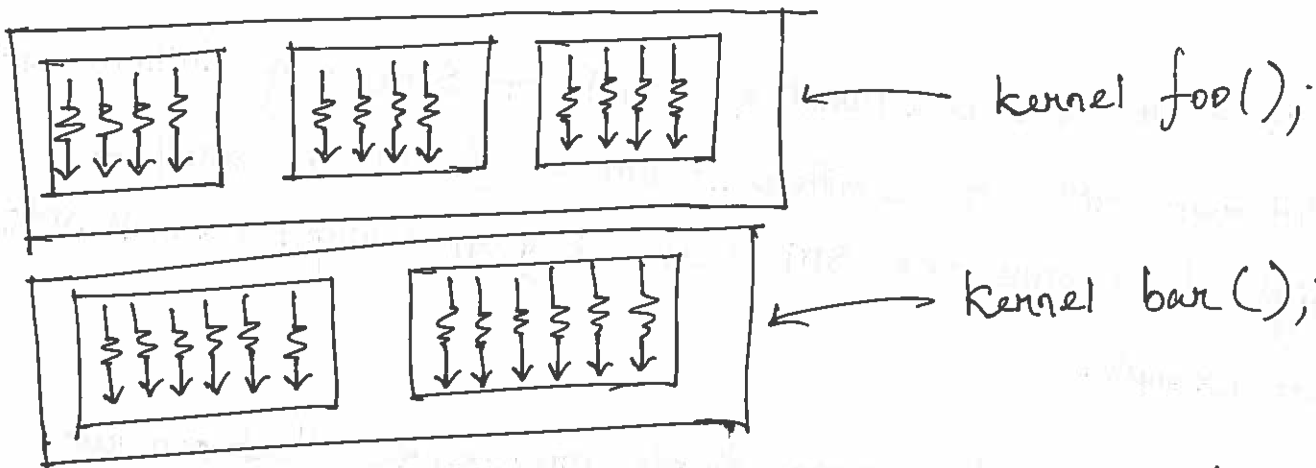
Summary of GPU Programming Model

kernels — c/c++ function

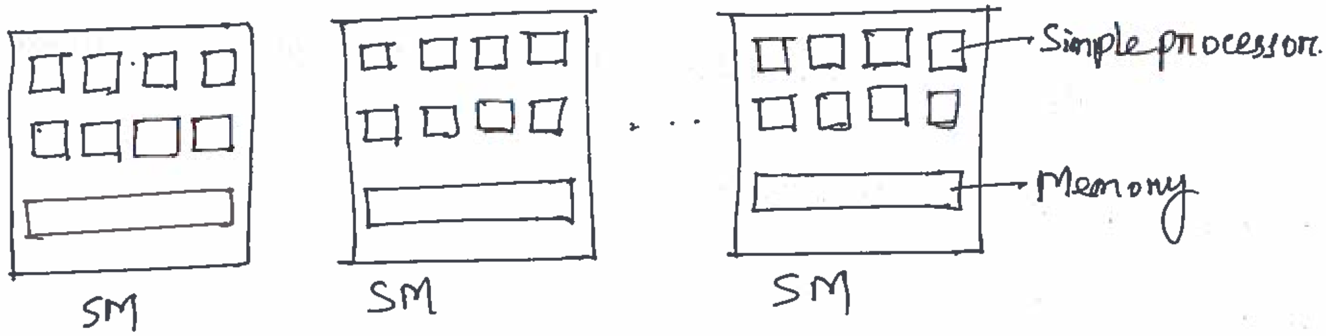


threads are shown using wiggly lines because they are not necessarily all going to take the same path through the code. There might be branches like if, switch statements, and loops, etc. So, different threads might take different paths.

The key thing about threads is that they come in thread blocks. A thread block is a group of threads that cooperate to solve a sub-problem.



A GPU program launches many threads to run one kernel, foo, and then they all run to completion and exit. The program launches many threads to run the next kernel like bar(). Here, for 2 different kernels, we run different number of thread blocks with different number of ~~kernels~~ threads. That is something the programmer can pick for each kernel.



A CUDA GPU is a bunch of SMs or Streaming Multiprocessors. Different GPUs have different number of SMs. A small GPU might have only one SM, but a big GPU might have 16 SMs for example.

An SM in turn has many simple processors that can run a bunch of parallel threads.

The important thing to understand is that the GPU is responsible for allocating blocks to SMs.

As a programmer, all you have to worry about is giving the GPU a big pile of thread blocks, and the GPU will take care of assigning them to run on the hardware SMs.

All the SMs run in parallel and independently.

<u>Some facts about thread blocks and SMs</u>

A thread block contains many threads.

An SM may run more than one block.

A block cannot be run on more than one SM.

All the threads in a threadblock may cooperate to solve a subproblem.

Since an SM may run more than one thread block and different thread blocks cannot cooperate, all the threads that run on a given SM might not be able to cooperate if they belong to different blocks.

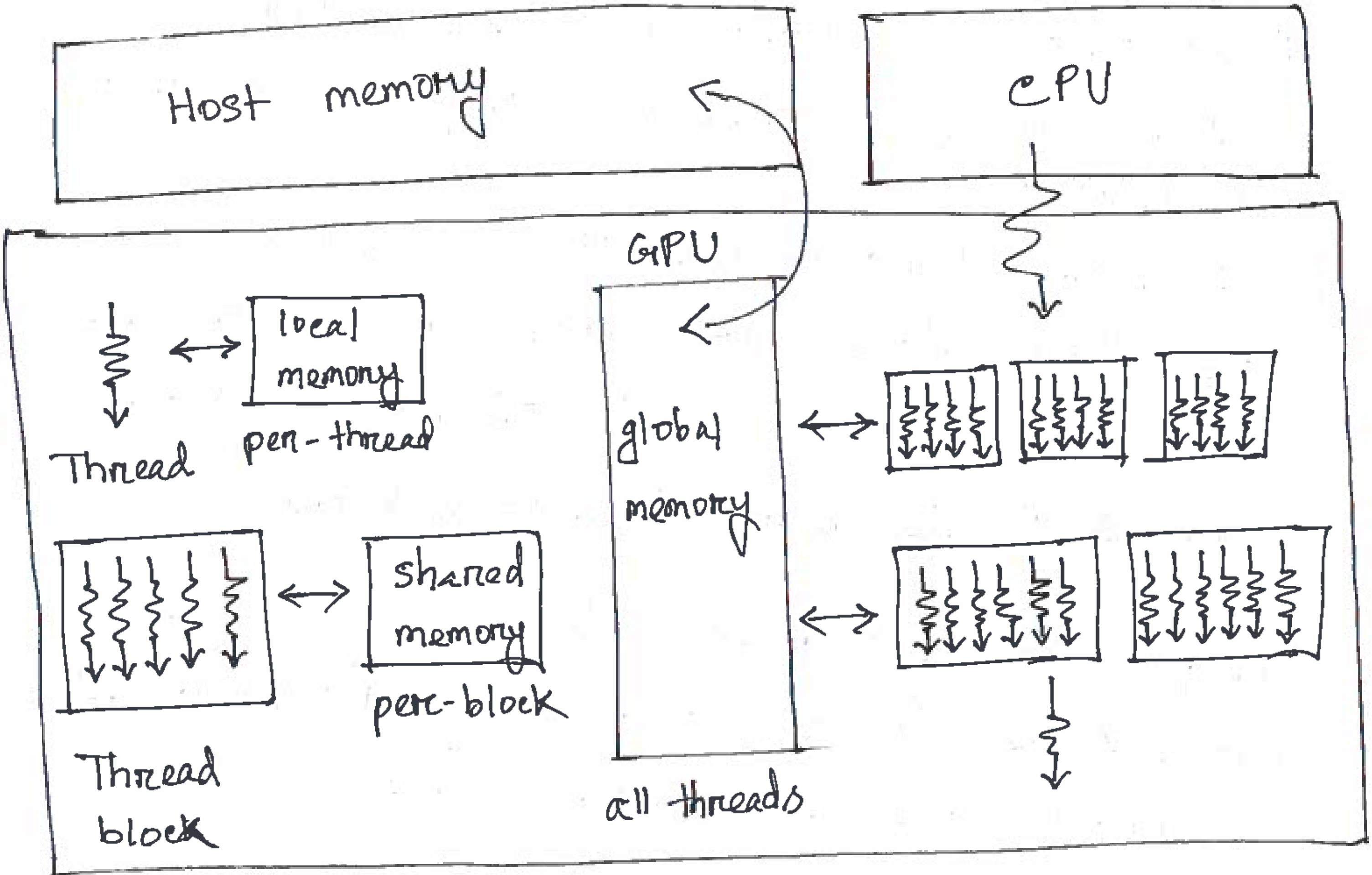
The programmer is responsible for defining the thread blocks in the software, whereas the GPU is responsible for allocating the thread blocks to hardware SMs.

<u>CUDA guarantees that</u>

all threads in a block run on the same SM at the same time.

all blocks in a kernel finish before any blocks from the next kernel run.

Memory Model



Every thread has access to its local memory, which is private to that thread, things like its local variables. So, the thread can read and write from local memory.

The threads in the thread block also have access to something called shared memory. All the threads in a thread block can read and write to per block shared memory. It is important to understand that shared memory is shared among the threads in a block. This is a small amount of memory that sits on the SM directly.

Finally, there is global memory. Every thread in the entire

system at any time can read and write to global memory.

The CPU thread launches the work on the GPU. The CPU has access to its own memory which in CUDA we call host memory. Usually data is copied to and from this host memory into the GPU's fast global memory before launching a kernel to work on that data.

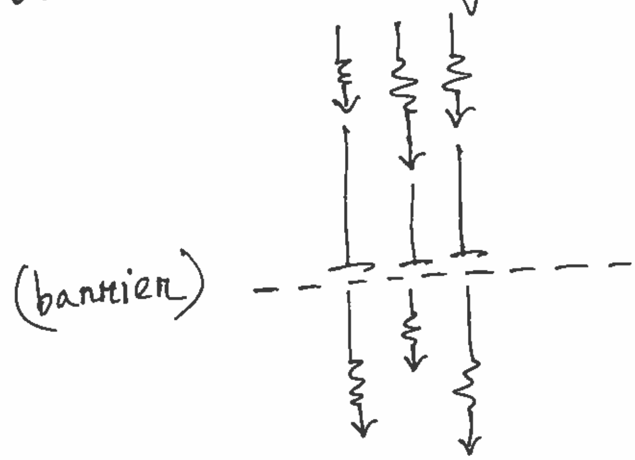
How the CUDA threads can access host memory directly will be discussed later.

Synchronization

Threads can access each other's results through shared and global memory, which means that they can work together. But, what if a thread reads a result before another thread writes it. So, we need synchronization among the threads.

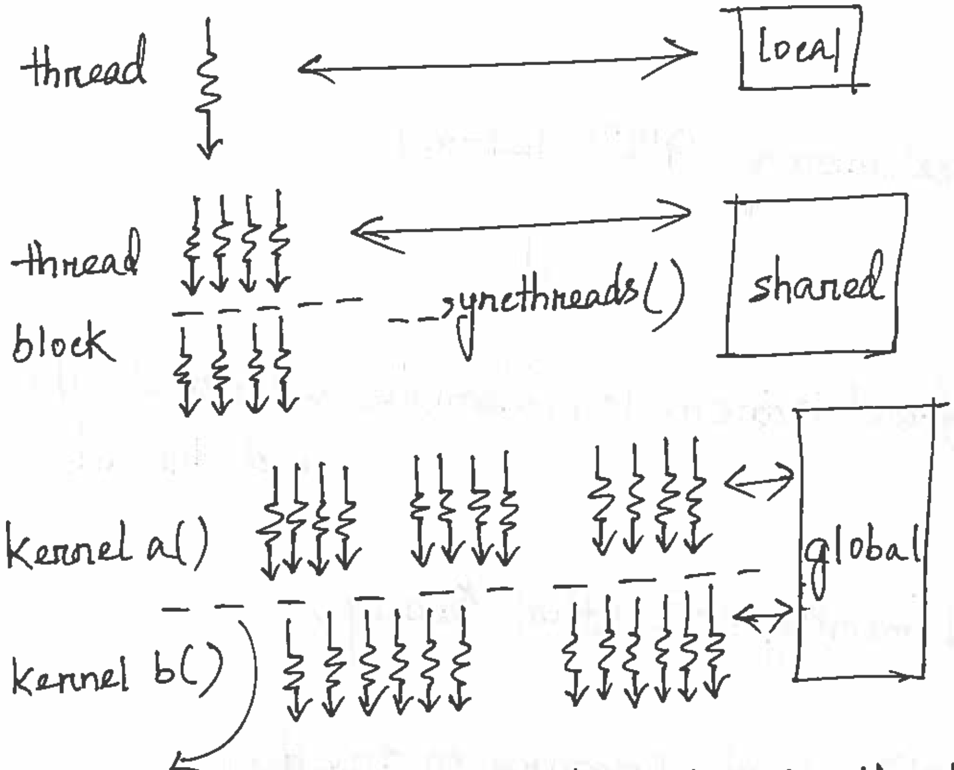
The simplest form of synchronization is called a barrier.

Definition of Barrier: A barrier is a point in the program where threads stop and wait for the rest of the threads. When all the threads have reached the barrier, they can proceed.



Example of synchronization

```
// Left-shift an array by one index
int idx = threadIdx.x ; // local variable
__shared__ int array [128]; //shared variable
// set array values by their indices
array [idx] = threadIdx.x;
__syncthreads ();
if (idx < 127) {
    int temp = array [idx+1];
    __syncthreads ();
    array [idx] = temp;
    __syncthreads ();
}
```



CUDA

is a hierarchy of

— computation that would be threads, thread blocks, and kernels

— with the corresponding hierarchy of memory spaces: local, shared, and global

— and synchronization primitives.

Implicit barrier between kernels. All the threads in one kernel completes before starting the execution of the next kernel.

Writing Efficient Programs

High-level strategies

① Maximize (arithmetic intensity $= \frac{\text{math}}{\text{memory}}$)

— maximize the number of compute operations per thread.

— minimize the time spent on memory accesses per thread.

Minimize the time spent on memory access

Move frequently accessed data to fast memory

local > shared >> global >> CPU "host"

↳ (similar to the registers or L1 cache)

Both the arguments and variables inside the kernels are local memory.

```
__global__ void use_local_memory_GPU(float in)
{
        float f;
        f = in; // both f and in are in local memory and private it
                                                          each thread
}

__global__ void use_global_memory_GPU(float *array)
{
  // array is a pointer (local) into global memory on the device
  array[threadIdx.x] = 2.0f * (float) threadIdx.x;
}
```

The point here is that since all the parameters to a function are local variables, and private to that thread, if you want to manipulate global memory, you have to pass in a pointer to that memory.

```
__global__ void use_shared_memory_GPU (float *array)
{
    //local variables, private to each thread
    int i, index = threadIdx.x;
    float average, sum = 0.0f;
    //shared variables are visible to all threads in the thread block
    // and have the same lifetime as the thread block
    __shared__ float sh_arr[128];
    sh_arr[index] = array[index];
    __syncthreads();
    for(i=0; i<index; ++i) {sum += sh_arr[i]; }
    average = sum/(index+1.0f); //average of all previous elements
    if (array[index]>average) {array[index] = average;}
}
```

<u>Example</u>

```
__global__ void foo (float *x, float*y, float *z)
{   __shared__ float a,b,c;
    float s,t,u;
      ⋮ ⋮ ⋮
    s= *x; // fast -(3)
    t = s;  // fast -(1)
    a=b;  // fast -(2)
    *y= *z; // fast -(4)
}
```