Introduction to Parallel Programming (CS344)

By Erik Pärlstrand

Lesson 1: The GPU Programming Model

Assume that we want to build faster processors. The following options are available:

1. Run with a faster clock speed (i.e. shorter amount of time on each step of the computation).
2. Do more work per clock cycle.
3. Add more processors.

In this course we will take approach number three and in particular we are interested in GPU (Graphical Processing Unit). Recent trends is that the clock rates for transistors does not increase and therefore need to increase the number of cores.

Latency versus throughput

- CPUs are low latency, low throughput.
- GPUs are higher latency, high throughput.

By latency we mean "time to respond" while throughput (i.e. bandwidth) is computation per time unit. An analogy is that the CPU is a Ferrari, the GPU is a bus and we want to move people from one location to another.

GPU design tenets

1. Many simple compute units, little control logic.
2. Explicitly parallel programming model (unlike many CPU-compiler-toolchains that makes the hardware details a bit more opaque, especially across different CPU hardware).
3. Optimize for throughput instead of latency.

CUDA

CUDA is a parallel computing platform created by Nvidia. In CUDA one usually differs between:

- Host: part of program runs on the CPU.
- Device: part of program that runs on the GPU.
- Assume host and device have separate memory to store data.

Note that the CPU is "in charge", i.e. it's responsible for:

- Move data from CPU memory to GPU.
- Move data from GPU back to CPU.

- Allocate GPU memory.
- Launch kernel on GPU .

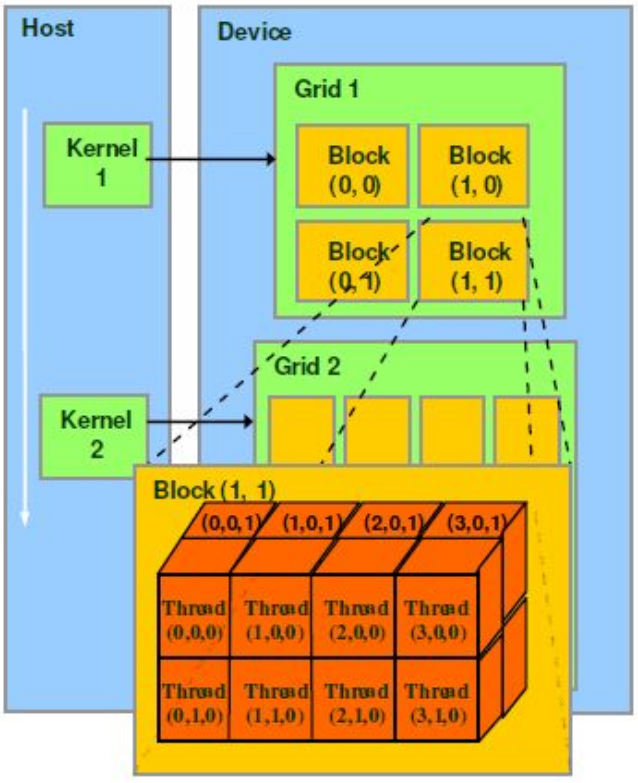
In CUDA we write kernels (a function to run on GPU) as a serial program. Parallelism is introduced by the CPU telling the GPU to launch multiple instances (i.e. threads) of the kernel. GPU's are good at launching a lot of threads and running them in parallel.

Threads and blocks

The syntax for specifying blocks of threads in CUDA is <<<B, T>>>, where B is the number of blocks and T is the number of threads. There is a maximum number of threads per block (512 for older GPUs, 1024 for newer).

For example, in order to launch 128 threads we can either launch 2 blocks with 64 threads each or 1 block with 128 threads. Note that each thread knows the index of its block and thread.

In the previous example we used one dimensional block/threads, however we can define them in three dimensions, i.e. B = dim3 (x,y,z) and T = dim3 (x,y,z). Below is an example of the organization of threads within a block:

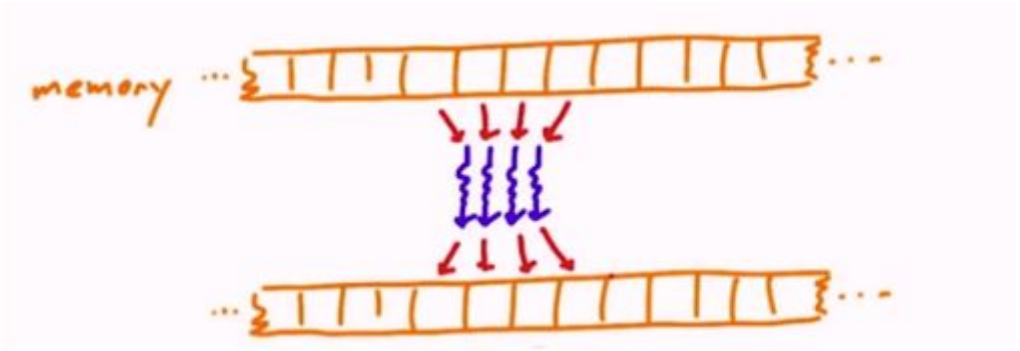


Lesson 2: GPU Hardware and Parallel Communication Patterns

In parallel computing we are using multiple threads together in order to solve various problems. Therefore, communication between threads are important. Below are some common communication patterns in parallel computing:

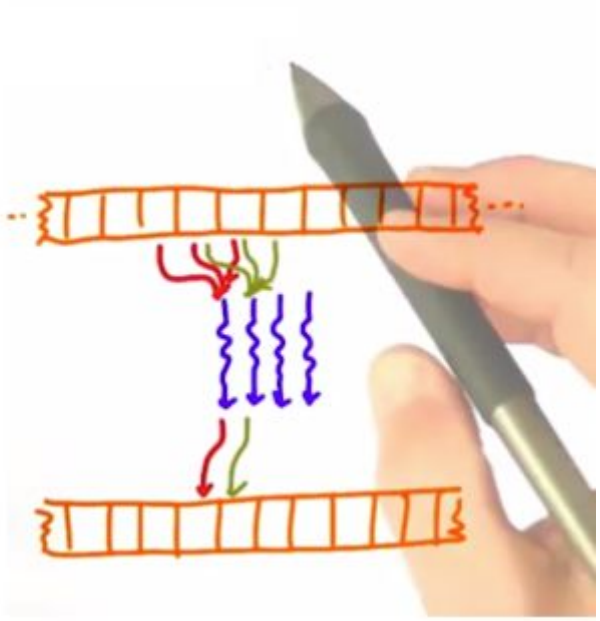
Map

Map is a task that reads/writes to a specific data element. Below is an illustration of the process:



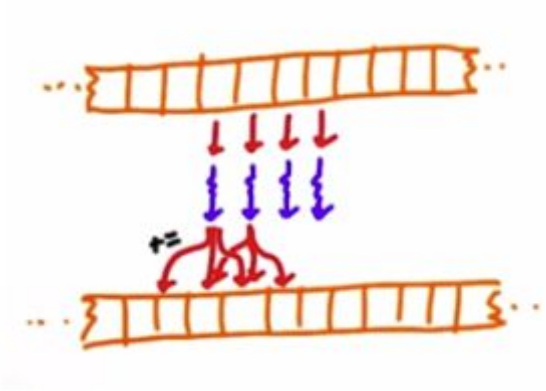
Gather

Gather is a task that read from multiple elements and writes to one element. Below is an illustration of the process:



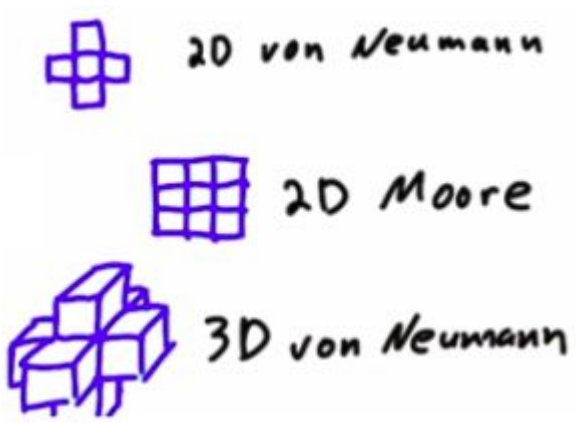
Scatter

Scatter is a task that read from one element and writes to multiple elements. Below is an illustration of the process:



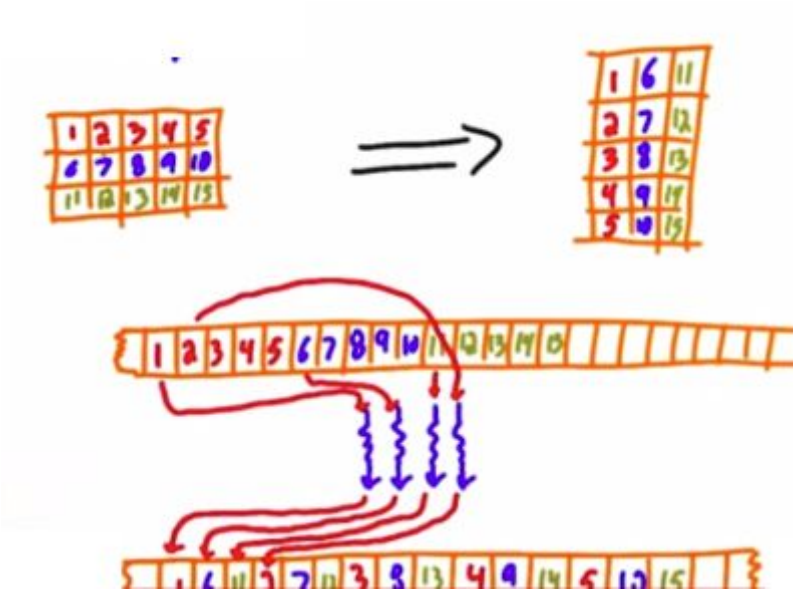
Stencil

Stencil is a task that reads from a fixed neighborhood of an element and writes to it. Below is an illustration of different type of stencils:



Transpose

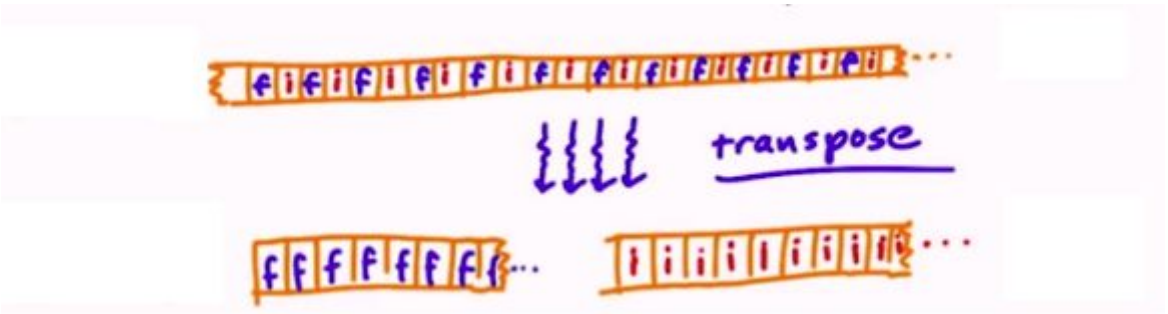
Transpose is a task that rearrange an array. One can transpose an array, matrix and image. Below is an example of the rearranging:



A common application of transpose is to rearrange structs. Assume that we have a struct of the following format:

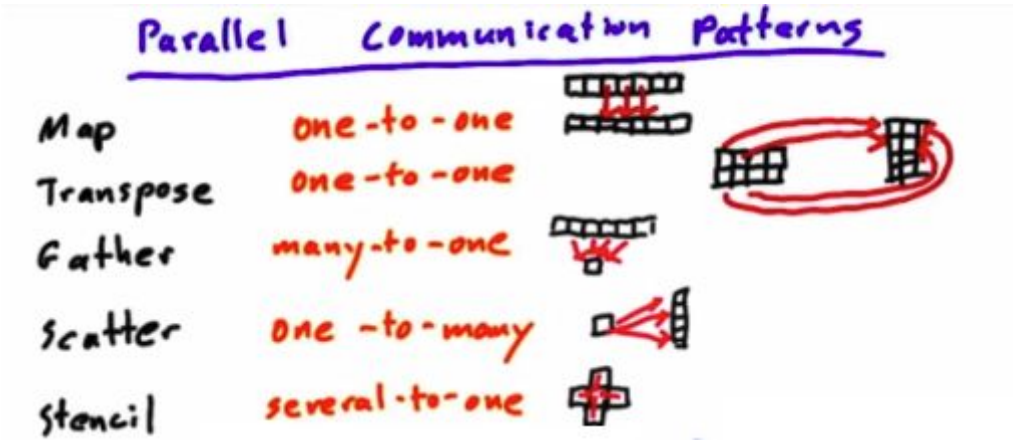
```
struct A {
   float f;
   int i;
};
```

If we would store this struct as above it would be called an Array Of Structures (AOS). This is illustrated as the first figure below. Sometimes, we would like to arrange similar type together, i.e. Structures Of Array (SOA). This is illustrated in the second figure below:



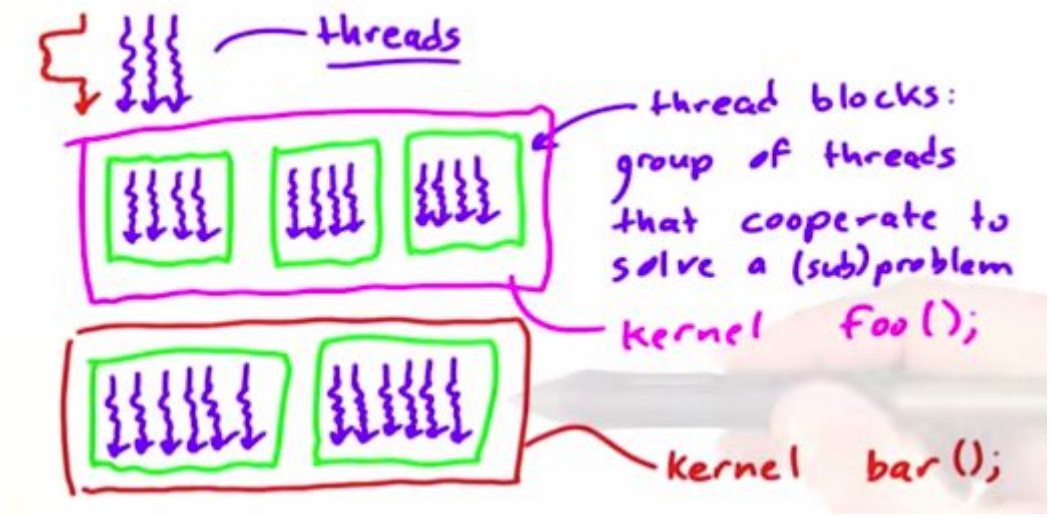
Recap of communication patterns

To recap, the following figure describes the different parallel communication patterns:

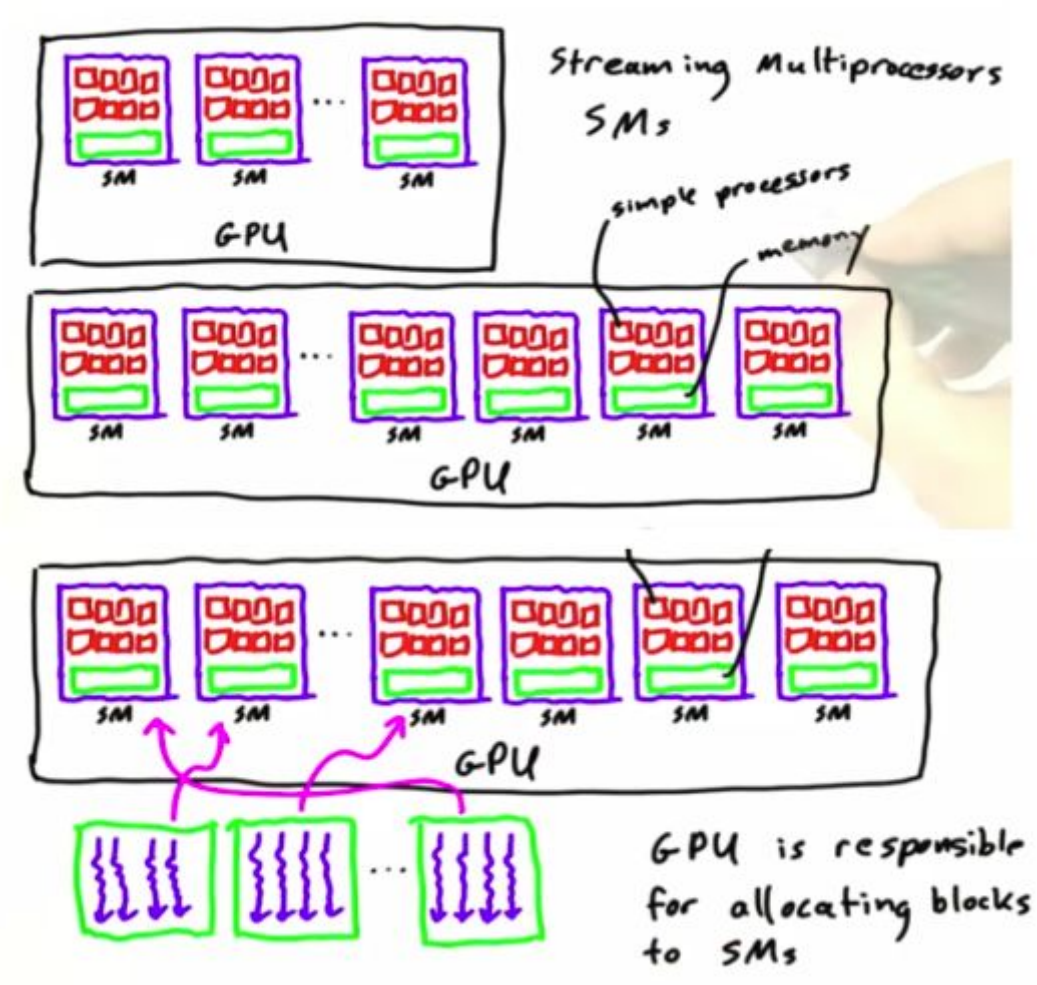


Summary of programming model

The kernel is a C/C++ function that is being run on the GPU. The function is being run on different block of threads. In CUDA, a kernel function is defined with the syntax __global__. Thread blocks are a group of threads that cooperate to solve a sub-problem.



The GPU consist of numerous streaming multiprocessors (SMs). They in turn consist of multiple simple processors and has its own memory. A thread block is being run on one of these SMs. Note that the GPU is responsible for allocating blocks to SMs.



CUDA makes few guarantees about when and where thread blocks will be runned. The advantage of this approach is:

- Hardware can run things efficiently.
- No waiting on slow blocks.
- Scalability; the same code can be runned on a cell phone or a super computer, the only difference is the number of SMs.

However there are some disadvantages:

- One cannot make assumptions about which SM will run which block.
- There is no communication between blocks.
- Threads and blocks must complete.

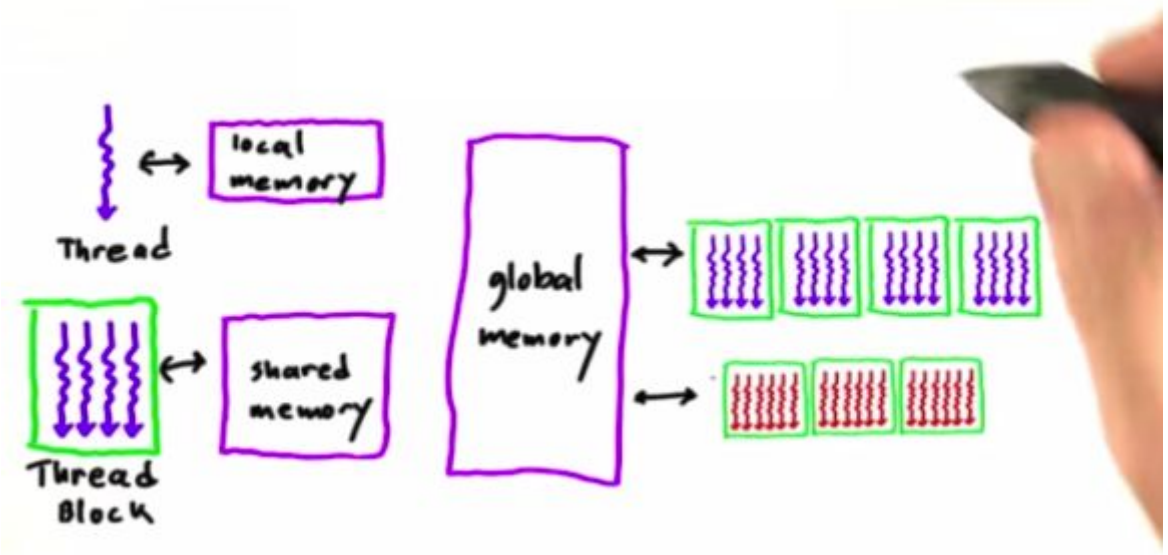
The following is guaranteed by CUDA:

- 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 is runned.

Memory model

There are three different types of memory in CUDA:

- Global memory - accessible by all kernels/blocks/threads.
- Shared memory - accessible by a given thread block.
- Local memory - accessible by a given thread.



Synchronization

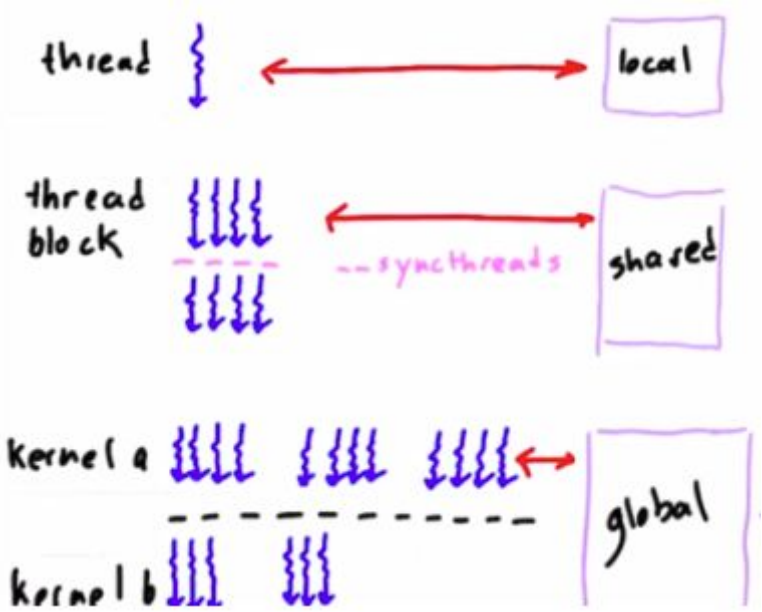
Threads can access each other's results through shared and global memory. This allows them to work together. However, there is a danger if a thread reads a result before another thread writes it. Therefore, for some applications the threads need to synchronize.

Barrier

Point in the program where threads stop and wait. When all threads have reach the barrier they can proceed. In CUDA this is done with the syntax __syncthreads().

In essence, CUDA is a hierarchy of:

- Computation.
- Memory spaces.
- Synchronizations.



Writing efficient programs

High level strategies:

- Maximize arithmetic intensity, i.e.
    - Maximize computing operations per thread.
    - Minimize time spent on memory per thread.
        - Put data in fast memory.
        - Use coalesced global memory access.
- Avoid thread divergence.

Put data in fast memory

A rule of thumb for the time to access data from memory for a GPU:

local < shared << global << CPU

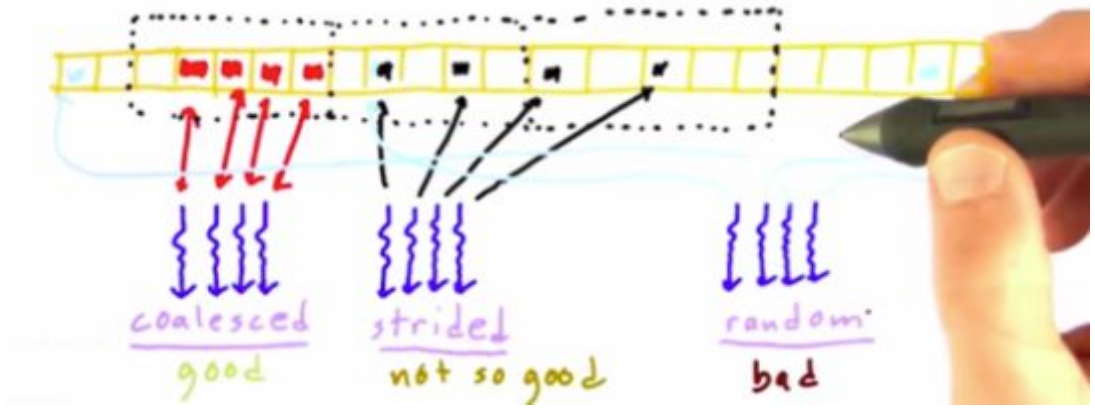
Therefore we would like to move frequently-accessed data to fast memory.

Use coalesced global memory access

The GPU is most efficient when threads reads and writes contiguous memory location. There are three types of global memory access:

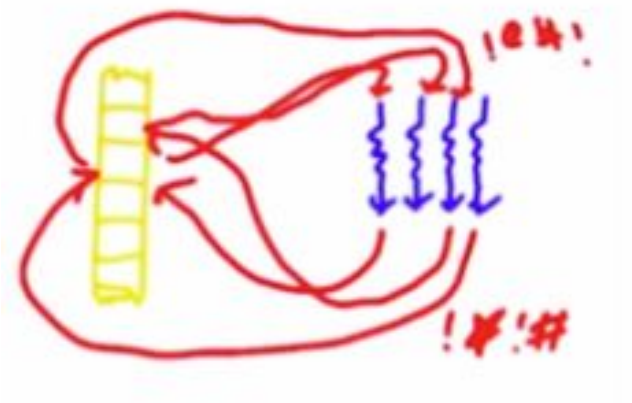
- Coalesced: The reads/writes are in the same “neighborhood”.
- Strided: The reads/writes are in the same “neighborhood” with some space between them.
- Random: The reads/writes are from random locations.

Below is an example of the three types:



Atomic

Assume that lots of threads reads and writes to the same memory locations. This will lead to a race-condition. This can be solved with CUDA using the atomic operation, e.g. atomicAdd().



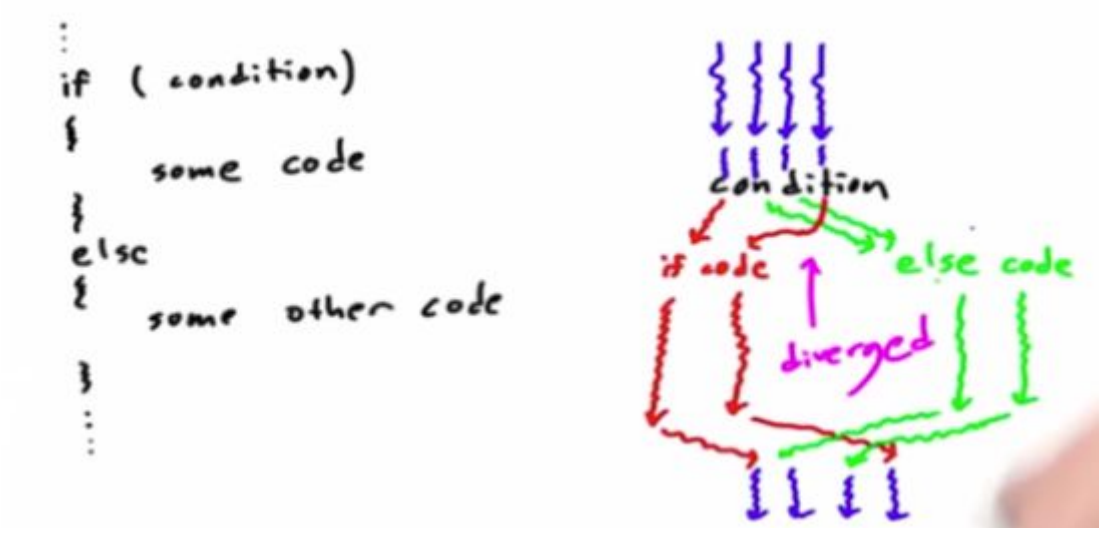
However there are some limitation with atomics:

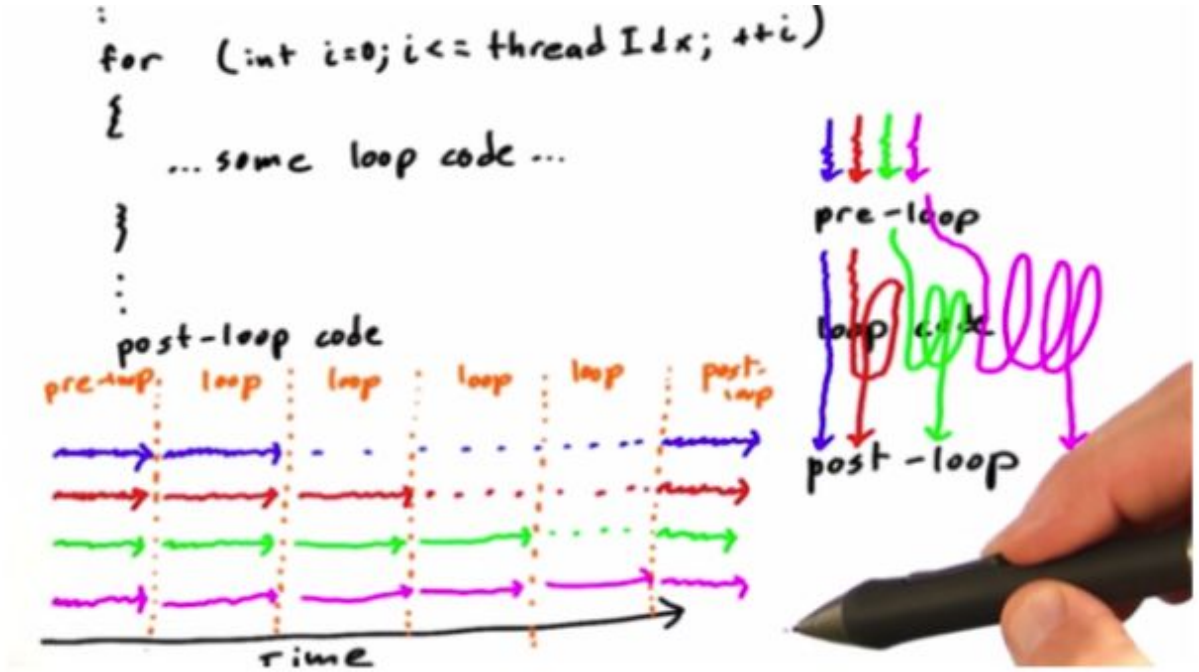
- Only certain operations and data types are supported (mostly integers). However a work around is to use CAS (Compare And Swap).
- No ordering contains.
- Seralizes access to memory (i.e. leads to slow operations).

Thread divergence

Avoid (if possible) threads doing “different operations”, e.g. if thread id is larger than 10 do … else …

Thread divergence can be introduced by if statements or with for loops, see example below:




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