Programming for GPUs

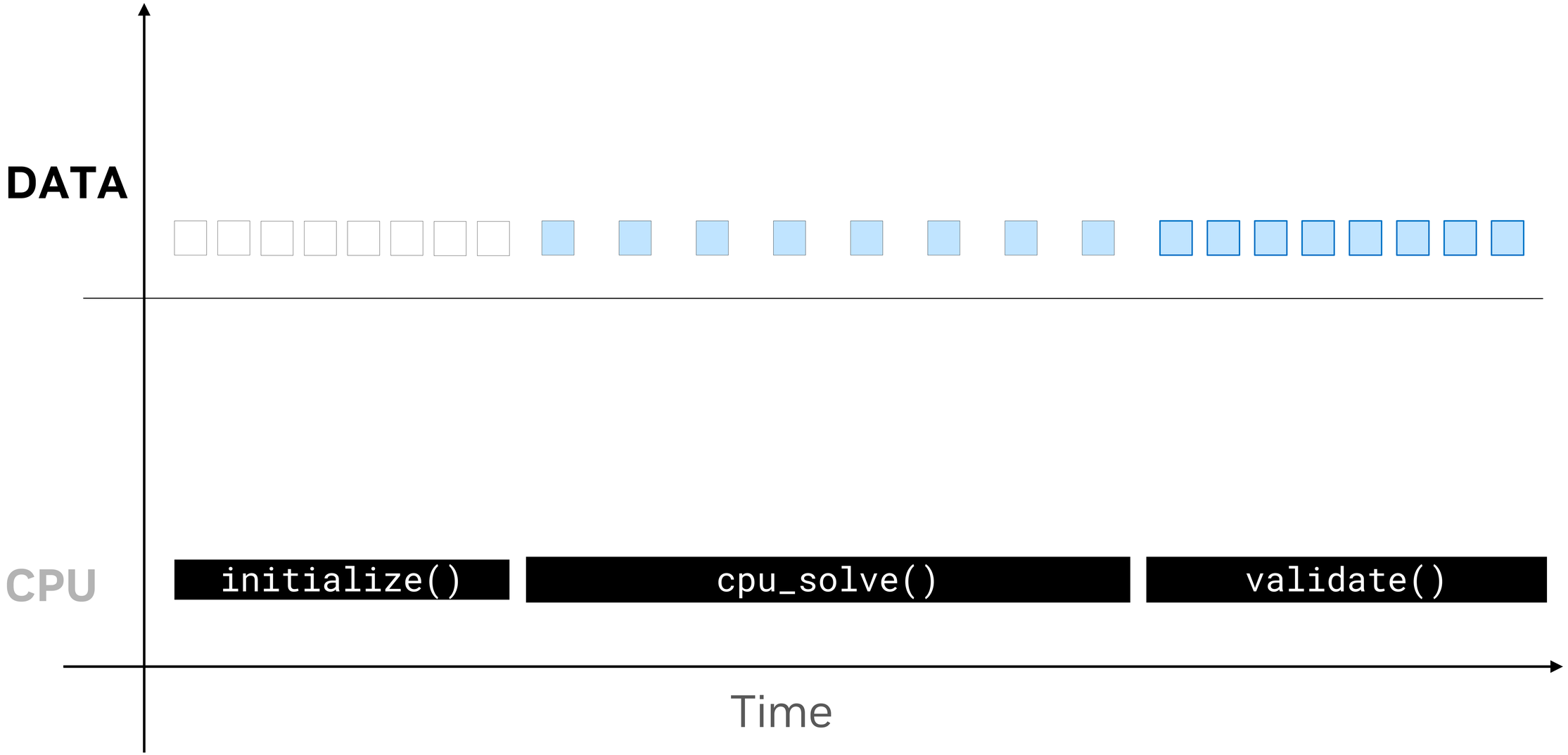
Daniel Cámpora | Senior AI Devtech Engineer, NVIDIA

Table of Contents

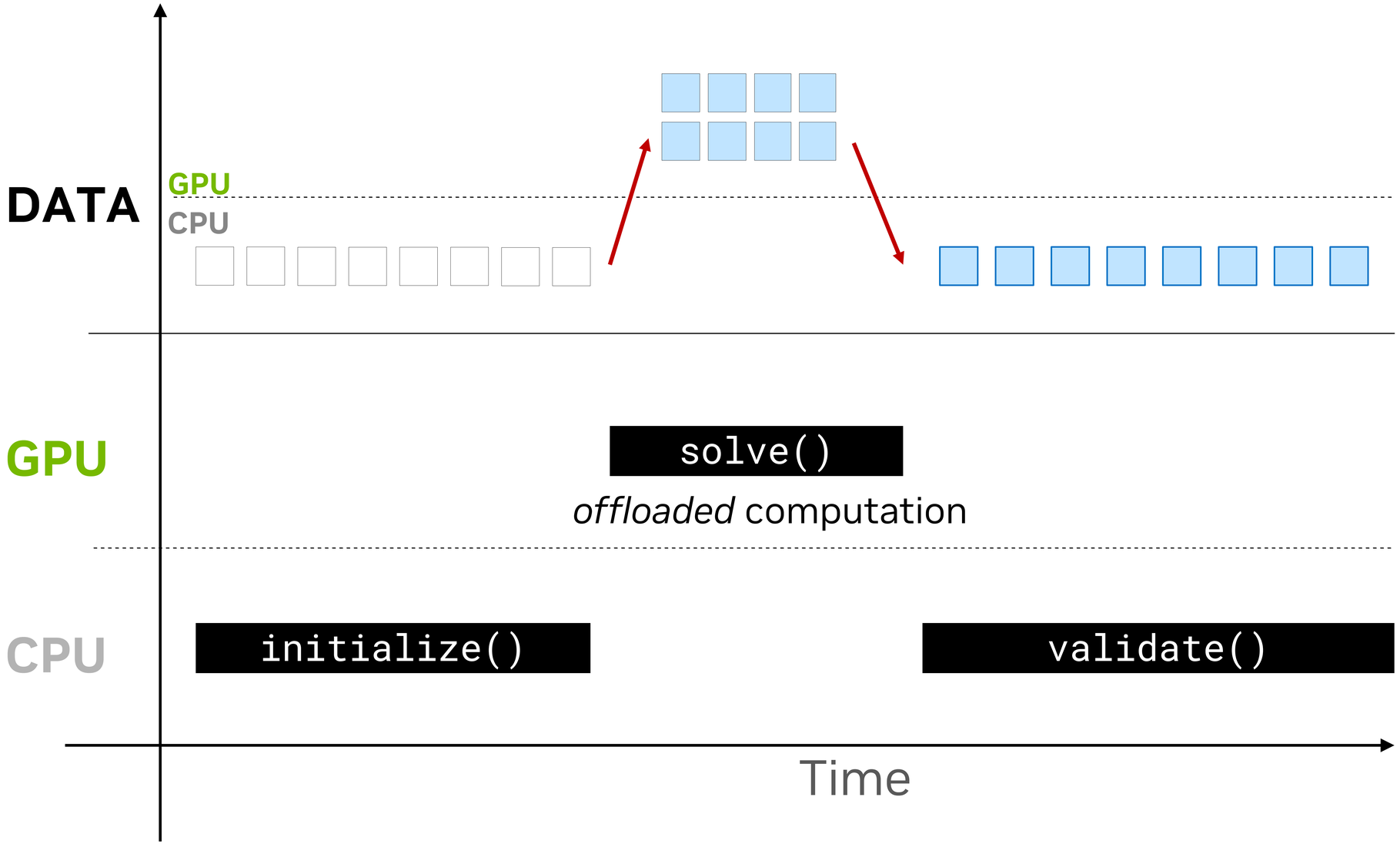
What Is CUDA?

- CUDA stands for *Compute Unified Device Architecture.*
- It is a programming model introduced in 2006 by NVIDIA as a set of extensions to the C programming language.
- Nowadays, it works with a variety of languages: Python, C, C++, Fortran, etc.
- It allows GPUs to be used for general purpose computing (also referred to as **GPGPU** or **GPU computing**).
- Since its inception, other standards have emerged, such as OpenCL, ROCm or SYCL, to name a few. We will discuss them in depth in lecture *Design patterns and best practices.*

A CPU Application

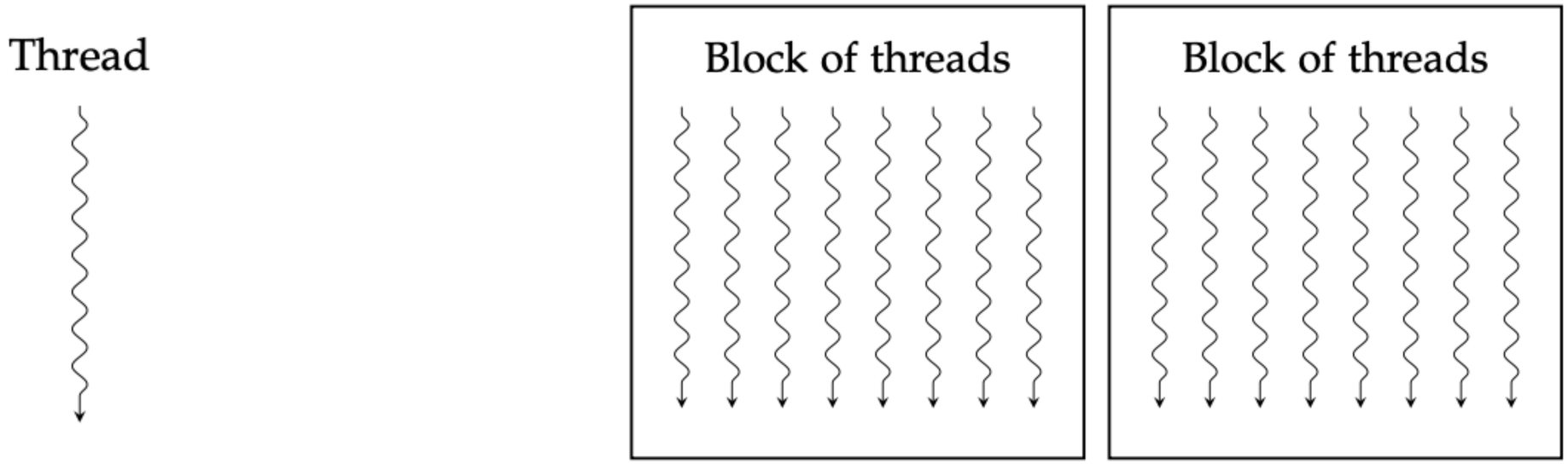


A GPU Application



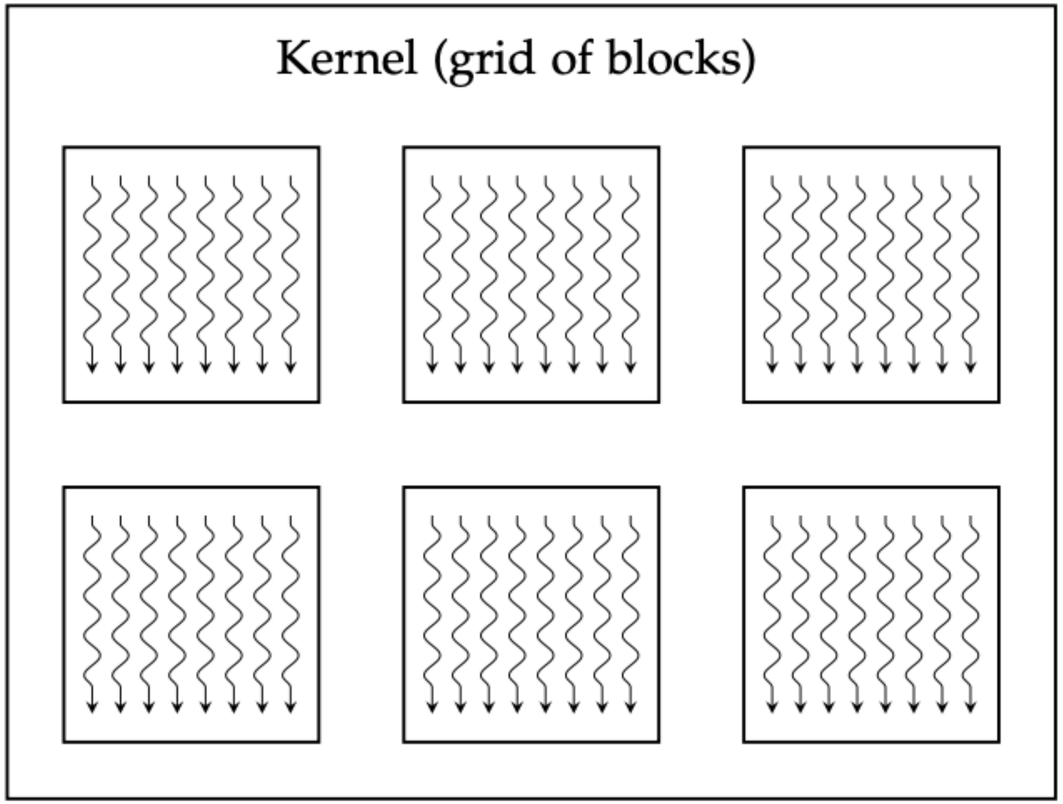
Parallel Processors

- GPUs are parallel processors that can execute many threads in parallel.
- Threads are organized in **blocks of threads**.



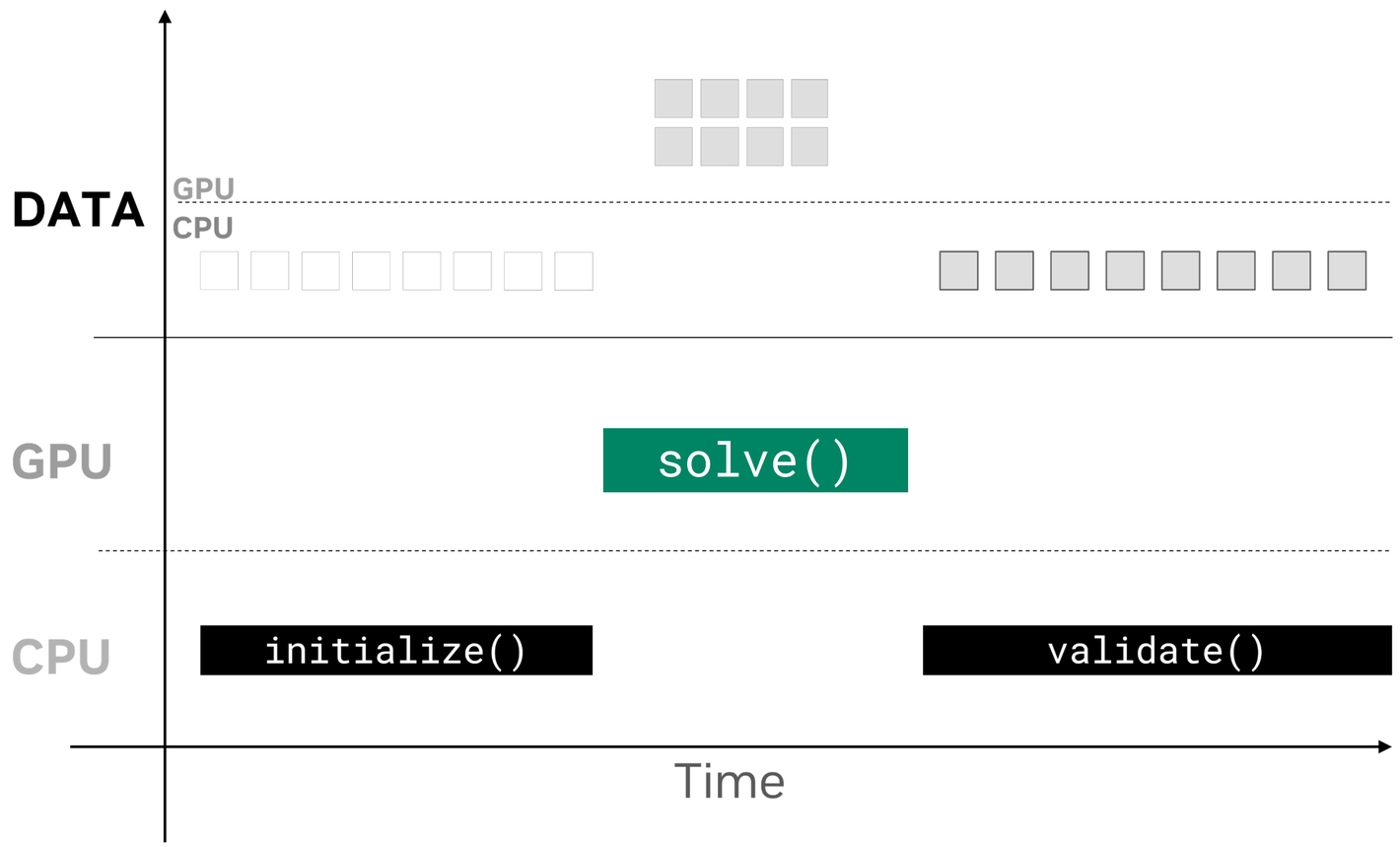
The Kernel

- A function in CUDA, also called a **kernel**, is invoked with a configurable **grid of blocks**, each with the same number of threads.



Kernel Execution Example

- The **`solve`** kernel runs blocks and threads in parallel.
- According to *Amdahl's law*, it's the best target to parallelize in this example.
- Let's inspect it.



A Day in a Kernel's Life

- `solve` is invoked with a configuration of 2 blocks and 4 threads per block.
- Each block runs independently from one another.
- Each thread runs independently from one another.
- All blocks in a grid must have the same number of threads.

Indices

- Inside our kernel execution, **indices** identify each individual thread.
- `gridDim.x` is the number of blocks in the grid, in this case 2.
- `blockIdx.x` identifies the current block within the grid.
- `blockDim.x` refers to the number of threads in a block, in this case 4.
- `threadIdx.x` identifies the current thread within the block.
- The formula **`blockIdx.x * blockDim.x + threadIdx.x`** uniquely identifies threads in our kernel in the grid.

Coordinating Parallel Threads

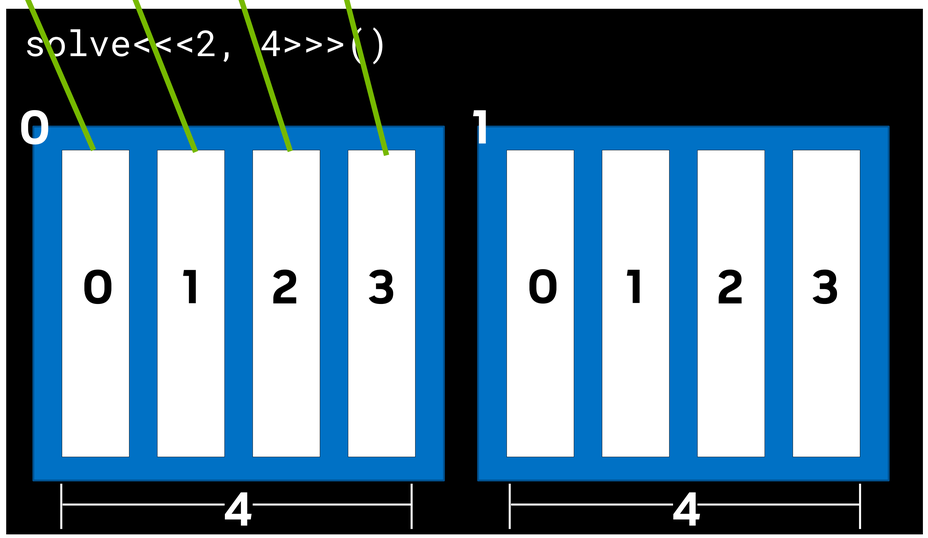
GPU
DATA

0	1	2	3
4	5	6	7

blockIdx.x	*	blockDim.x	+	threadIdx.x
0		4		3

data index
3

GPU



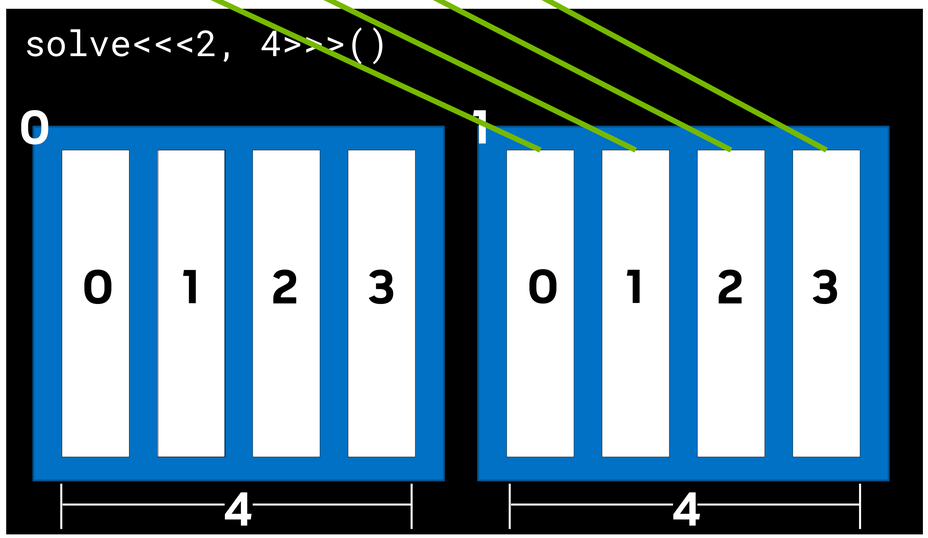
Coordinating Parallel Threads

GPU
DATA

0	1	2	3
4	5	6	7

blockIdx.x	*	blockDim.x	+	threadIdx.x
1		4		3

data index
7



GPU

Configuration Goodies

- Grid and block size can be configured with 3 dimensions.
- The 3 dimensions allow for simplicity when assigning tasks to each block / thread.
  - Eg. a configuration of `{32,4,4}` threads will generate 512 threads in total.
- **There is a maximum of 1024 threads per block**
  - The multiplication of the three dimensions should not exceed the maximum.

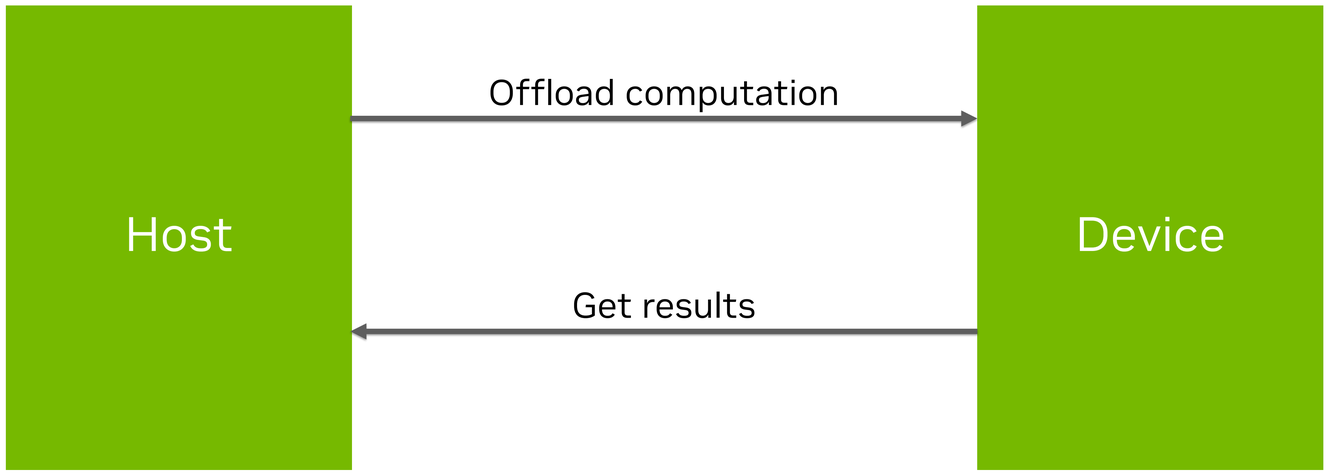
- Dimensions can be accessed with members x, y and z:
  - `gridDim.x, gridDim.y, gridDim.z`
  - `blockDim.x, blockDim.y, blockDim.z`
  - `blockIdx.x, blockIdx.y, blockIdx.z`
  - `threadIdx.x, threadIdx.y, threadIdx.z`

Table of Contents

- The CUDA Programming Model
- **Host, device and memory**
- Writing a kernel
- GPU architecture
- Common data parallel techniques
- Summary

Host and Device

- A GPU-accelerated application runs like any other OS application.
- It requires a CPU, which launches the application and acts as the **host**.
- The host can offload some of the work onto the GPU, which acts as the **device**.
- The host is in charge of the application at all times.
  - It can kill the application, react to interrupts, etc.



CUDA Syntax

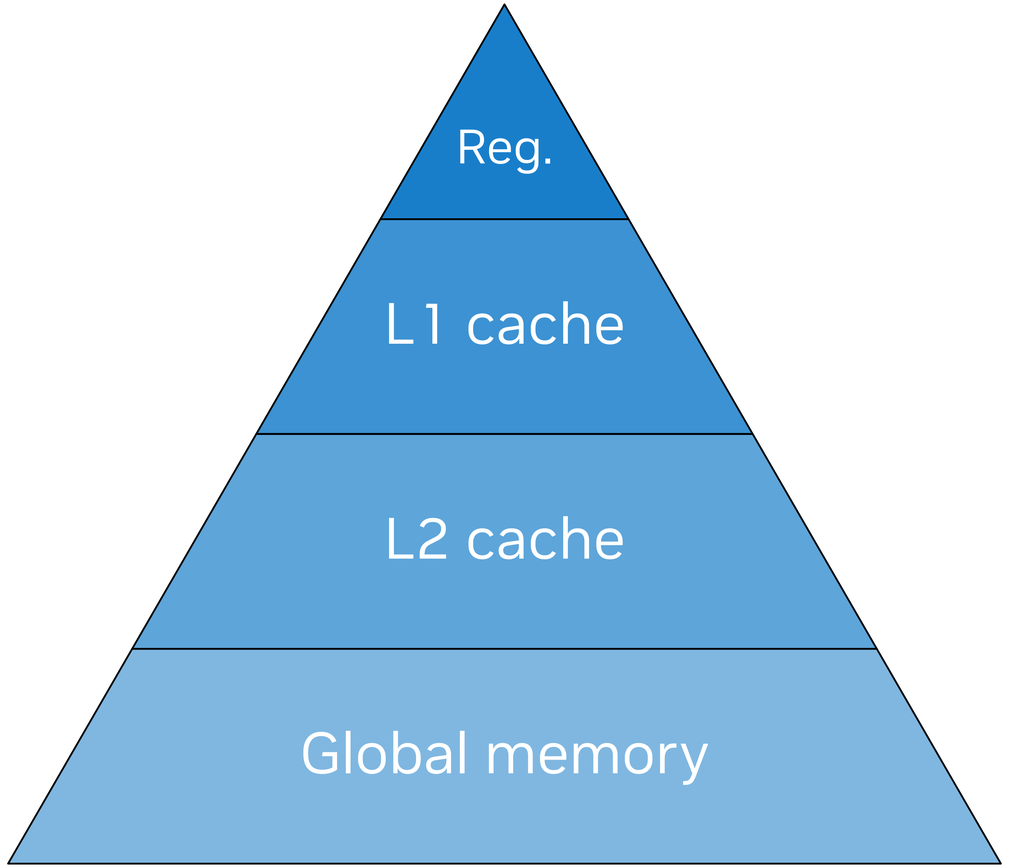
Function qualifiers:

- `__host__` A function that is invoked by the **host**, and runs on the **host**.
- `__device__` A function that is invoked by the **device**, and runs on the **device**.
- `__global__` A function that is invoked by the **host or device**, and runs on the **device**.
  - `__global__` functions always execute asynchronously.

Memory Hierarchy

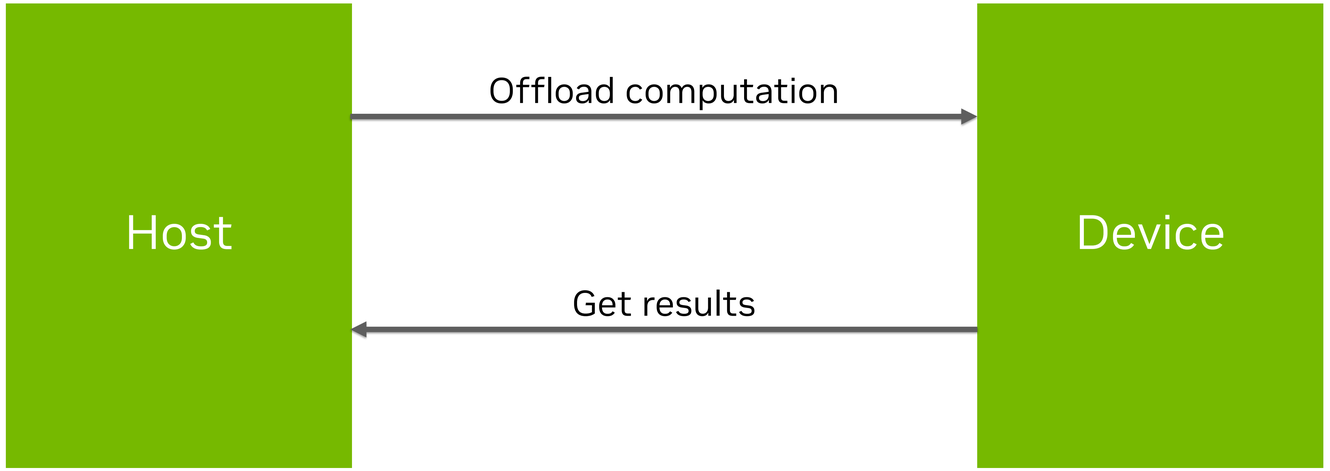
The GPU has three main kinds of memory:

- **Global memory** – High latency, GBs of space.
- **Caches** – Lower latency.
  - L2 cache – MBs of space.
  - L1 cache – KBs of space.
- **Registers** – Lowest latency.
  - A configurable 64-255 registers available per GPU core.



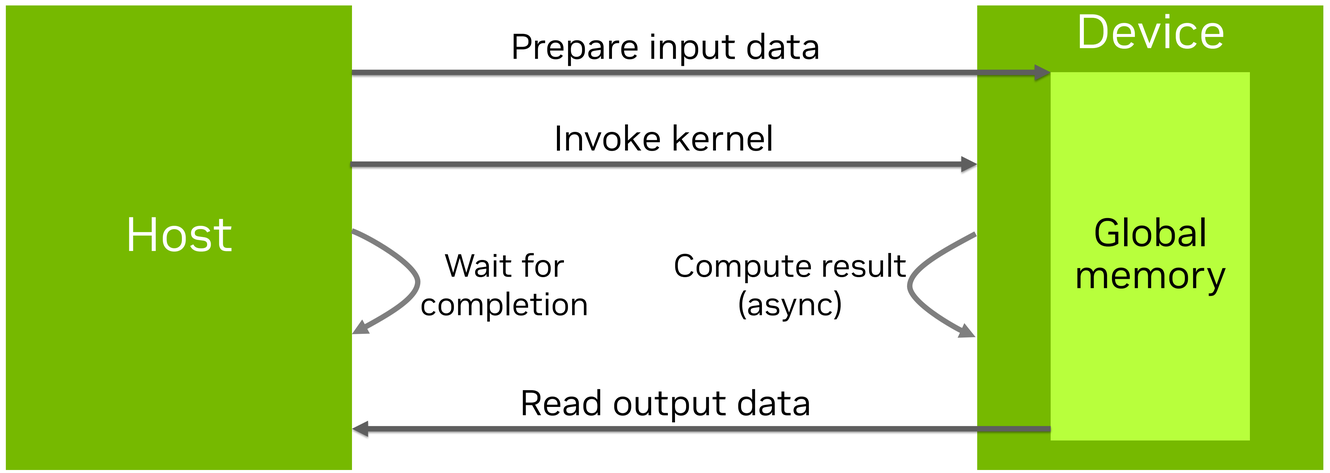
Host – Device Communication

- The host can only access device global memory.
- Therefore, all **input data** to the GPU needs to be populated on **global memory** prior to starting our kernel.
- All **output data** needs to be put on **global memory** before the kernel ends executing.



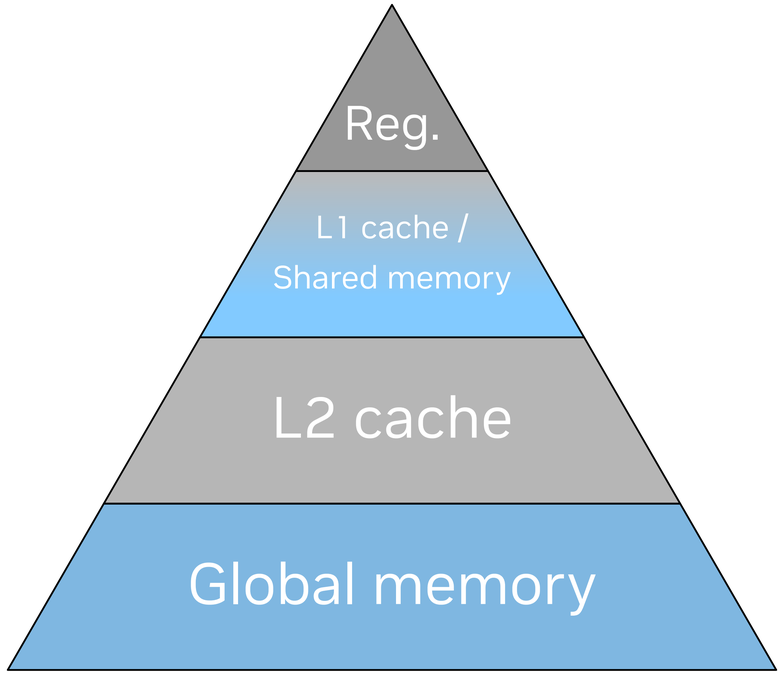
Host – Device Communication (2)

- The host can only access device global memory.
- Therefore, all **input data** to the GPU needs to be populated on **global memory** prior to starting our kernel.
- All **output data** needs to be put on **global memory** before the kernel ends executing.



Shared memory

- GPUs offer a low-level optimization that is not available on CPUs.
- A part of **L1 cache** can be configured as **shared memory**.
  - Memory used for shared memory will not be used for L1 (tradeoff).



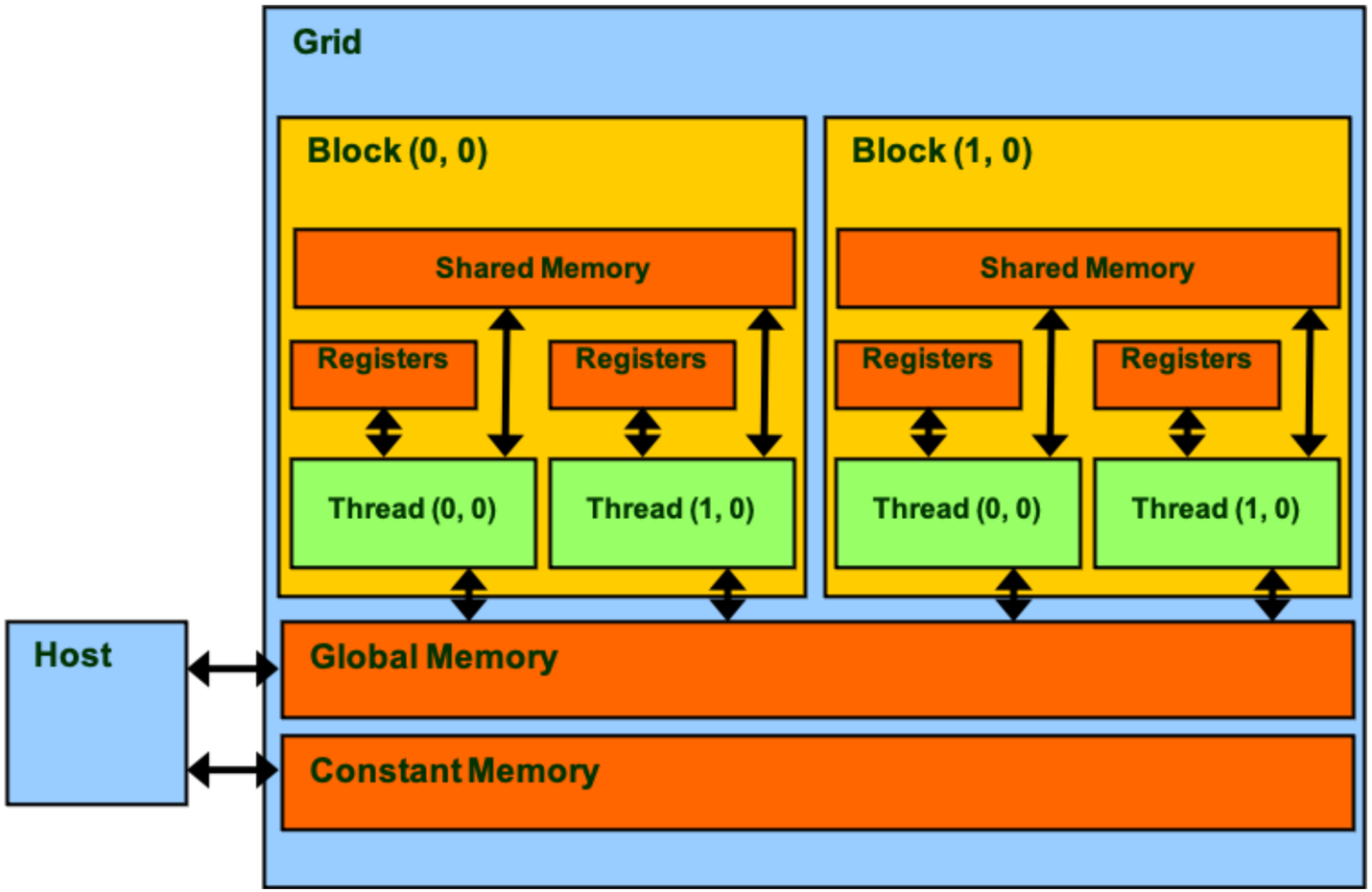
Shared Memory (2)

- Shared memory is:
  - Fast memory you have direct control over.
  - Limited in size.
  - Shared among the block.
  - It can only be accessed from the device.
  - Its contents are flushed after the `__global__` function terminates.

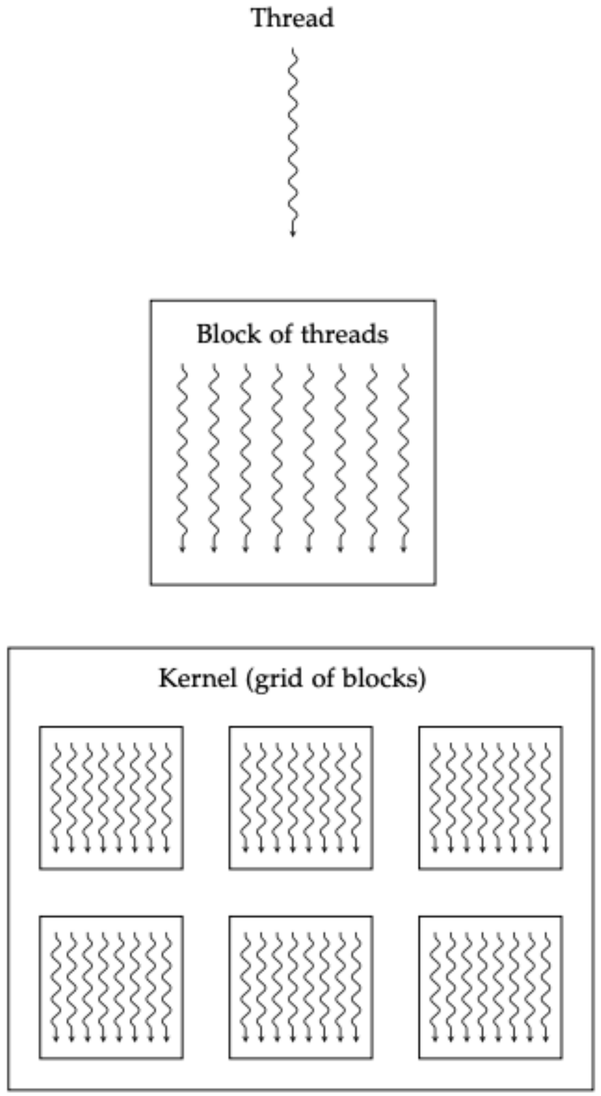
- Shared memory size is limited:
  - On the Tesla T4 that you will use in the tutorials, its processors (*SMs*) allow a maximum of **48 KBs**.
  - Generally speaking, it is configurable and its maximum size depends on the architecture.

- We'll see an example in lecture *Performant Programming for GPUs*.

Memory Overview



Memory Schema



registers

local memory

We will see this in the next lecture.

shared memory

global memory

constant memory

texture memory

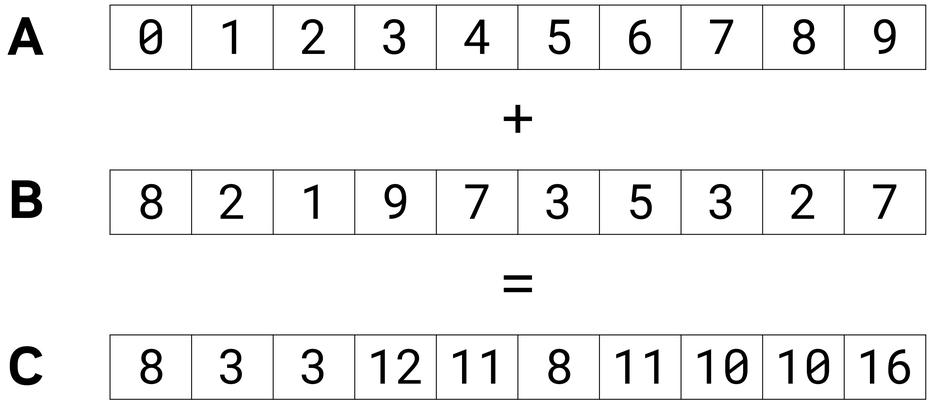
You can ignore constant and texture memory.

Table of Contents

- The CUDA Programming Model
- Host, device and memory
- **Writing a kernel**
- GPU architecture
- Common data parallel techniques
- Summary

Vector Addition

- Let’s write a vector addition kernel.



- **A**, **B**, and **C** are arrays of `float` of size **N**.
- Compute to global memory access ratio (arithmetic intensity in FLOPs / Bytes):

10 FLOPs / 120 Bytes = 0.08

Vector Addition

- Recall the Roofline model.
- The peak performance on the T4 is:

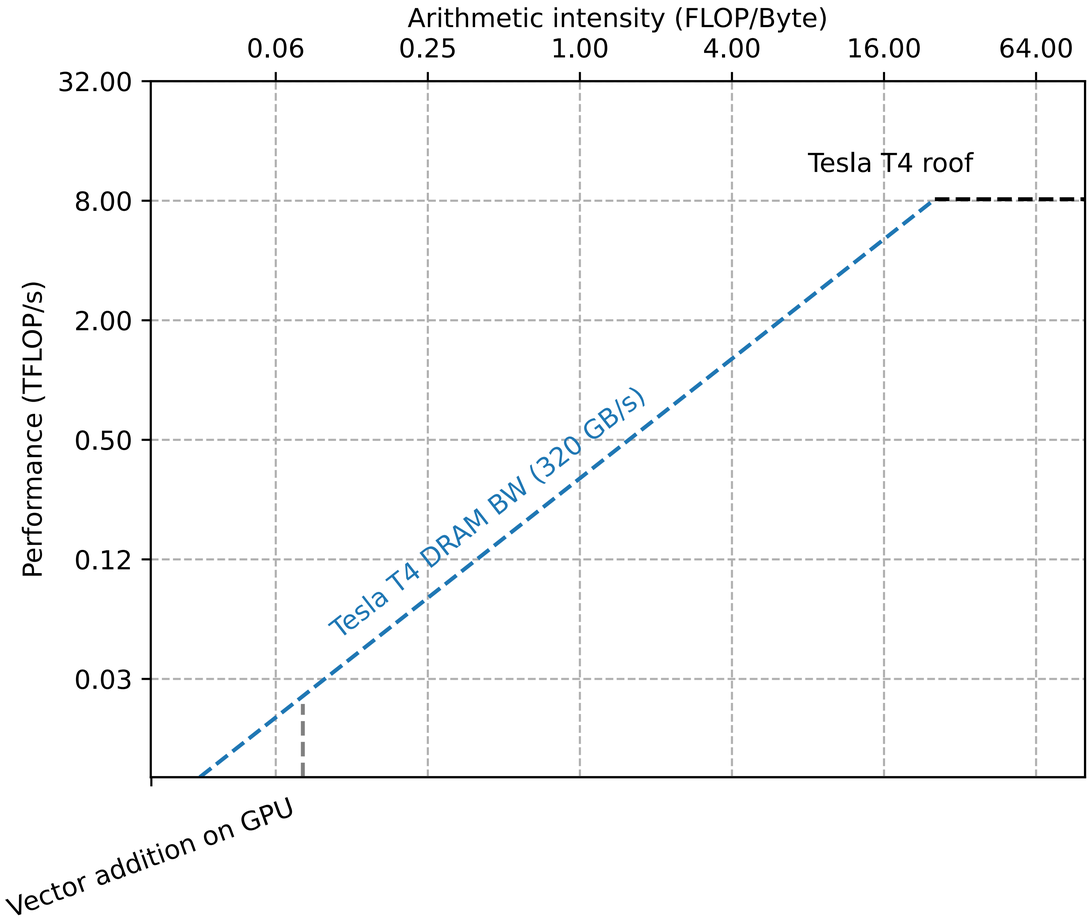
  8 TFLOPS

- Our peak performance on the T4 is:

  0.32 TBps * 10 FLOPS / 120 B = 0.027 TFLOPS

- What is the main limitation of this code?

  We are heavily **memory bound**, and we can obtain at best 0.027 / 8 = 0.0034 = **0.34% of the performance the T4 has to offer with a vector addition**.



CUDA Syntax Reminder

- Function attributes: `__global__`, `__device__`, `__host__`
- Indices: `threadIdx.x`, `blockIdx.x`
- Dimensions: `gridDim.x`, `blockDim.x`

- Kernels (`__global__` functions) invocation must specify grid and block dimensions as follows:
  - `fn<<<grid_dim, block_dim>>>(arg0, arg1, ...);`

Vector Addition Parallelized Across Single Block of Threads

This function is marked `__global__`, it can be invoked on the host and it will be executed on the device.

The value of `threadIdx.x` will be different for each thread in the block.

```
__global__ void vector_addition(float* A, float* B, float* C) {
  unsigned i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main() {
  ...
  vector_addition<<<1, N>>>(A, B, C);
  ...
}
```

`vector_addition` is invoked from the host. It will run asynchronously and non-blockingly. Control is returned immediately to the host.

The kernel will be invoked with a grid dimension of 1, and a block dimension of N. In other words, a single block of N threads.

Flexibility

- The prior kernel assumes the block dimension to be N.

```
__global__ void vector_addition(float* A, float* B, float* C) {
  unsigned i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

- Invoking the kernel with any other block dimension leads to incorrect results or out of bounds accesses.

Block-dimension Strided Loops

- A common practice is to make loops involving threadIdx.x to be block dimension-strided.
- Now, invoking the kernel with any number of threads will give a correct result.

for-loop is now
block dimension-strided

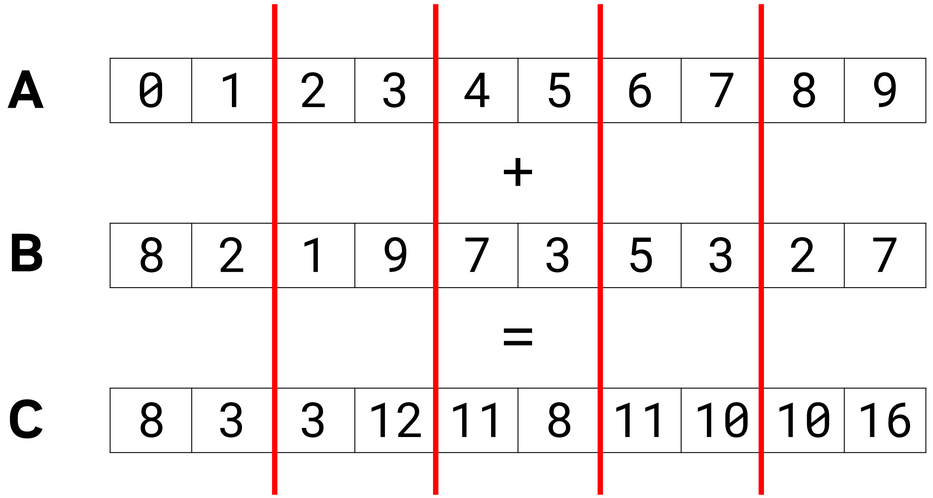
```
__global__ void vector_addition(float* A, float* B, float* C) {
  for (unsigned i = threadIdx.x; i < N; i += blockDim.x) {
    C[i] = A[i] + B[i];
  }
}

int main() {
  ...
  vector_addition<<<1, n>>>(A, B, C);
  ...
}
```

Any number of threads
will yield the same result.

Moving to Multiple Blocks

- We are so far using a single block. We could do better!
- Splitting the work across several blocks will ensure the GPU is better utilized for the task.



- A good size for block dimension is **256**.
  - If it's too few threads, the processors will be underutilized.

Vector Addition Parallelized Across Several Blocks

We now iterate through all threads across all blocks, evenly assigning work.

Stride is adjusted to account for blocks.

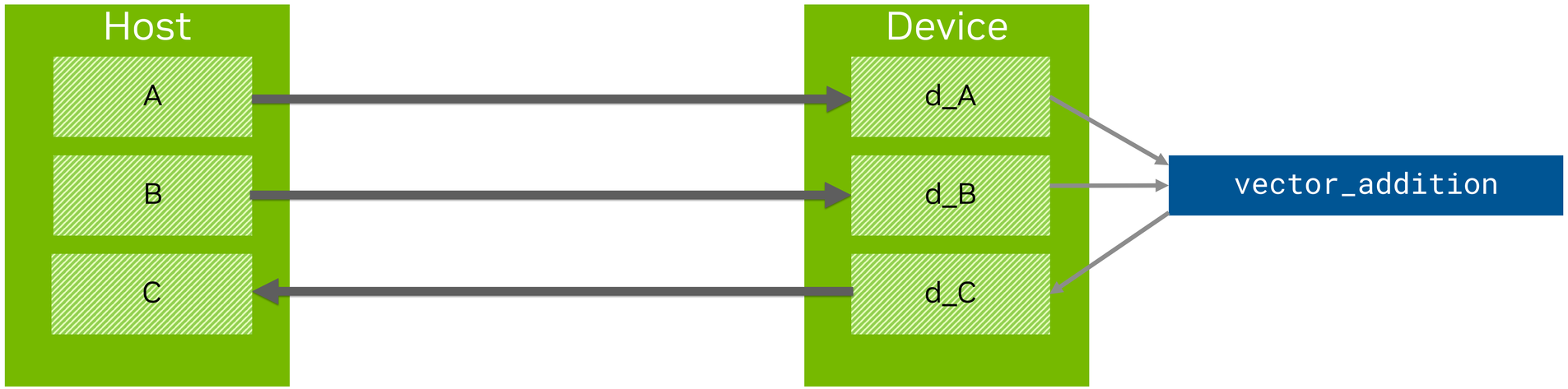
```
__global__ void vector_addition(float* A, float* B, float* C) {
  for (unsigned i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += gridDim.x * blockDim.x)
{
    C[i] = A[i] + B[i];
  }
}

int main() {
  ...
  vector_addition<<<m, n>>>(A, B, C);
  ...
}
```

Grid and block dimensions are configurable at runtime.

What About the Missing Sections?

- We also need to perform **data preparation**, **synchronization** and **data retrieval**:
  1. Allocate memory on the GPU.
  2. Populate inputs.
  3. Invoke kernel.
  4. Synchronize with kernel completion.
  5. Read outputs.



Data Handling Syntax

- There are several manners to control memory.
  - Unified memory allows for a more high-level API where host – device copies occur behind the scenes.
  - **A lower-level API** allows for explicit allocation, deallocation and copying.

- We will use the latter:
  - It assumes less from the user.
  - Allocations and copies are slow processes.
    - Unified memory requires doing prefetching for finer control, harder to get right.
  - They are fully translatable to other languages.

Data Handling Example

1. Allocate memory on the GPU.

```
int main() {
  // Assumes A, B, and C are already allocated. A and B already populated.

  int *d_A, *d_B, *d_C;
  cudaMalloc((void**) &d_A, N * sizeof(float));
  cudaMalloc((void**) &d_B, N * sizeof(float));
  cudaMalloc((void**) &d_C, N * sizeof(float));

  cudaMemcpy(d_A, A, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, N * sizeof(float), cudaMemcpyHostToDevice);
  // Start timer
  vector_addition<<<m, n>>>(d_A, d_B, d_C);

  cudaDeviceSynchronize();
  // End timer
  cudaMemcpy(C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);
}
```

2. Populate inputs.

3. Invoke kernel.

4. Synchronize with kernel completion.

5. Read outputs.

Table of Contents

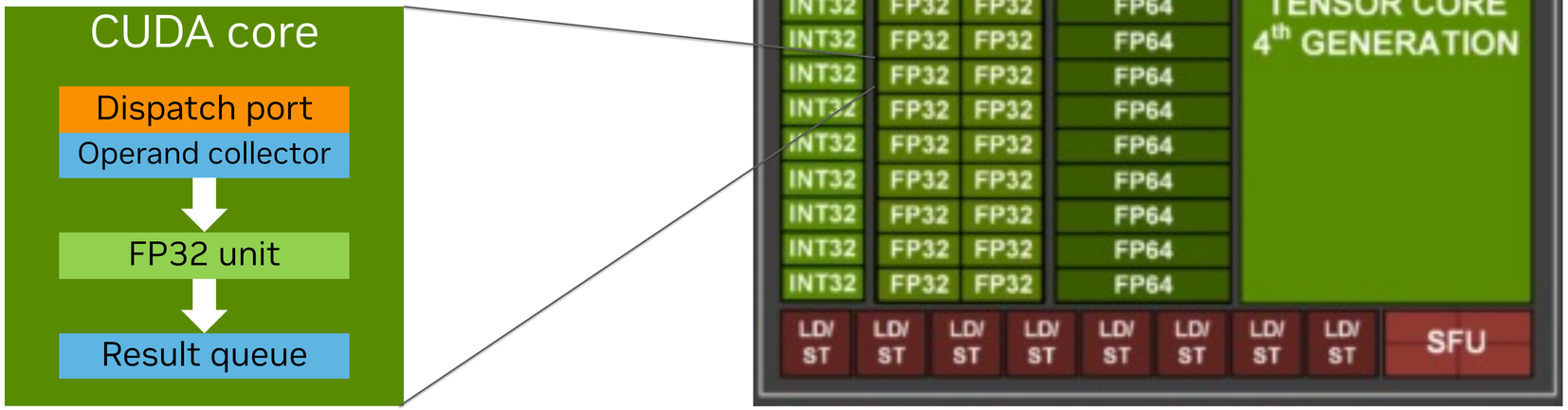
- The CUDA Programming Model
- Host, device and memory
- Writing a kernel
- **GPU architecture**
- Common data parallel techniques
- Summary

The Streaming Multiprocessor

- GPUs are made of processors known as **Streaming Multiprocessors (SMs)**.
- Each SM contains:
  - A small control unit.
  - Many arithmetic units.
  - L1 cache and register memory (more on this later).

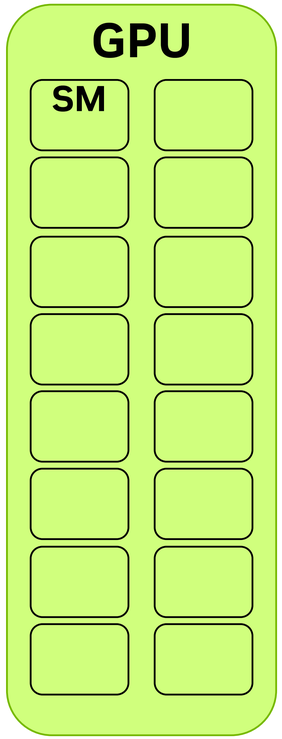
The Streaming Multiprocessor (2)

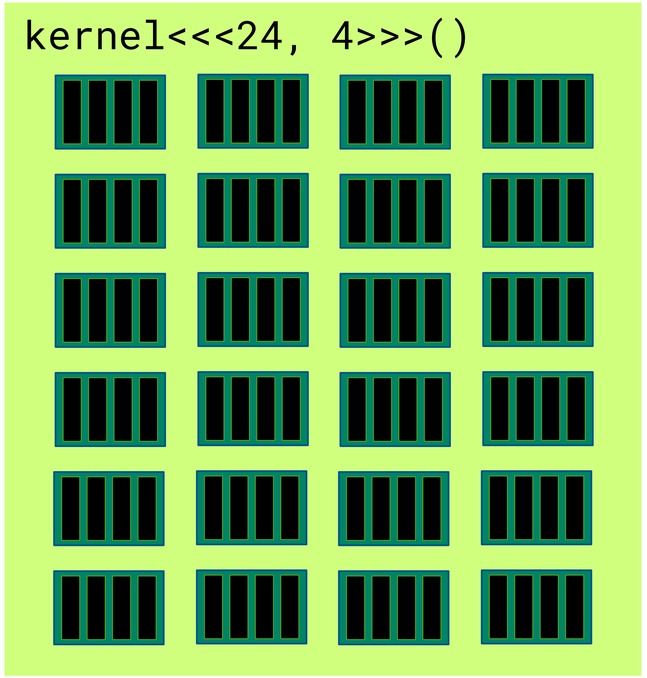
- The heavy lifting is done by CUDA cores:
  - INT32, FP32 , FP64 units and SFUs.
- Tensor cores are processors specialized for AI.
  - They allow faster matrix multiplications + additions.
  - They can also be used with CUDA.



Transparent Scaling

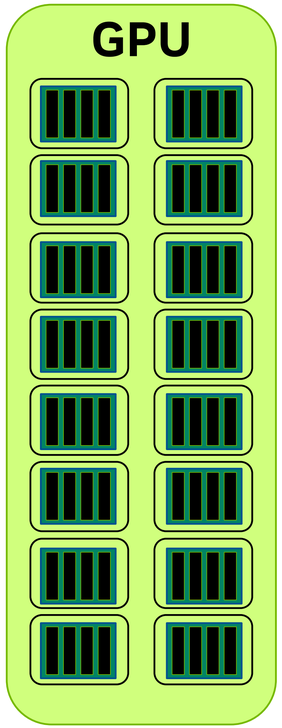
- Blocks of threads are scheduled to run on SMs.

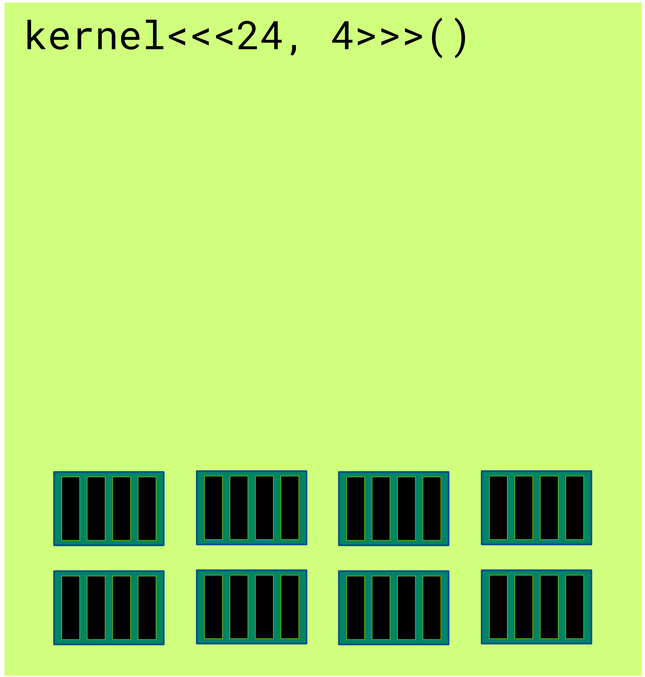




Transparent Scaling (2)

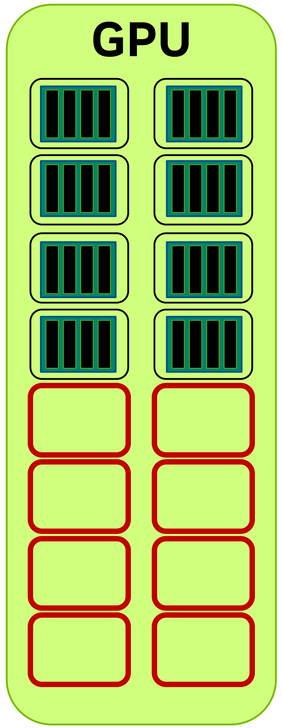
- Depending on the number of SMs, their capabilities, and the requirements of each block, more than one block may be scheduled on a SM.

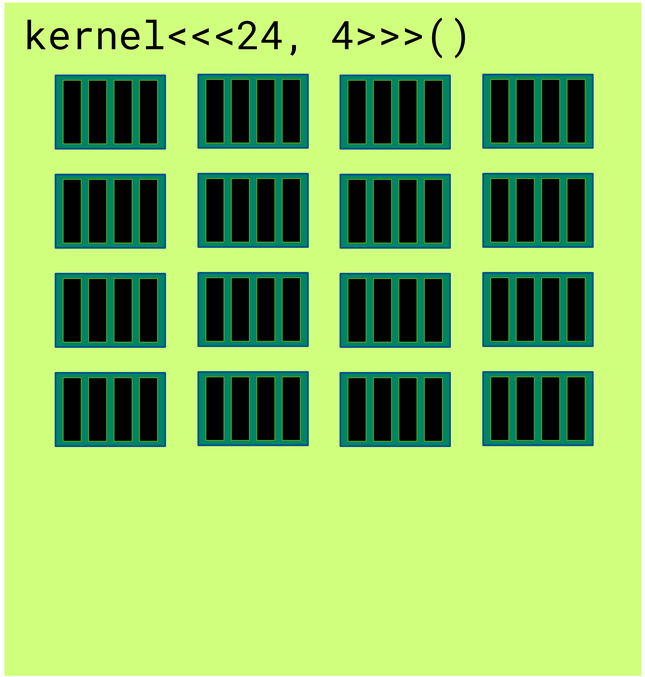




Transparent Scaling (3)

- Invocation configuration that are not multiple of number of SMs lead to a **wave quantization** effect: we don't use the GPU to its fullest some of the time.
  - How relevant this is depends on your use-case.





Saturating the GPU

- We will use Tesla T4s for the tutorials.
  - Tesla T4s have **40 SMs**.
  - Tesla V100s have **80 SMs**.
  - GH200s have **132 SMs**.
- During a kernel invocation, each block of threads is executed *preferably* on a separate SM.
- Hence, a kernel with at least 40 blocks would use all SMs of the Tesla T4.

Saturating the GPU – Slightly More Detail

- There are three parameters that determine how many blocks can be scheduled in parallel:
  - **Invocation configuration** (i.e. number of blocks, number of threads).
  - **Register usage** of the kernel.
  - **Shared memory usage** of the kernel.
- Of course, if the GPU is busy processing other tasks that will also impact the performance of the kernel.
- The CUDA scheduler assigns work to the SMs and manages the GPU resources.
  - In particular, it is possible to runs several kernels asynchronously.
  - Or even several CUDA applications.

Table of Contents

- The CUDA Programming Model
- Host, device and memory
- Writing a kernel
- GPU architecture
- **Common data parallel techniques**
- Summary

Data Parallelism

- GPUs are very efficient for **data parallel workloads**.
  - Perform the same operation across a dataset.
  - As opposed to instruction / thread / process level parallelism.
- **Data dependencies are key factors** to take into consideration!

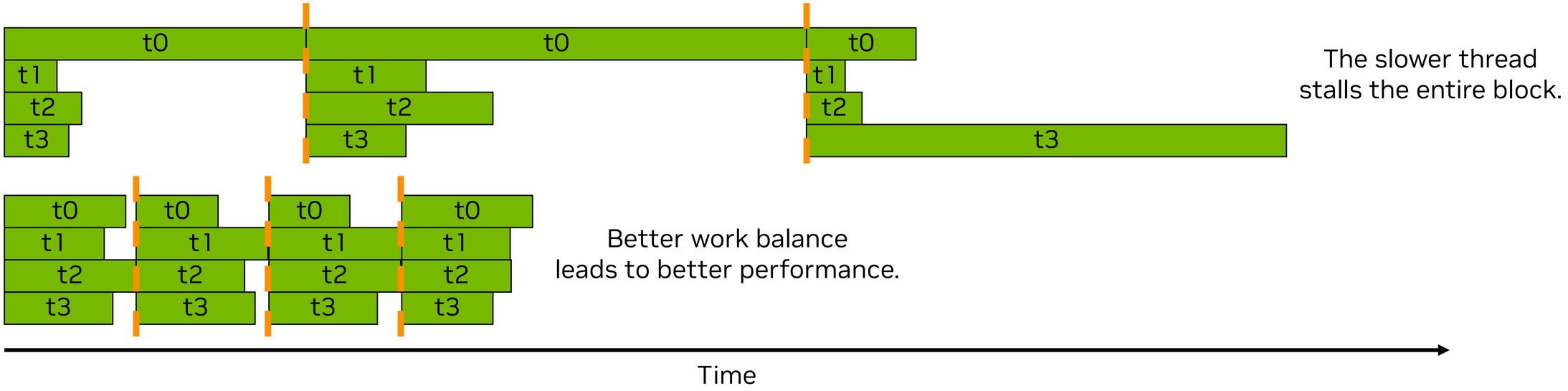
Constructing a tower is not **data parallel**

Image by freepik

Calculating a filter is **data parallel**

Work Balance

- Blocks are small executable pieces that can be carried out by a single SM.
  - Cache hits are more likely.
  - There should be enough work to efficiently use the SM resources.
- Work imbalance is important: **a single thread can stall resources of the entire block**.
  - Ideally all threads should have a similar amount of work to do.



Synchronising Threads

- Threads in a block can be synchronized through the `__syncthreads()` command.
- It acts as a *control flow barrier*, it will wait for all threads to reach that instruction.

```
__global__ void vector_addition(float* A, float* B, float* C) {
  for (unsigned i = threadIdx.x; i < N; i += blockDim.x) {
    C[i] = A[i] + B[i];
  }

  __syncthreads();

  for (unsigned i = 1 + threadIdx.x; i < N; i += blockDim.x) {
    C[i] += C[i - 1];
  }
}
```

Blocks Are Independent

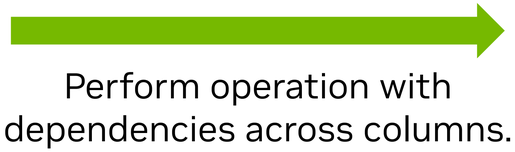
- Blocks cannot communicate to one another.
  - In fact, *it is not guaranteed* that any two blocks will even execute concurrently.
- You can reuse blocks smartly to divide the work considering data dependencies.
  - Use `__synchronize()` and then reassign the role of each thread.
  - Eg. imagine we need to encrypt an image with an encoding that has a dependency across columns:

Assign blocks to sections.

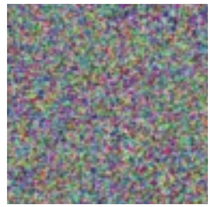
Load section of picture onto shared memory

Use all threads to load pixels in parallel.

Synchronize



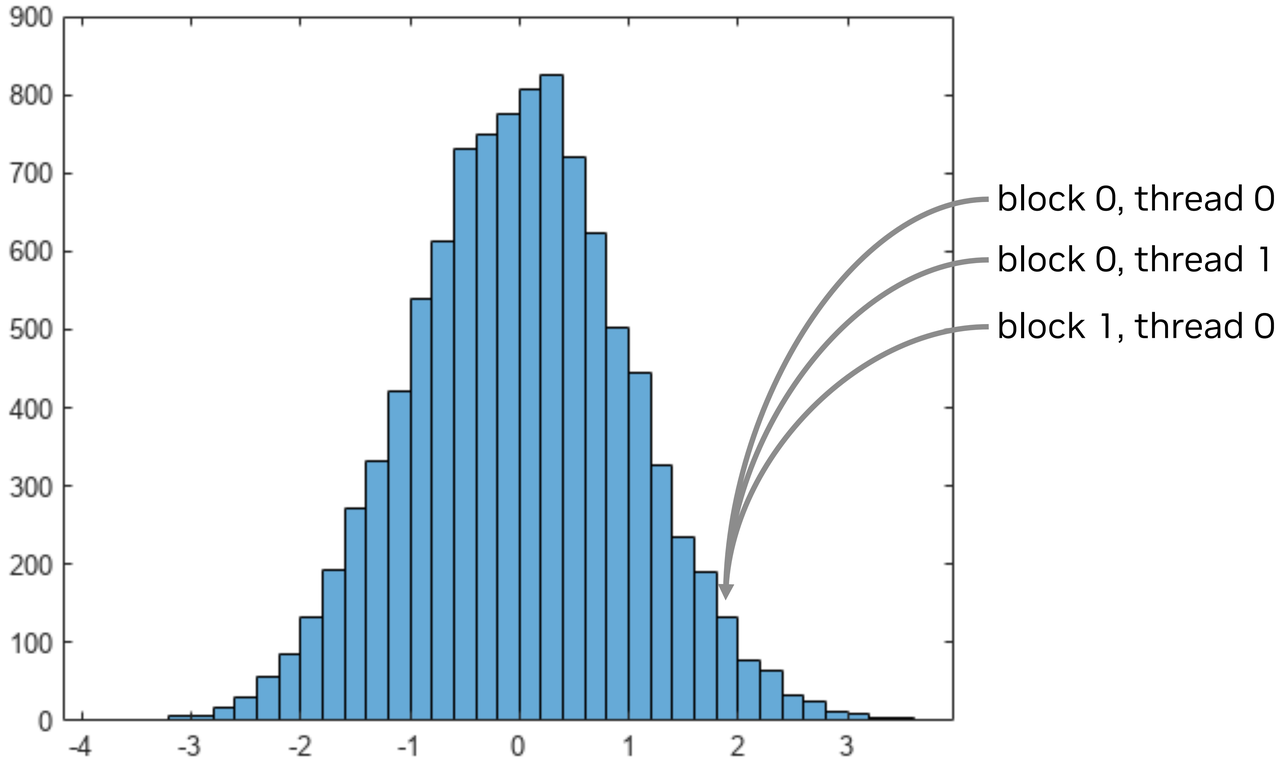
Perform operation with dependencies across columns.



Each thread gets assigned a different row.

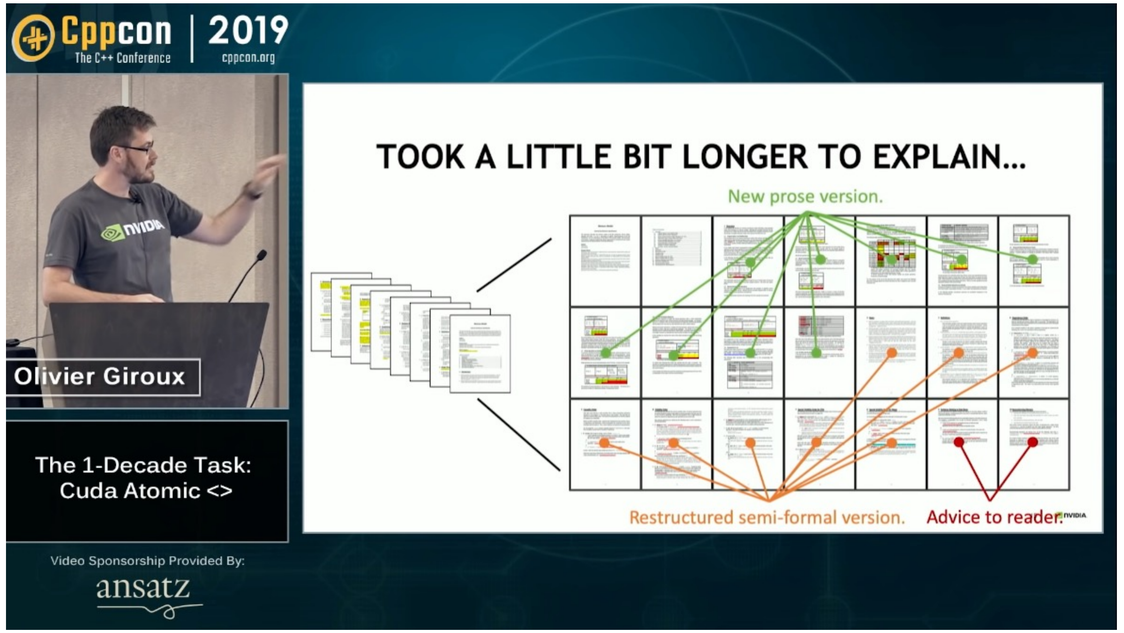
Atomics

- CUDA provides operations that allow atomic accesses over data (on global memory or shared memory).
- Atomic accesses guarantee data will be coherent and prevents race conditions between threads.
- There are many use cases for atomics:
  - Counting elements.
  - Searching eg. elements on an array.
  - Histogramming…
- Atomics can be used over global data that is accessed by several threads or blocks.
  - They can also be used over shared data that is accessed by threads on a block.



Atomics (2)

- Atomics could be potentially very slow if they are abused.
- The good news is that atomics are automatically optimized (both software and hardware).
  - Just because they are needed so often.
- Two syntaxes are supported on CUDA:
  - Functions: `atomicAdd`, `atomicInc`, `atomicOr`, …
  - `cuda::atomic` (it took 10 years to get this one working in CUDA).



`cuda::(std::)atomic`

- `cuda::atomic` behaves like what you would expect from the C++ standard `std::atomic`.
- It also allows to define a scope where the atomic takes effect:
  - The scope could e.g. be a single block, a cluster of blocks or the entire GPU.
  - Hence, it could live in shared memory or global memory.

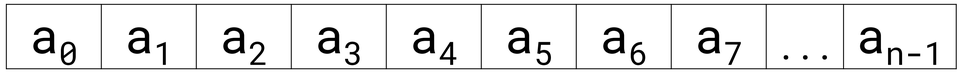
One detail to know about C++ atomics:

- `std::atomic` owns the memory.
  - Cannot be copied.
- `std::atomic_ref` is a lightweight non-owning wrapper around a user-specified memory location.
  - Meant to be passed by value, can be copied.

One Last Example: an Atomic Addition

- Adding together numbers in a floating point array.

A




```
__global__ void vector_addition(float* A, cuda::atomic<float>& sum) {
  for (unsigned i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += gridDim.x * blockDim.x) {
    sum += A[i];
  }
}
```

*Bear in mind: **Floating point** atomics will result in a non-deterministic result!*

Integer atomics don't have this issue.

Table of Contents

- The CUDA Programming Model
- Host, device and memory
- Writing a kernel
- GPU architecture
- Common data parallel techniques
- **Summary**

Summary

- We have gone through the basic building blocks of CUDA.
- The **host** is in charge, the **device** is used for offloading computation.
- *Kernels* are functions invoked on the host, run on the device.
- Computation is divided in blocks and threads.

- Knowing your hardware leads to better software.
  - SMs execute blocks in parallel.
  - Global memory allows for communication with the host and device, and must be preallocated.

- GPUs excel at data parallelism.
  - Identifying the right problem to tackle on GPU is half the work.

Resources Used in the Talk

- GPU Teaching Kit on Accelerated Computing.
- NVIDIA Deep Learning Institute materials.
- CUDA Programming Guide.
- Talk by O. Giroux on *The One-Decade Task: Putting std::atomic in CUDA*.