NYU | Center for Data Science

Week 13.2: General purpose GPUs

DS-GA 1004: Big Data

This week

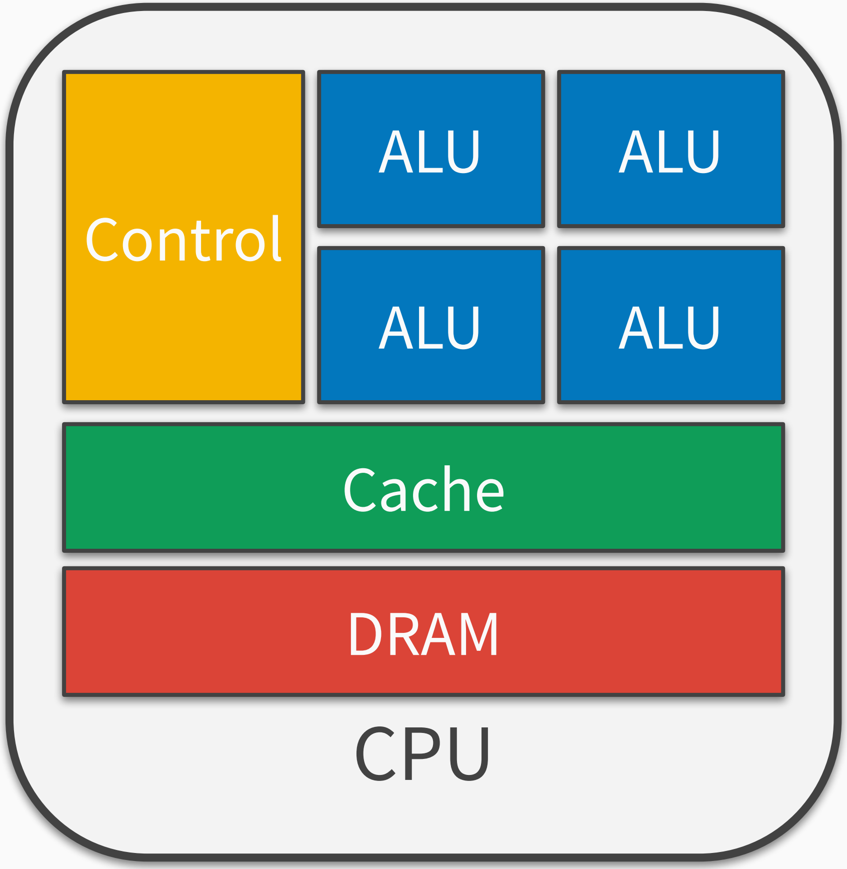
- Graphics processing units (GPUs)
- **GPGPUs and CUDA**
- Software frameworks

Shaders ⇒ Kernels

- General purpose GPUs (GPGPUs) remove the distinction between vertex- and pixel-cores
- "**Shader**" is replaced by "**kernel**" abstraction
    - This allows all cores to operate as either "vertex" or "pixel" cores…
    - Or as something else entirely!
- This is what NVidia's **CUDA** (Compute Unified Device Architecture, 2006) API does

CPU vs GPU architecture

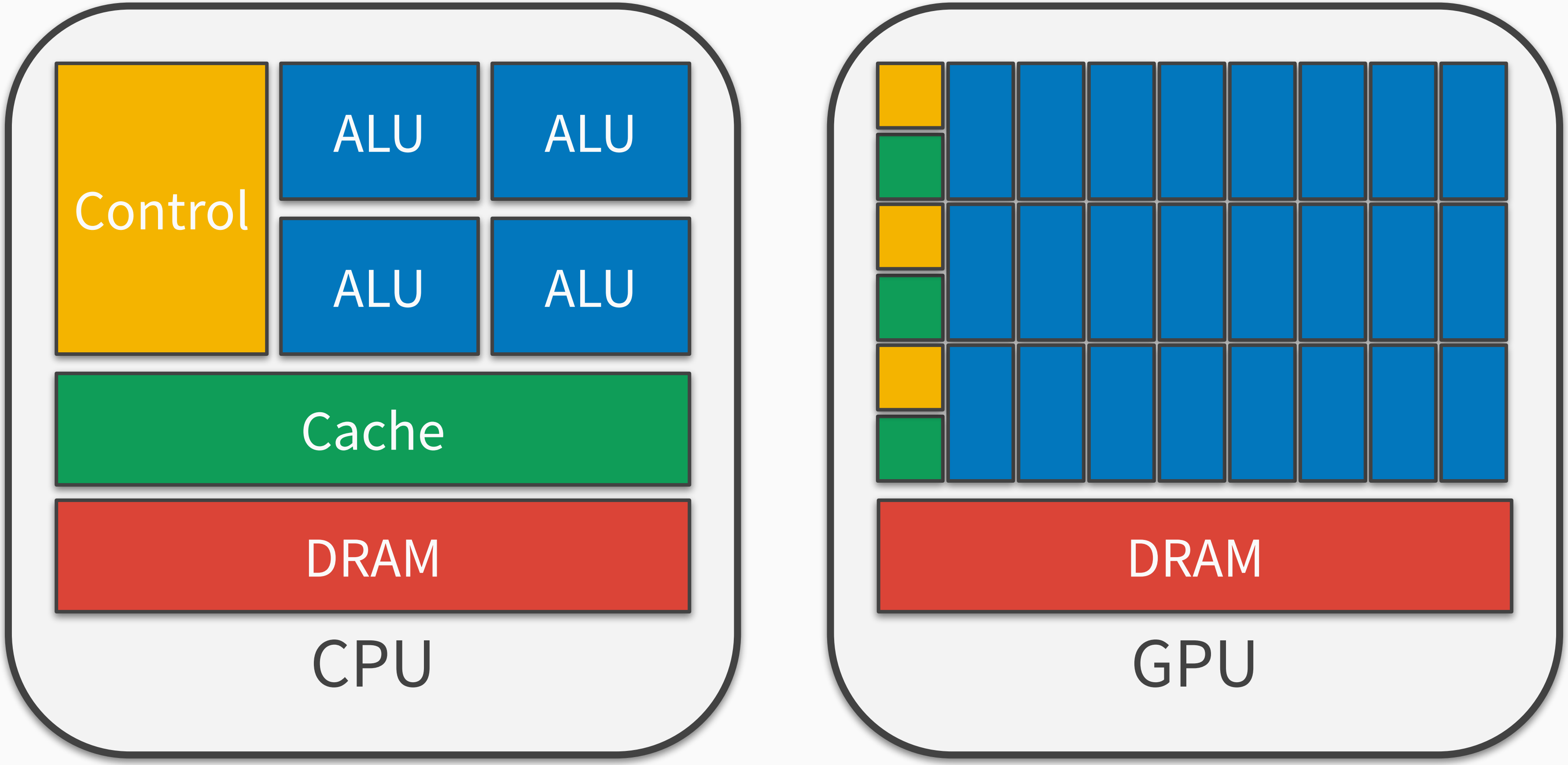
Adapted from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html



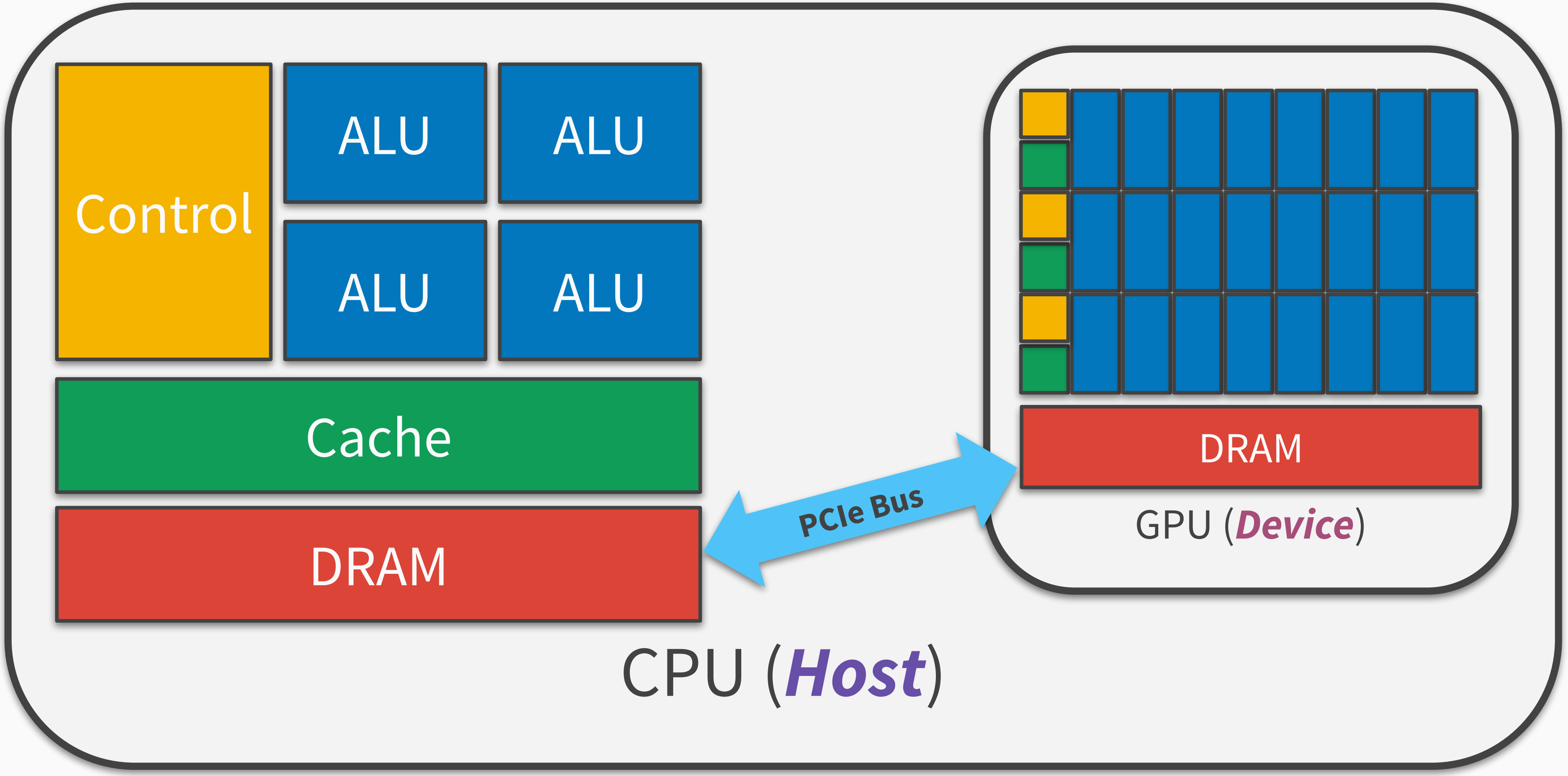
- ALU = Arithmetic logic unit (basic math operations)
- Control = program flow (branching, jumping, etc)
- Cache = on-CPU memory cache
- DRAM = Main system memory

CPU vs GPU architecture

Adapted from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html



A slightly more accurate picture

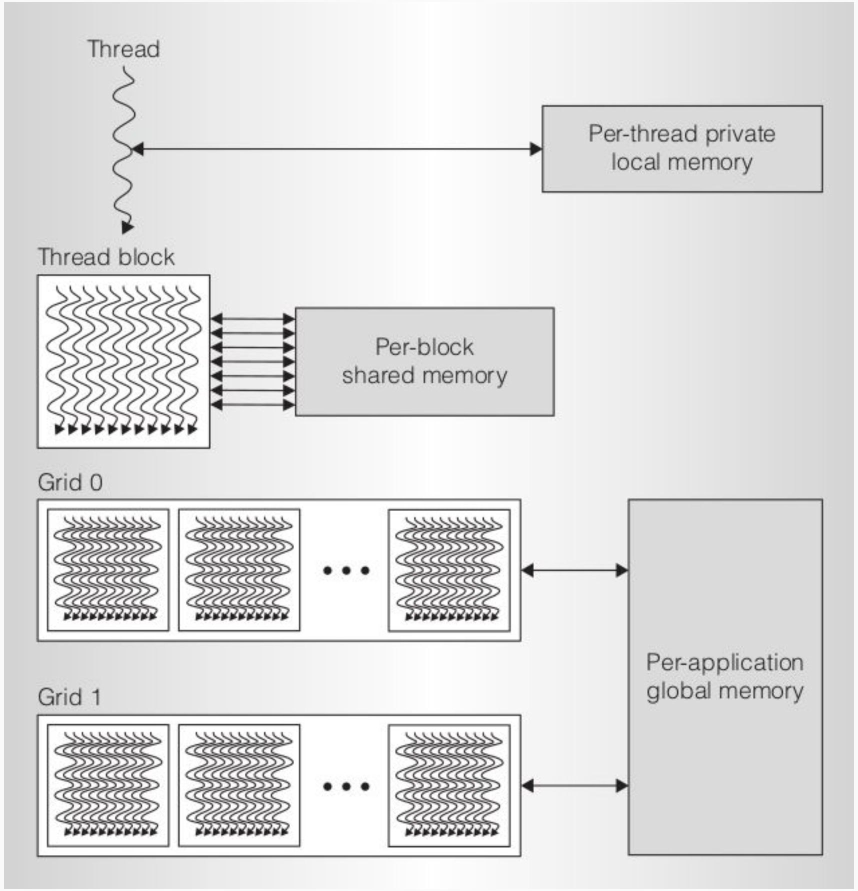


Threads, blocks, and grids

- CUDA arranges kernel execution into

  THREADS, **BLOCKS**, and **GRIDS**

- **Thread**:
  - One execution
  - All threads execute the same kernel (but on different data)
  - Has local, private memory

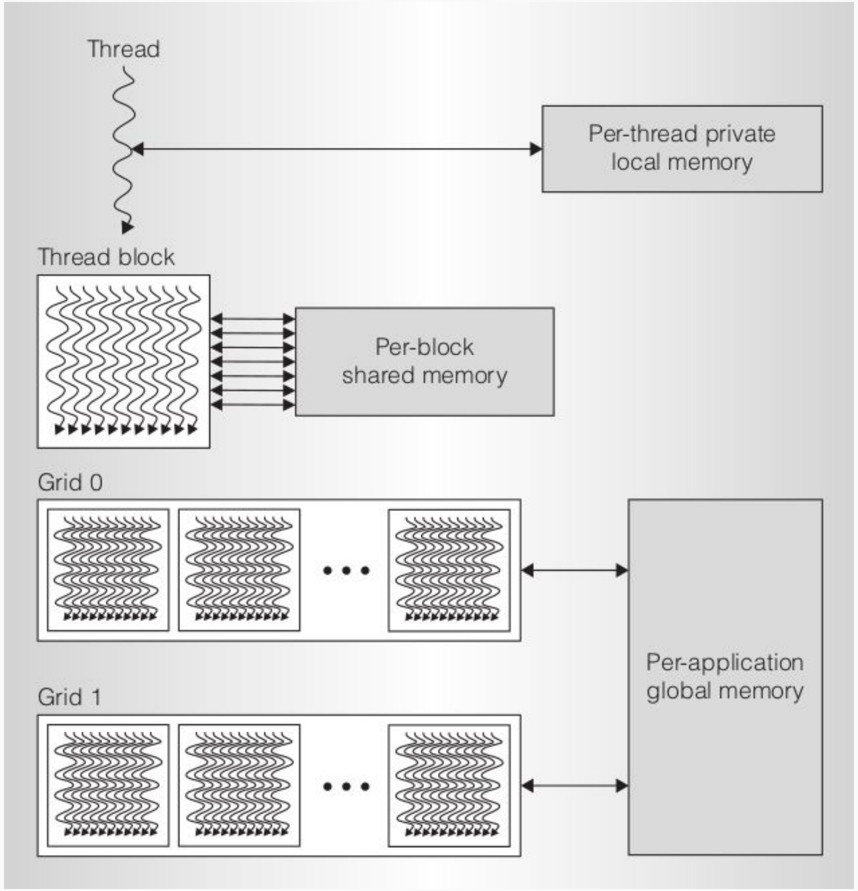

Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

Threads, blocks, and grids

- CUDA arranges kernel execution into

  THREADS, **BLOCKS**, and **GRIDS**

- **Thread**:
  - One execution
  - All threads execute the same kernel (but on different data)
  - Has local, private memory
- **Block**:
  - A group of (possibly related) **threads**
  - Has shared memory for all **threads**

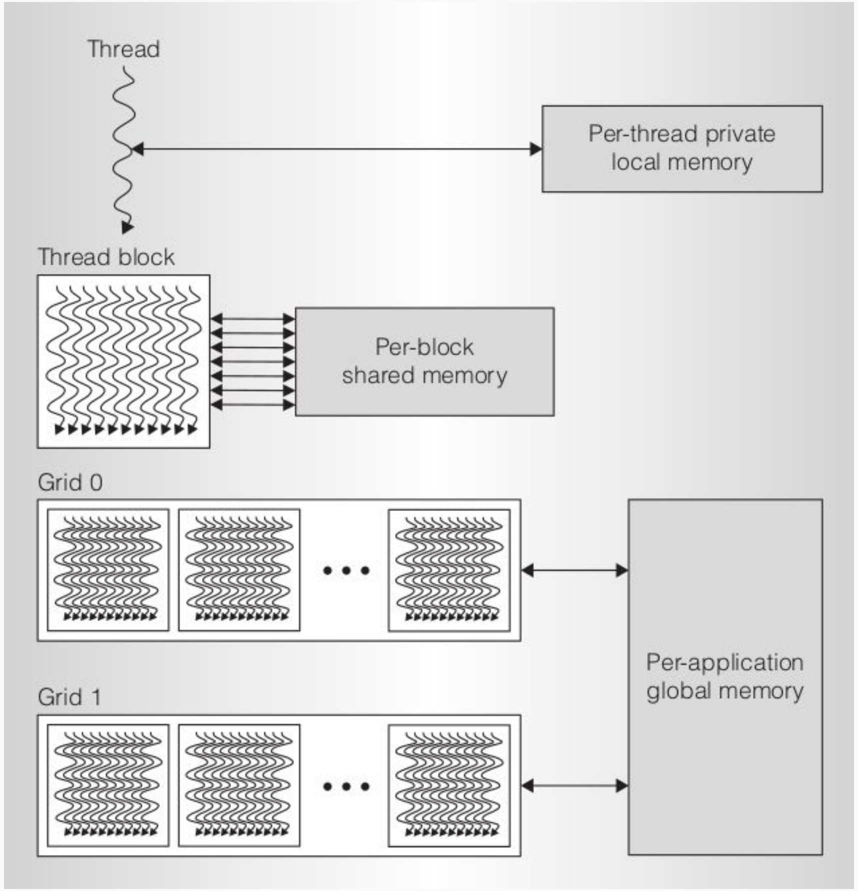

Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

Threads, blocks, and grids

- CUDA arranges kernel execution into

  THREADS, **BLOCKS**, and **GRIDS**

- **Thread**:
  - One execution
  - All threads execute the same kernel (but on different data)
  - Has local, private memory
- **Block**:
  - A group of (possibly related) **threads**
  - Has shared memory for all **threads**
- **Grid**:
  - A collection of **blocks**
  - All grid cells have access to shared global memory (application-level)


Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

Example “saxpy”: $y \leftarrow a*x + y$

```
def serial_saxpy(N, a, x, y):
    for i = 0 .. N-1:
        y[i] = a * x[i] + y[i]



a = <some number>
x = <array of N numbers>
y = <array of N zeros>

serial_saxpy(N, a, x, y)
```

- Single-precision ax+y
- Cost for serial implementation: O(N)
- Computational dependencies are trivially parallel

Example “saxpy”: y ← a*x + y

```
def serial_saxpy(N, a, x, y):
    for i = 0 .. N-1:
        y[i] = a * x[i] + y[i]


a = <some number>
x = <array of N numbers>
y = <array of N zeros>

serial_saxpy(N, a, x, y)
```

```
def cuda_saxpy(N, a, x, y):
    i = blockIdx.x * blockDim.x + threadIdx.x
    if i < N:
        y[i] = a * x[i] + y[i]

a = <some number>
x = <array of N numbers>
y = <array of N numbers>
d_x ← copy x to GPU
d_y ← copy y to GPU

cuda_saxpy<<< ⌈N/256⌉, 256 >>>(N, a, d_x, d_y)

y ← copy d_y from GPU
```

Example "saxpy": $y \leftarrow a*x + y$

- Data access is managed by the thread:
    - blockIdx.x = current block index
    - blockDim.x = size of the current block
    - threadIdx.x = current thread within block

- ⌈N/256⌉ = number of **blocks**
  256 = number of **threads per block**

- "**if** i < N …" avoids array bound errors from integer math in block division

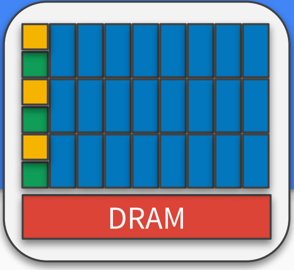
```
def cuda_saxpy(N, a, x, y):
        i = blockIdx.x * blockDim.x + threadIdx.x
        if i < N:
                y[i] = a * x[i] + y[i]

a = <some number>
x = <array of N numbers>
y = <array of N numbers>
d_x ← copy x to GPU
d_y ← copy y to GPU

cuda_saxpy<<< ⌈N/256⌉, 256 >>>(N, a, d_x, d_y)


y ← copy d_y from GPU
```

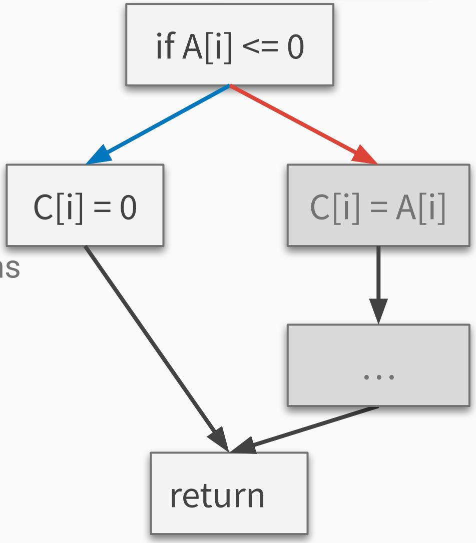
Thread execution and warps



- **Threads within a block** execute in parallel via SIMT
  - Single-Instruction, Multiple-Thread

- Threads run in groups of 32 called a ***warp***
  - Threads start at the same instruction, **but can follow different execution paths**
  - **Warp finishes when all threads have finished**
  - Pre-Volta (2017) GPUs share the instruction counter, disable cores on different paths
  - For maximum efficiency: **threads should follow a common path**!
  - Threads can be explicitly synchronized: __syncthreads();

- **Blocks** need not execute all simultaneously
  - E.g. if you have more blocks than cores
  - This is why you can't share memory between blocks



Example 2: Dot product (single-block)

```
// Vector dot product with shared memory
__global__ void dot(float *a, float *b, float *c, int N)
{
        // Each thread acts as a mapper
        __shared__ float temp[N];
        int i = threadIdx.x;
        temp[i] = a[i] * b[i];

        // Wait here for all threads to finish
        __syncthreads();

        // Use thread 0 as reducer
        if (threadIdx.x == 0) {
                float sum = 0.0;
                for (int i = 0; i < N; i++) {
                        sum += temp[i];
                }
                *c = sum;
        }
}
```

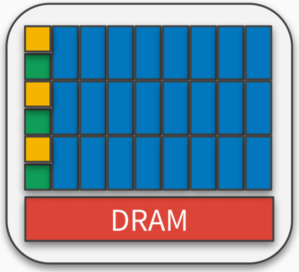
This could be improved in several ways…

- Use blocks to compute partial sums
  - ⇒ like **combiners** in map-reduce!
- Divide-and-conquer for sum aggregation
- Use atomic updates for output buffer
  - ⇒ like **accumulators** in spark!

Threads, blocks, grids, and hardware

- **Like mappers**, kernels must be independent of one another and able to execute **in any order**

- **Unlike mappers,** kernels are not pure functions.
  - In fact, they cannot have return values at all!
  - Outputs are written to pre-allocated memory buffers

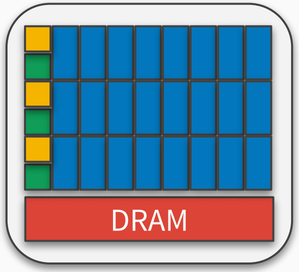


Threads, blocks, grids, and hardware

- **Like mappers**, kernels must be independent of one another and able to execute **in any order**

- **Unlike mappers,** kernels are not pure functions.
  - In fact, they cannot have return values at all!
  - Outputs are written to pre-allocated memory buffers

- To exploit shared memory within blocks, you must be careful about organizing your data (**just like RDD partitions!**)



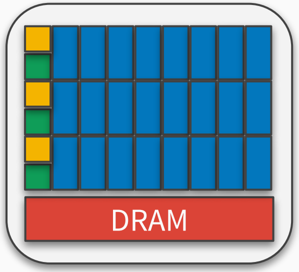
Threads, blocks, grids, and hardware

- **Like mappers**, kernels must be independent of one another and able to execute **in any order**

- **Unlike mappers,** kernels are not pure functions.
  - In fact, they cannot have return values at all!
  - Outputs are written to pre-allocated memory buffers

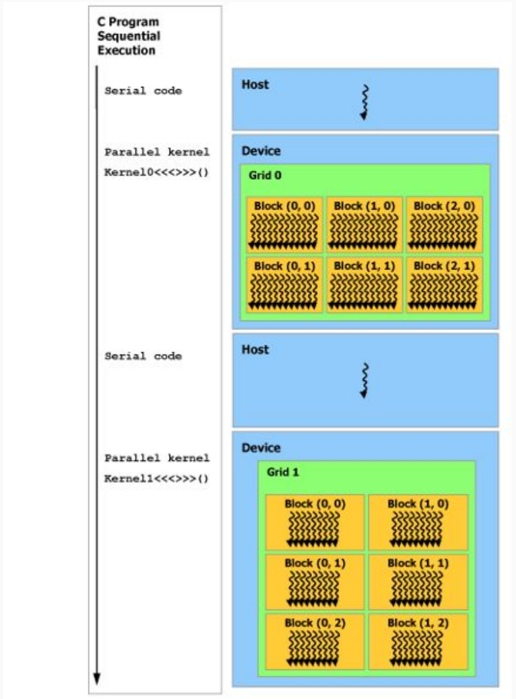
- To exploit shared memory within blocks, you must be careful about organizing your data (**just like RDD partitions!**)

- Different hardware will have different # cores
  - Thread/block abstraction hides the hardware core layout from the software
  - Current GPUs allow up to 1024 threads per block



Complex programs

- CUDA kernels are usually "simple" operations
  - Vector addition, matrix multiplication, etc.
- Usually we want to combine these to make complex programs
  - E.g. back-propagating through a convolutional network
- Not dissimilar from spark interleaving **transformations** with **actions**



Pitfalls of GPU usage

- Efficient usage relies on keeping the GPUs busy at all times
- Usually an idle GPU is waiting for data from the CPU
    - Memory transfer (**communication**) is often the biggest bottleneck!
    - Try to keep as much data on the GPU as possible without transfers
- Programs with complex control flow (eg conditionals)
- Writing custom GPU code is a daunting task… use existing frameworks!

Summary

Part 2: GPGPUs

- GPGPUs were originally designed to overcome imbalance in graphics, but proved to be a very useful abstraction!
- Shaders ⇒ Kernels
- Distributed programs with simple control flow on shared memory