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 .

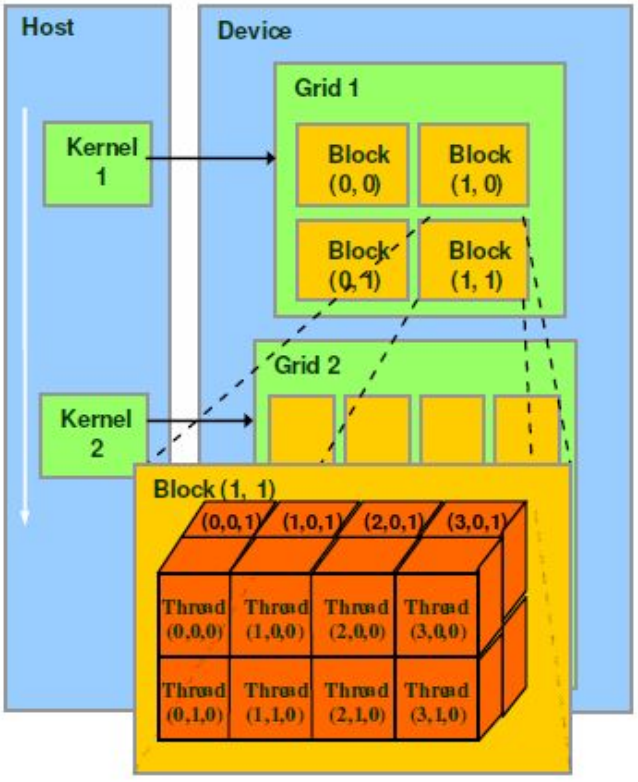
In CUDA we write kernels (a function to run on GPU) as a serial program. Parallelism is introduced by the CPU telling the GPU to launch multiple instances (i.e. threads) of the kernel. GPU's are good at launching a lot of threads and running them in parallel.

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:

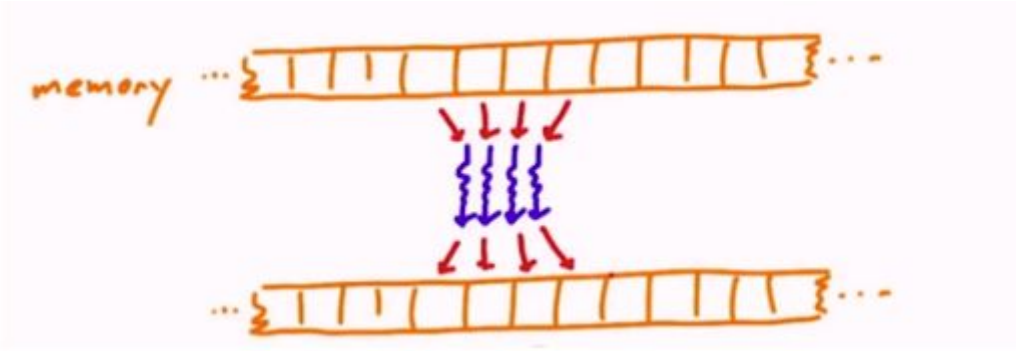


Lesson 2: GPU Hardware and Parallel Communication Patterns

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:

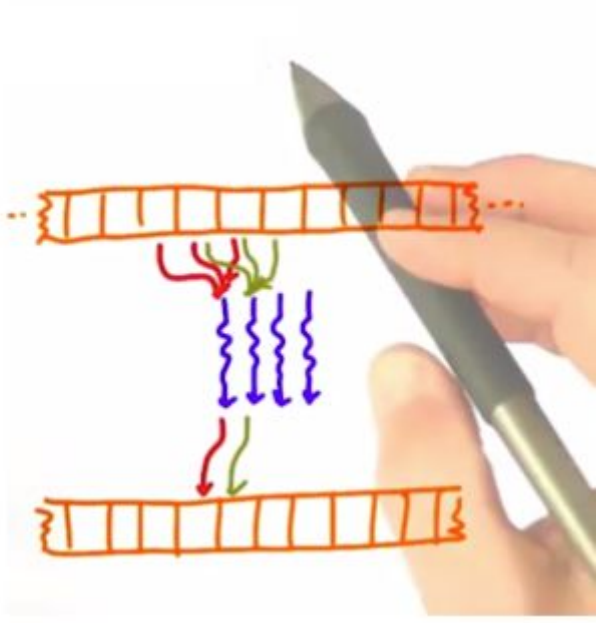
Map

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



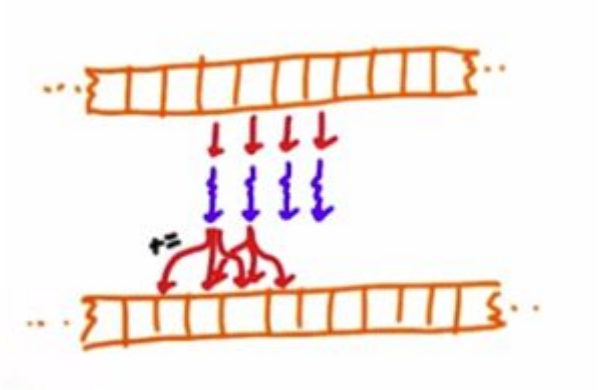
Gather

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



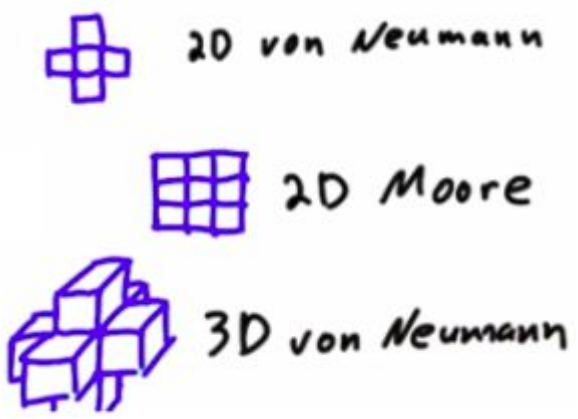
Scatter

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



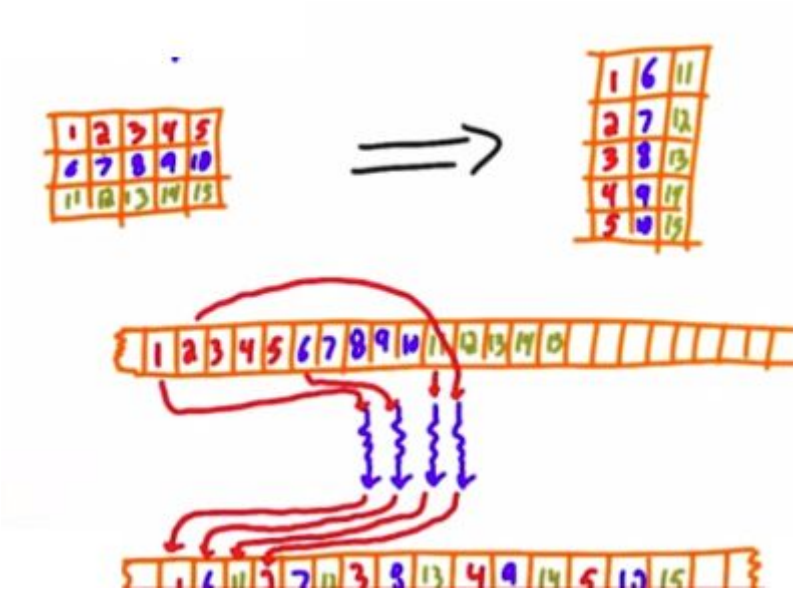
Stencil

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:



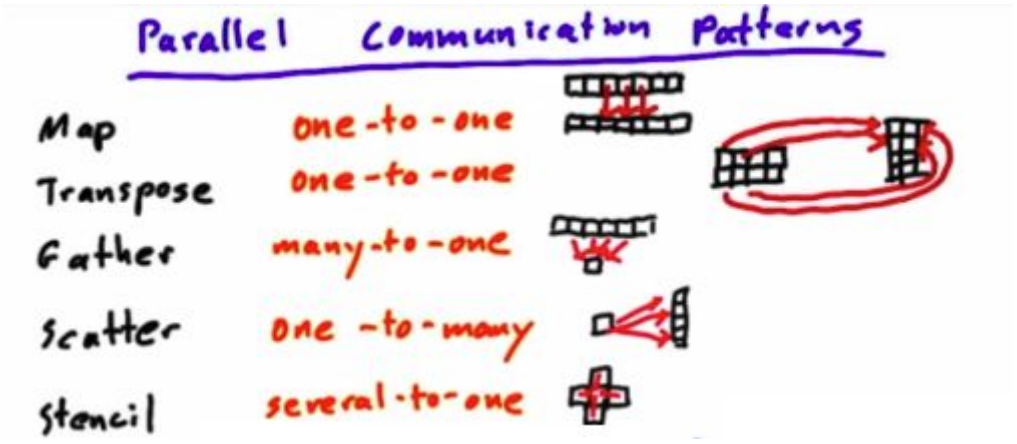
A common application of transpose is to rearrange structs. Assume that we have a struct of the following format:

```
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:

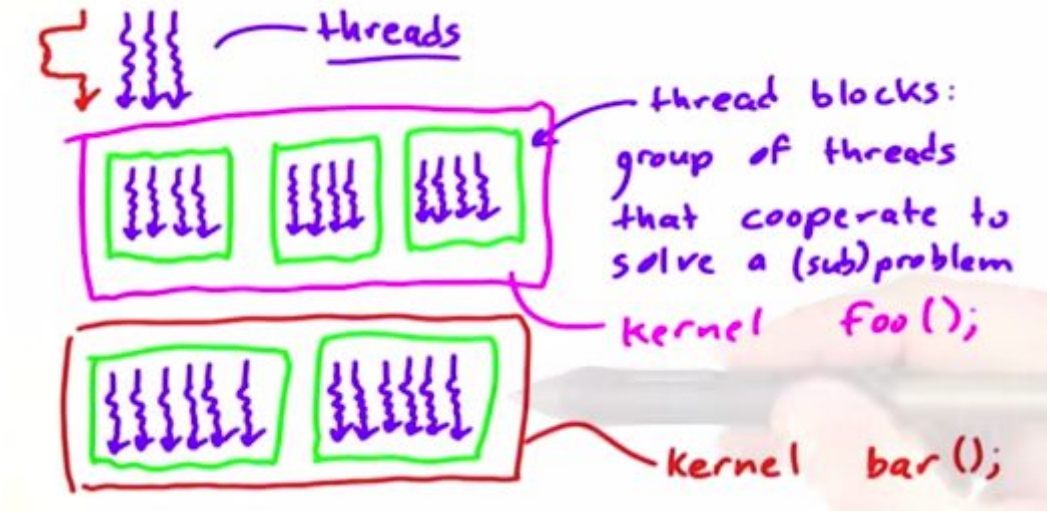
Recap of communication patterns

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

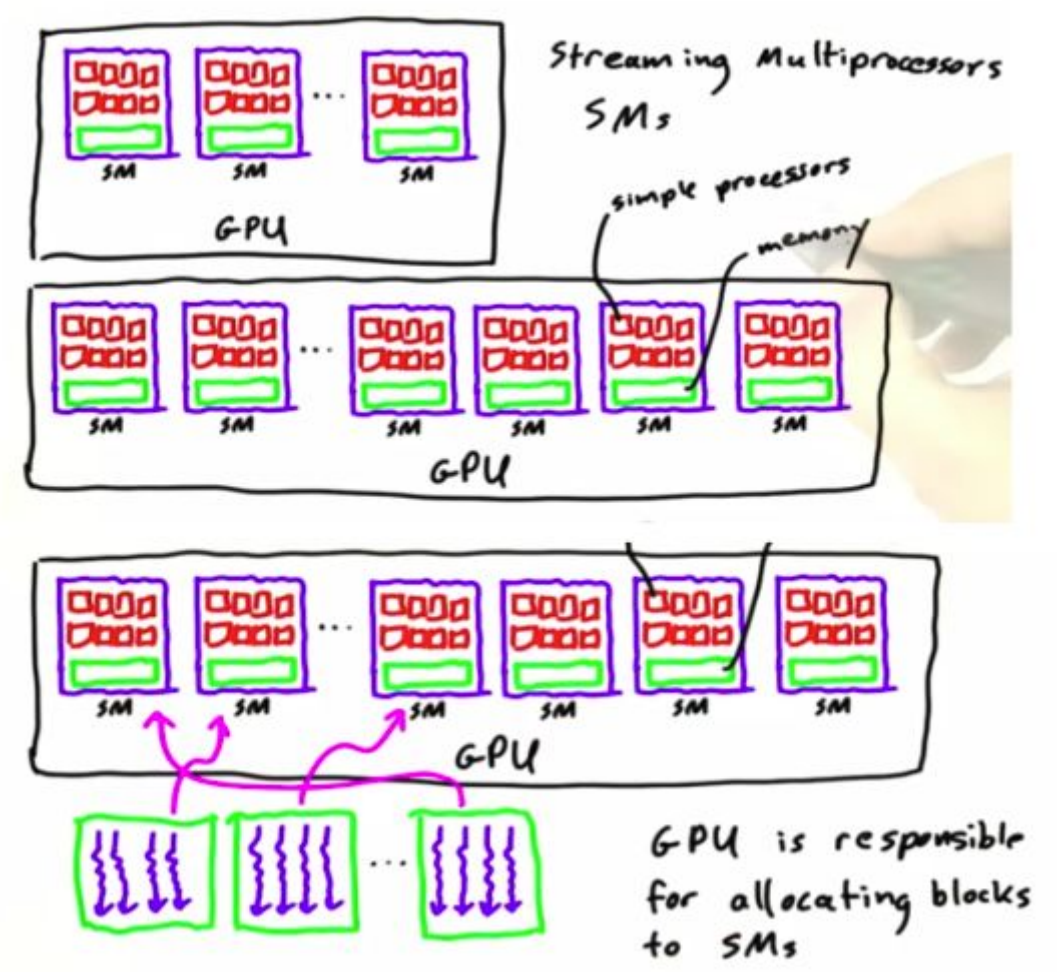


Summary of programming model

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.

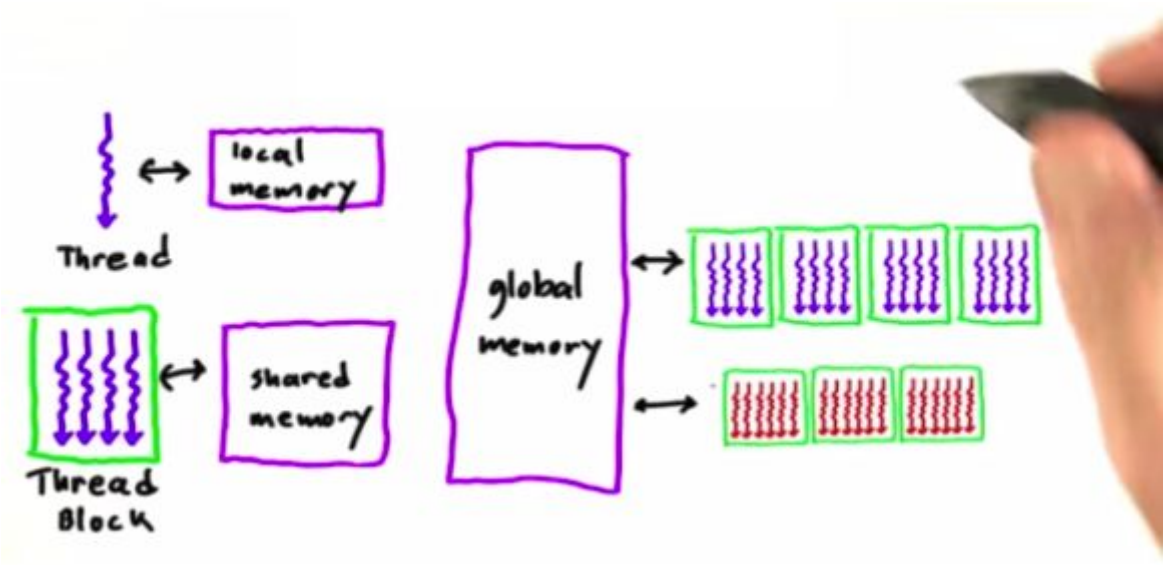
The following is guaranteed by CUDA:

- 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

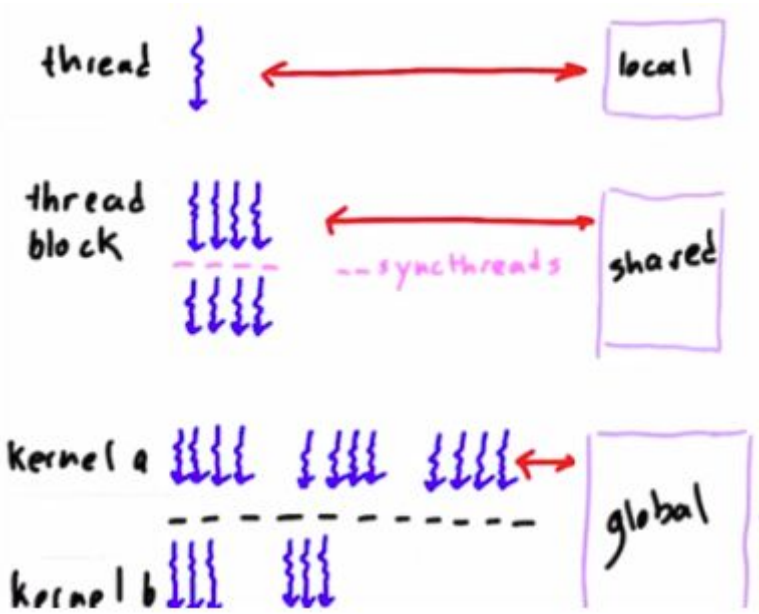
Threads can access each other's results through shared and global memory. This allows them to work together. However, there is a danger if a thread reads a result before another thread writes it. Therefore, for some applications the threads need to synchronize.

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

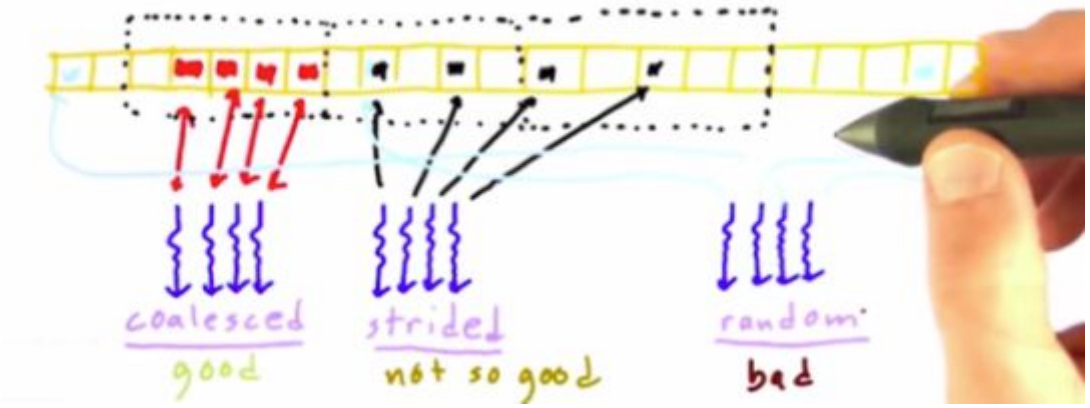
Therefore we would like to move frequently-accessed data to fast memory.

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:

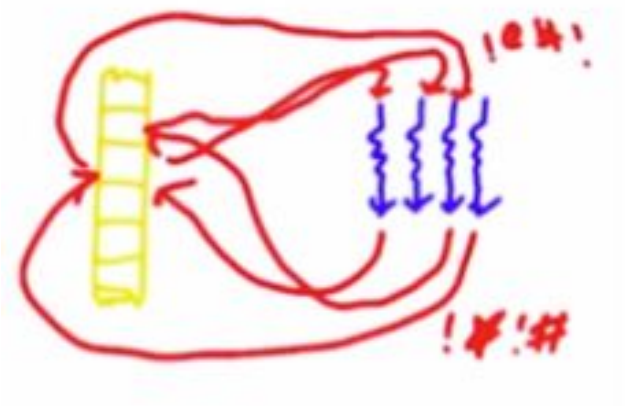
- Coalesced: The reads/writes are in the same “neighborhood”.
- Strided: The reads/writes are in the same “neighborhood” with some space between them.
- Random: The reads/writes are from random locations.

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().



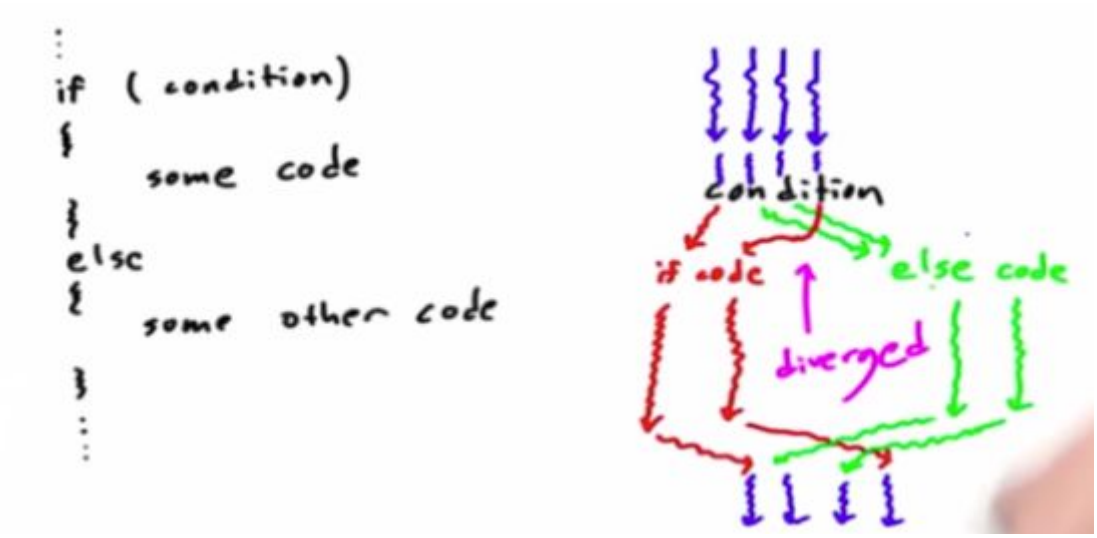
However there are some limitation with atomics:

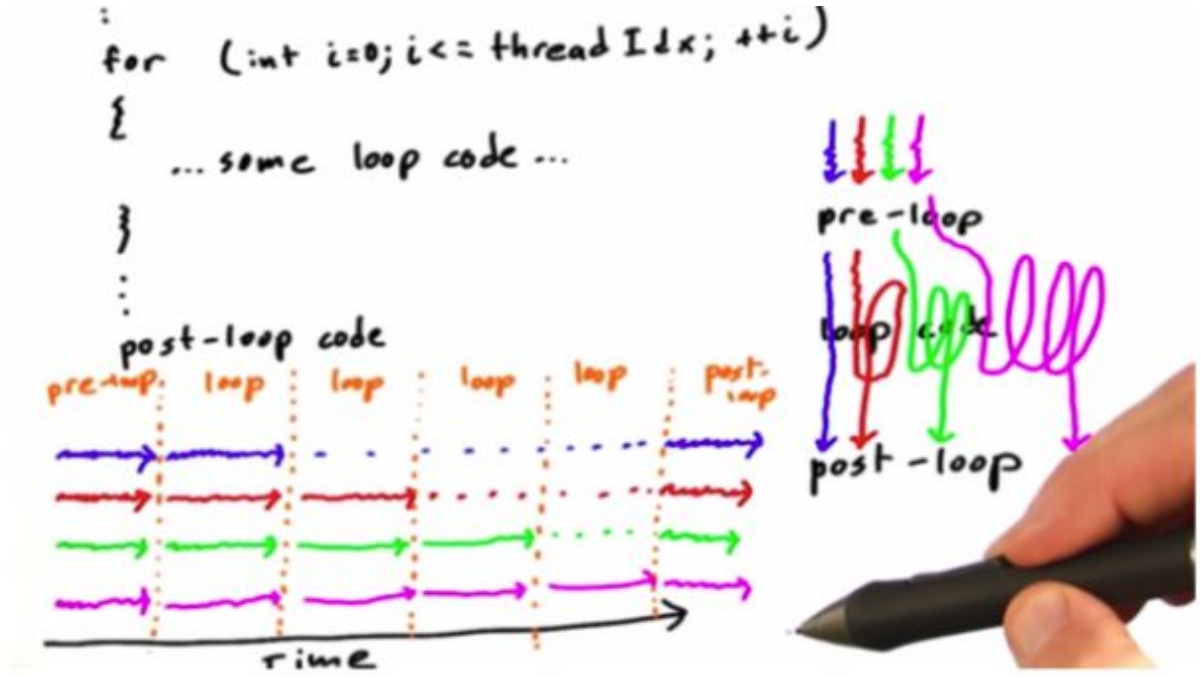
- Only certain operations and data types are supported (mostly integers). However a work around is to use CAS (Compare And Swap).
- No ordering contains.
- Seralizes access to memory (i.e. leads to slow operations).

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:



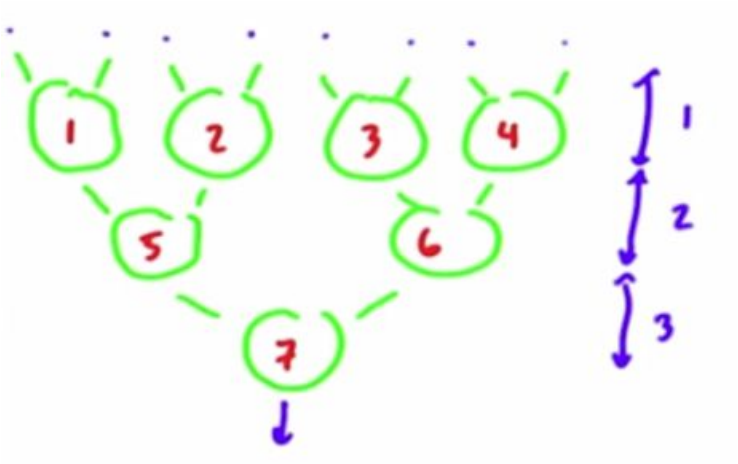


Lesson 3: Fundamental GPU algorithms (Reduce, Scan, Histogram)

In parallel programming we are interested in two types of complexity; step complexity and work complexity.

Work complexity means the total number of primitive operations that the algorithm performs. Step complexity means the “length” of the series of operations that have to be performed sequentially due to data dependencies.

In the below example, the work complexity is 7 and the step complexity is 3.



Reduce

A reduction combines all the elements in a collection into one element using a associate two-input, one output operation. The operator ($\oplus$) needs to be:

- Associate (i.e. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$).
- Binary (i.e. two inputs one output).

Some example of these operations are addition, multiplication, logical operators (and, or etc) and min/max.

One naive approach would be to write a serial algorithm, i.e:

```
float sum = 0.0f;

for(int i=0; i < length; i++){
   sum += array[i];
}
```

This approach has a work complexity of O(n) and step complexity of O(n). Below is an illustration of the setup:

Fortentually, reduce can be implemented as a parallel algorithm by splitting the operations into a “binary tree structure” as illustrated below. This approach has a work complexity of O(n) and a step complexity of O(log(n)). Below is an example:

Scan

The scan operation takes a operator $\oplus$ with identity I, and an array of n elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the ordered set $[I, a_0, (a_o \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$.

The operator needs to be:

- Associate (i.e. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$).
- Binary (i.e. two inputs one output).
- Have a identity element (i.e. $a \oplus I = a$).

Some example of these operators are addition (I=0), multiplication (I=1), max (I=0) etc. An example is

- Input: [1, 2, 3, 4]
- operator: addition
- Output: [0, 1, 3, 6]

This is an example of exclusive scan. We can also use an inclusive scan, i.e. the output is $[a_0, (a_o \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$. This gives:

- Input: [1, 2, 3, 4]
- operator: addition
- Output: [1, 3, 6, 10]

A serial implementation of (inclusive) scan is:

```
int acc = identity;

for(int i=0; i < element.length(); i++){
   acc = acc ⊕ element[i];
   out[i] = acc;
}
```

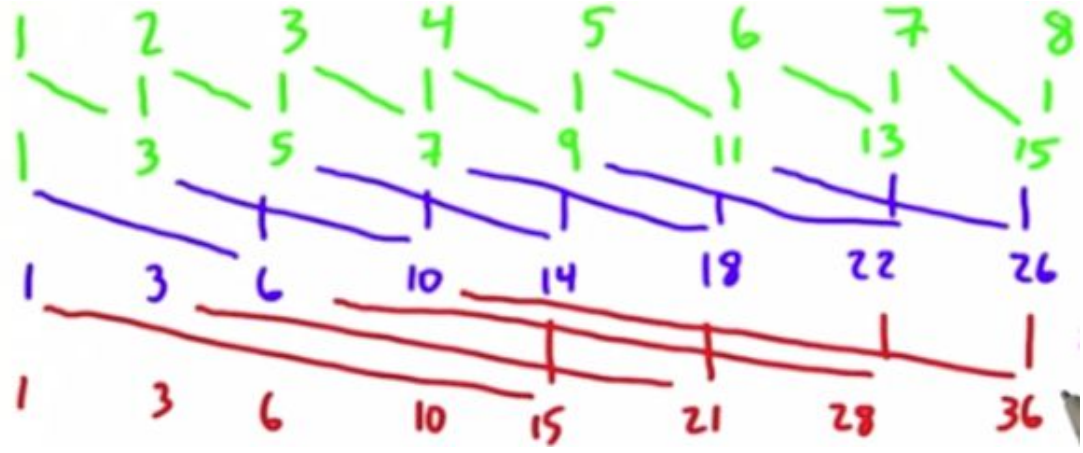
This would have a work complexity of O(n) and a step complexity of O(n).

Hillis & Steele scan

Hillis & Steele scan is a parallel algorithm implementing scan. Assuming that we do a sum scan, the algorithm follows:

for $i \leftarrow 0$ **to** $\lceil \log_2 n \rceil - 1$ **do**
  for $j \leftarrow 0$ **to** $n - 1$ **do in parallel**
    if $j < 2^i$ **then**
      $x_j^{i+1} \leftarrow x_j^i$
    else
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

where x_i^j is the j-th element of the array x in timestep i. It has a step complexity of O(log n) and a work complexity of O(n log n). Assume that we want to do a inclusive sum scan with the input [1, 2, 3, 4, 5, 6, 7, 8]. The Hillis & scan algorithm follows:



Blelloch scan

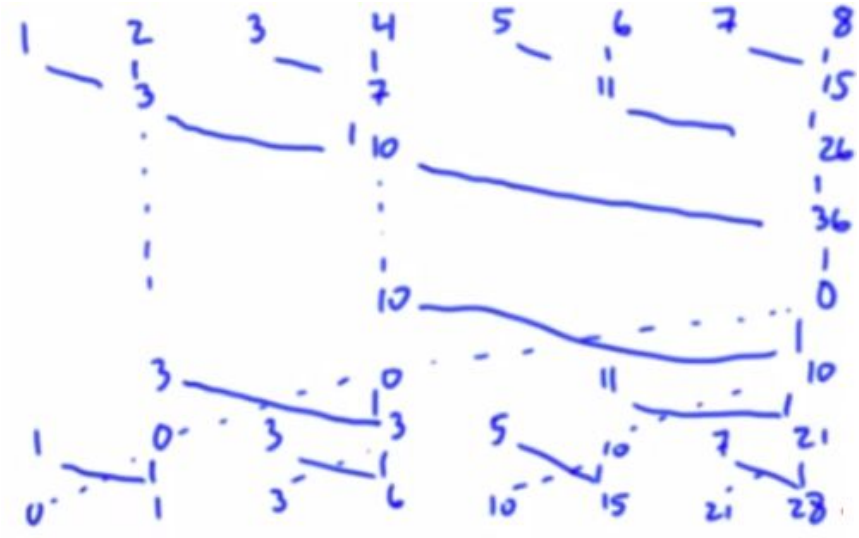
Blelloch scan is another parallel algorithm implementing scan. It has two steps, one up-sweep (reduce) step and a down-sweep step. The up-sweep follows:

for $d = 0$ to $\log_2 n - 1$ **do**
  for all $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
    $x[k + 2^{d+1} - 1] = x[k + 2^d - 1] + x[k + 2^d + 1 - 1]$

And the down-sweep follows:

$x[n-1] \leftarrow 0$
for $d = \log_2 n - 1$ down to 0 **do**
  for all $k = 0$ to $n - 1$ by $2^d + 1$ in parallel **do**
    $t = x[k + 2^d - 1]$
    $x[k + 2^d - 1] = x[k + 2^d + 1 - 1]$
    $x[k + 2^d + 1 - 1] = t + x[k + 2^d + 1 - 1]$

It has a step complexity of O(2 log n) and a work complexity of O(n). Assuming that we want to do a exclusive sum scan on the example above, we get:



Hillis & Steele versus Blelloch

If we have a lot of work and few processors, we would like to use Blelloch. If we have "normal" amount of work and many processors, we would like to use Hillis & Steele.

Lesson 4: Fundamental GPU Algorithms (Application of Sort and Scan)

Compact

Compact (also known as filtering) is a process where we filter a data structure based on some condition. For example, if we have an array of numbers from 1 to 1024, we would like to filter out all odd numbers and square the remaining ones (i.e. square all even numbers).

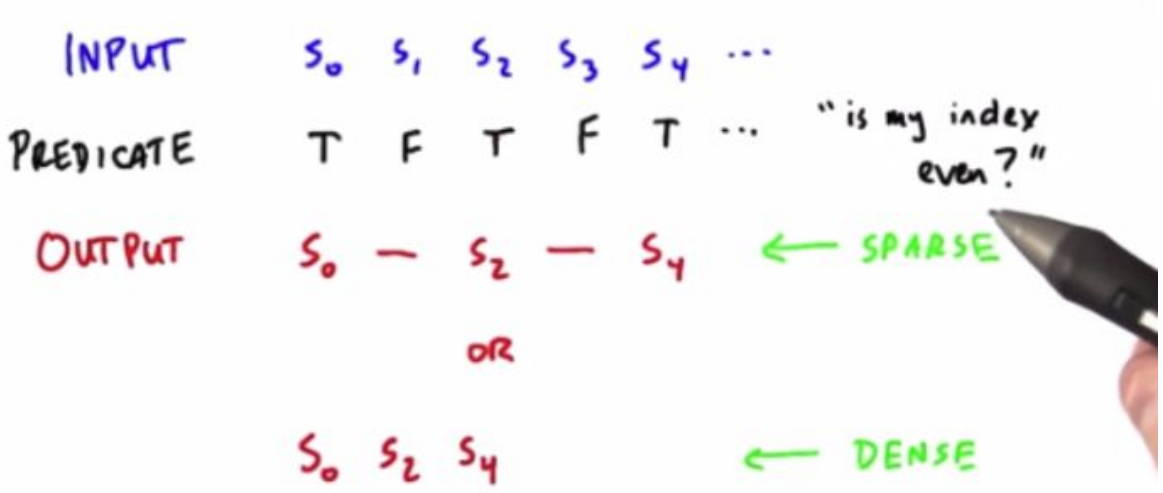
One approach is to launch one thread per item, check if it is even and if so square it, i.e:

```
if(isEven(in_array[i])){
   out_array[i] = square(in_array[i]);
}
```

This would require us to launch 1024 threads, each doing the filtering (i.e. isEven) and then do the squaring. Another way would be to first filter the array, then do the square computation, i.e:

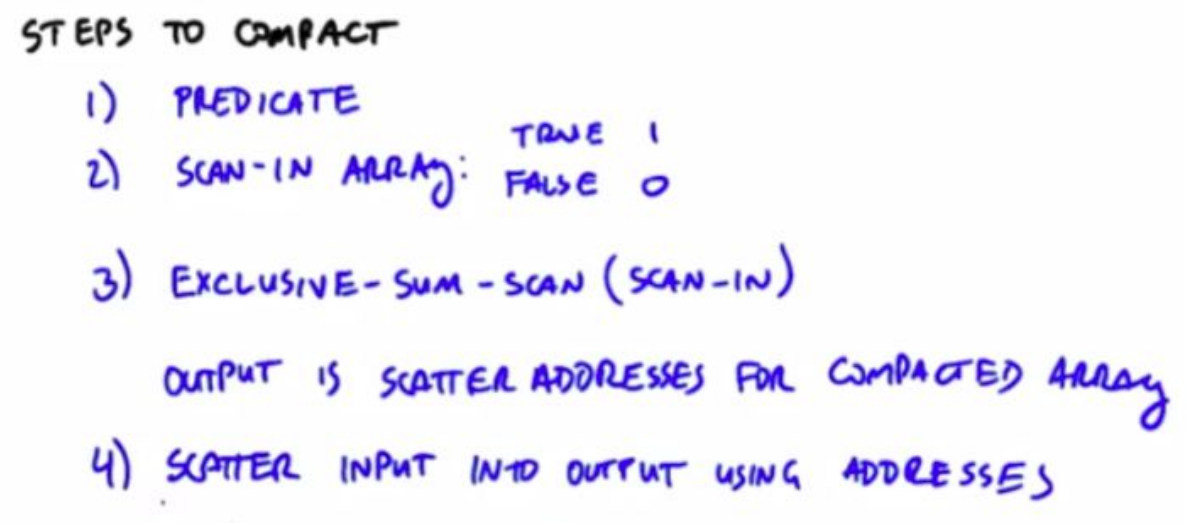
```
cc = filter(array, isEven);
map(cc, square);
```

This would require us to launch 1024 threads doing the filtering and 512 doing the square computation. One problem with the first approach is that half of the threads will be unused/waiting. This is a problem if the computation on the "surviving" elements are expensive (in our case squaring).
Therefore we want to do compact when the input array is large and the computation on the "surviving" elements are expensive. Compact follows:



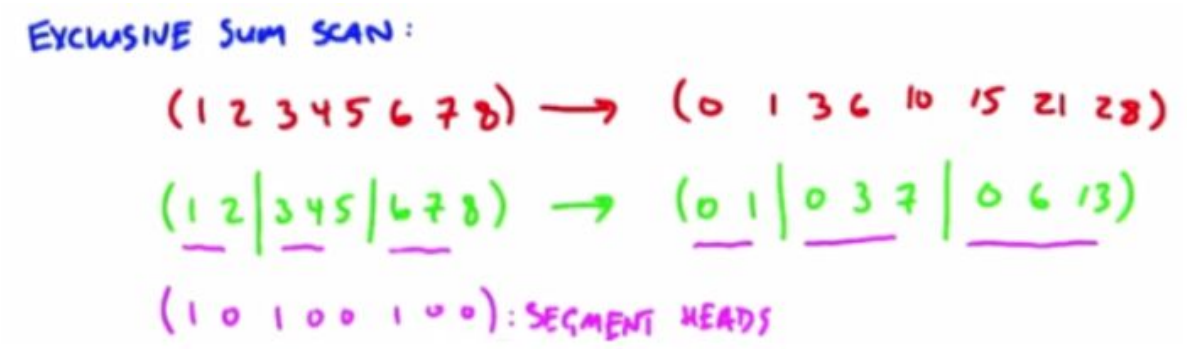
Where predicate describes whether an element fulfills the condition (True/False). The output can either be sparse or dense. Sparse means that the output array has the same size as the input array. Dense means that we only keep the elements where the predicate is true.

The step by step process for compact follows:



Segmented scan

In segmented scan we launch many small (independent) scans. They are combined in segments. Below is an example:



Note that we need a segment head in order to keep track of where the segments starts/ends.

Sorting

In parallel sorting algorithms we want:

1. Keep the hardware busy (lots of threads).
2. Limit branch divergence.
3. Prefer coalesced memory access.

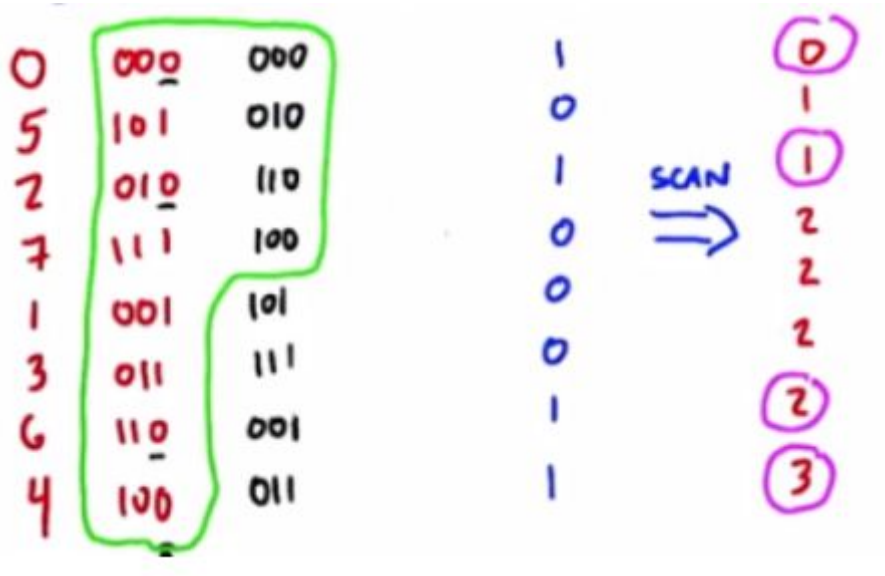
One category of sorting algorithms are comparison based sort (merge sort, quick sort etc). The most efficient ones have a work complexity of O(n log n) and a step complexity of O(log n). However there exist a more efficient sorting method, Radix sort.

Radix sort

Radix sort follows the following steps:

1. Start with the least significant bit (LSB).
2. Split the input into 2 sets based on the bit. Preserve the original order.
3. Move to the next significant bit and repeat step 2.

It has a work complexity of O(kn), where k is the number of bits in representation and n is the number of items. The step complexity is O(log n). Below is an example of the first step:



We use compact to do the splitting.

Key value sorts

In a lot of applications we want to sort keys, each having a value. For example, sort the cities in Sweden based on population count. The key is the population count and the value is the city name. If the value is large (in terms of memory), it is best to use a pointer to the value instead of the actual value when doing the sorting.

Lesson 5: Optimizing GPU programs

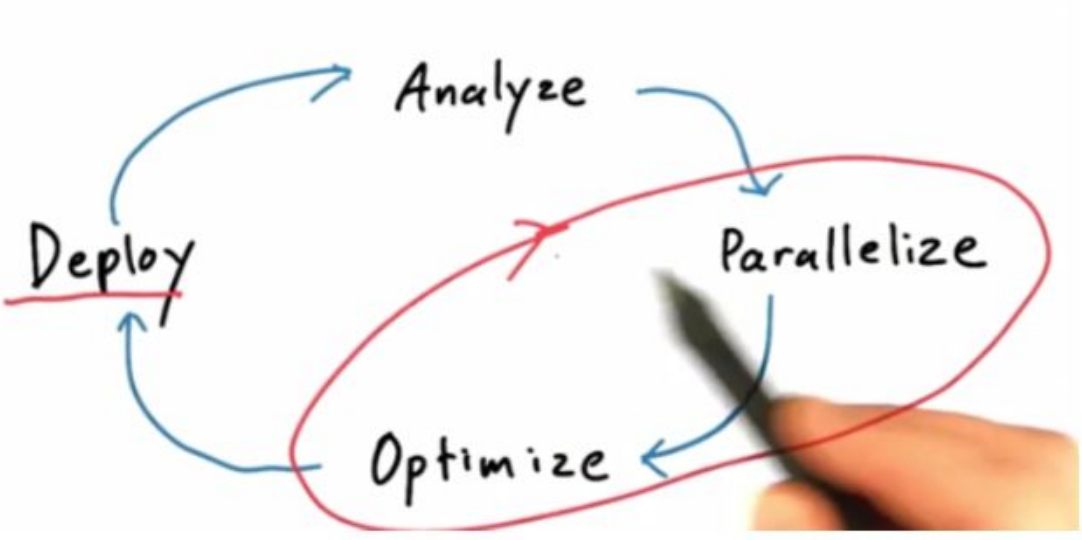
The goal with the optimizations is to solve problems faster, i.e. solve bigger problems / more problems. There are different levels of optimizations:

1. Picking good algorithms (i.e. fundamentally parallel algorithms).
2. Basic principles for efficiency (i.e. coalescing global memory, using shared memory etc).
3. Architecture specific detailed optimization (i.e. optimizing registers etc).
4. Micro-optimization at instruction level (i.e. “float point hacks” etc).

The most gain in terms of performance is achieved from step 1 & 2. Level 3 & 4 are usually not worth the effort for a GPU.

APOD - Systematic optimization

APOD (Analyze, Parallelize, Optimize, Deploy) is an approach to systematically optimize parallel code. It follows:



Analyze

- Profile the whole application.
    - Where can parallelism speed up the code?
    - By how much?

Parallelize

- Pick an approach
    - Libraries (cuBlas etc).

    - Directives (OpenMP etc).
    - Writing custom kernels (what we have done in this course).
- Pick an algorithm (this is a very important step!).

Optimize

- Profile-driven optimization (measure time of execution, use profilers, for example gProfile etc).

Deploy

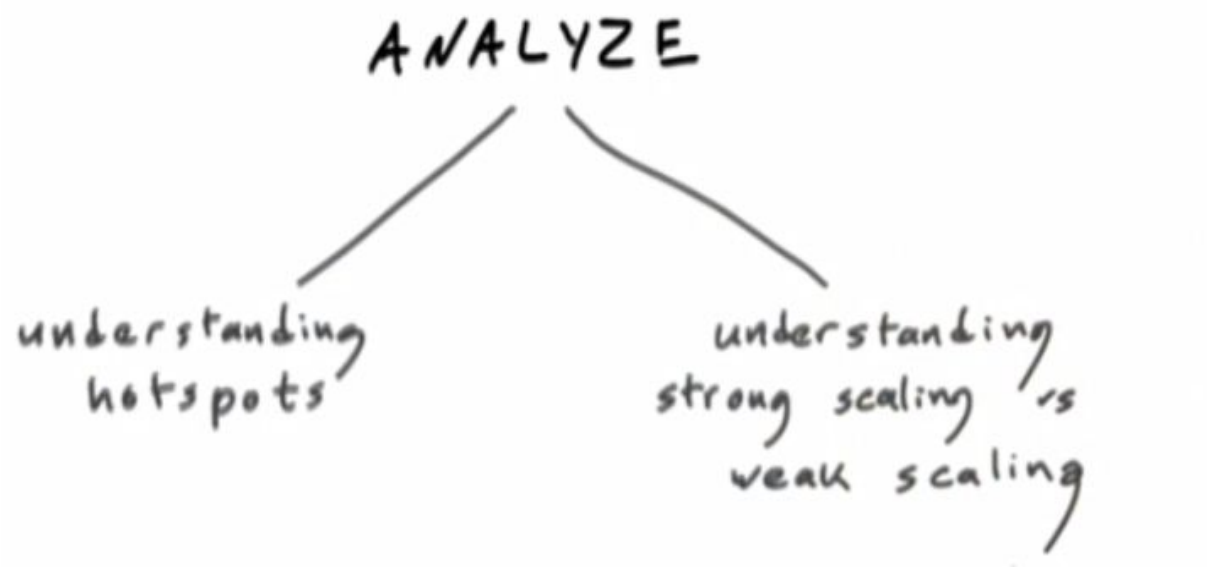
- Measure how the optimization works in real applications, don't optimize in a vacuum.

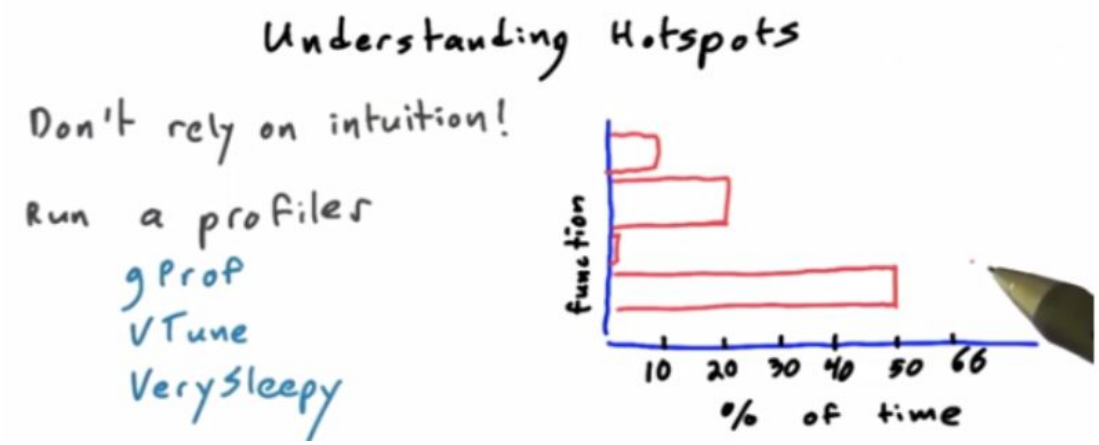
Weak versus strong scaling

Assume that a serial program solves a problem of size P in time T, for example fold a protein in an hour.

Weak scaling: Run a larger problem (or more of them), i.e. solution size varies with problem size per core. For example, fold a bigger protein, or more small ones.

Strong scaling: Run a problem faster, i.e. solution size varies with fixed total problem size. For example, fold the same protein in a minute.





Amdahl's Law

Total speedup from parallelization is limited by portion of time doing something to be parallelized. The max speed up is given by:

$$\frac{1}{1-p}$$

where p is parallelizable time. In the functions above, reducing the function that takes ~50% yields a max speedup of 2 times.