

Parallel Communication Patterns

Map - one to one correspondence between input and output

Gather(many to one) - Gather data from multiple locations and then write it to one memory location(convolution)

Scatter(one to many) - Data from a single location is written to multiple locations

(Several threads attempt to write at the same place at the same time) - may cause problems

Stencil(several to one) - Data Reuse - Many Thread accessing and using the same data

Transpose - Reorder Data elements in Memory

Reduce

Sort/Scan

Sharing Memory//Safely - GPU Hardware -

Summary of Programming Model -

Kernels - C/C++ function - Performed by many threads

Each kernel may have different number of threads per thread blocks

Thread Blocks and GPU Hardware -

- CUDA GPU - Streaming Multiprocessors (SM)
- SM -> Simple Processors/Memory
- GPU is responsible for allocating blocks to SMs
- All the SMs run in parallel and independently.
- A SM may run multiple blocks but A block cannot be run on multiple SMs
- Thread in different thread blocks must not cooperate with each other.
- Threads in a single thread block may cooperate with each other
- Programmer cannot specify any order of execution for the thread blocks.

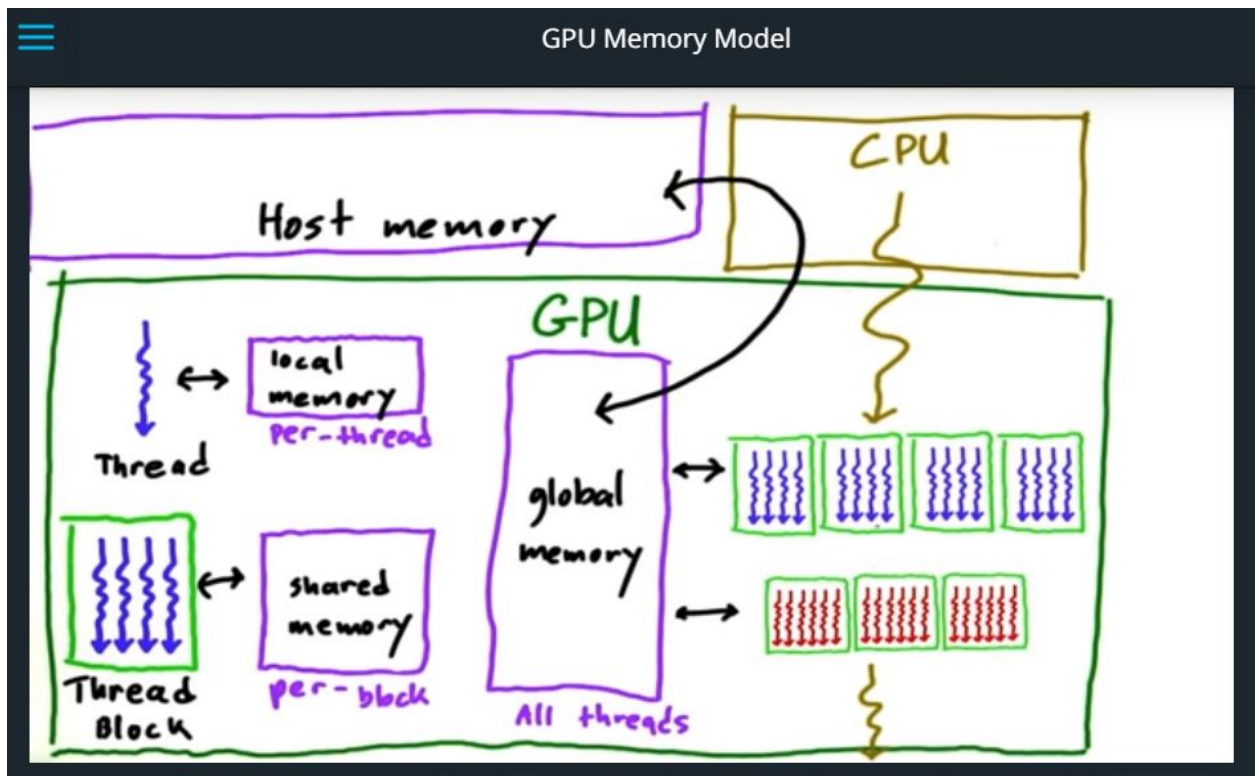
- No assumptions about what block will run on which SM (when and where)
 - Helps in scalability (adding multiple SM's)
- No communication between blocks - "DeadLocks"
- // force the printf(s) to flush
- `cudaDeviceSynchronize();`

When and Where - CUDA guarantees -

- 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 -

1. Local Memory - local variables/parameters
2. Shared Memory
3. Global Memory



Synchronization -

Threads need to synchronize. To avoid dirty read.

Barrier - Point in the program where all the threads stop and wait for others, and when all the threads have arrived at the barrier then they move further.

Use `__syncthreads()` to put barrier - sync threads within a block.. Why?



```
int idx = threadIdx.x;
--shared-- int array[128];
→ array[idx] = threadIdx.x;
--sync threads();
if (idx < 127) {
    int temp = array[idx+1];
    --sync threads();
    array[idx] = temp;
}
--sync threads();
```

CUDA a hierarchy of

- Computation
- Memory Spaces
- synchronization

Minimize time spent on memory -

1. Move frequently-accessed data to fast memory

Local > shared >> global >> CPU(host) (In terms of speed)

Local - In register or in L2 cache

Convention - h_ -> host

D_ -> device

Global Memory

```
25 }
26
27
62
63 int main(int argc, char **argv)
64 {
65     /*
66      * First, call a kernel that shows using local memory
67      */
68     use_local_memory_GPU<<<1, 128>>>(2.0f);
69
70     /*
71      * Next, call a kernel that shows using global memory
72      */
73     float h_arr[128]; // convention: h_ variables live on host
74     float *d_arr;     // convention: d_ variables live on device (GPU global mem)
75
76     // allocate global memory on the device, place result in "d_arr"
77     cudaMalloc((void **) &d_arr, sizeof(float) * 128);
78     // now copy data from host memory "h_arr" to device memory "d_arr"
79     cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
80     // launch the kernel (1 block of 128 threads)
81     use_global_memory_GPU<<<1, 128>>>(d_arr); // modifies the contents of array at d_arr
82     // copy the modified array back to the host, overwriting contents of h_arr
83     cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyDeviceToHost);
84     // ... do other stuff ...
85
86
95     return 0;
96 }
```

- Coalesce global memory accesses
 - GPU most efficient when threads read or write contiguous memory locations.

Atomics - Read Guide for different atomic functions `atomicAdd(ptr, val)` for example.

Limitations -

- Only certain operations, data types(integers)
- Implement any atomic using CAS()
- Still no ordering constraints. (Non associativity of floating points)
- Serializes access to memory
- Atomics take time!

Thread Divergence -

When kernel have if statements / loops.

In loops some threads just sit around without doing anything.

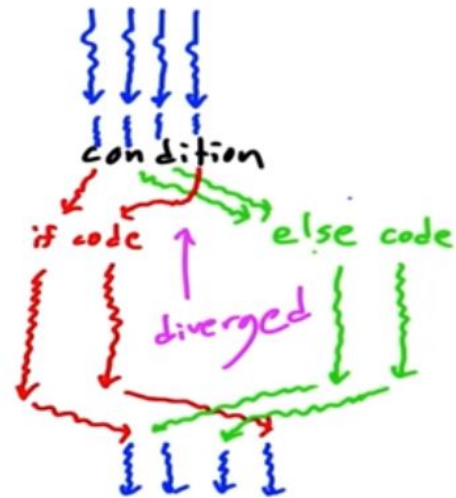
Loop divergence slows the execution down.

2. Avoid thread divergence

```

...
if ( condition )
{
    some code
}
else
{
    some other code
}
...

```

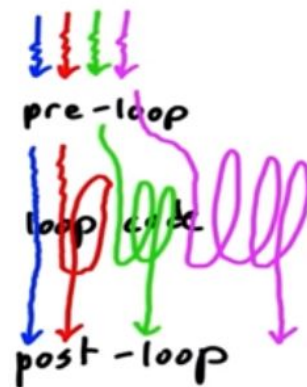
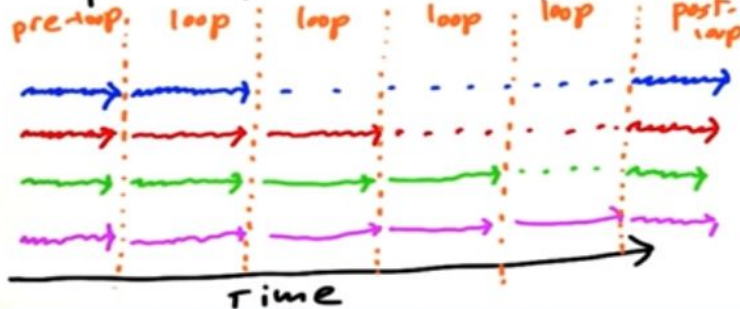


Thread Divergence

```

:
for (int i=0; i<= threadIdx; ++i)
{
    ... some loop code ...
}
:
post-loop code

```





Summary

- Communication patterns
gather, scatter, stencil, transpose
- GPU hardware & programming model
SMs, threads, blocks, ordering
Synchronization
Memory model - local, global, shared, atomics
- Efficient GPU programming
 - Access memory faster
 - Avoid thread divergence
 - coalescing global mem
 - use faster memory