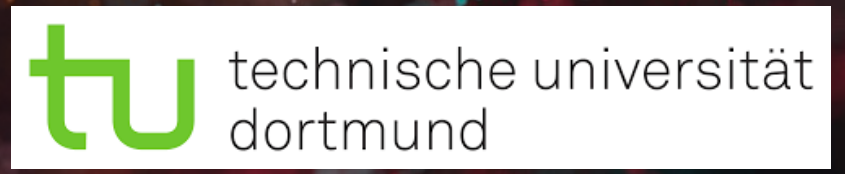
HETEROGENEOUS COMPUTING: GPU PROGRAMMING

Alessandro Scarabotto
TU Dortmund, Germany
Email: alessandro.scarabotto@cern.ch

Fast and Efficient Python Programming School
August 2024
Aachen, Germany



INTRODUCTION

- Bachelor in Physics at Ferrara University (2015-2018)
- Master degree in Physics: double degree Ferrara – Paris Sud (2018-2020)

- PhD in Paris in particle physics working for the LHCb experiment (2020-2023)

- Postdoctoral researcher at TU Dortmund for the LHCb group working on (2023-):
  - Data analysis of beauty and charm decays
  - LHCb trigger system: reconstruction algorithms

OVERVIEW

- Section 1: Heterogeneous architectures and their applications
- Section 2: GPU programming

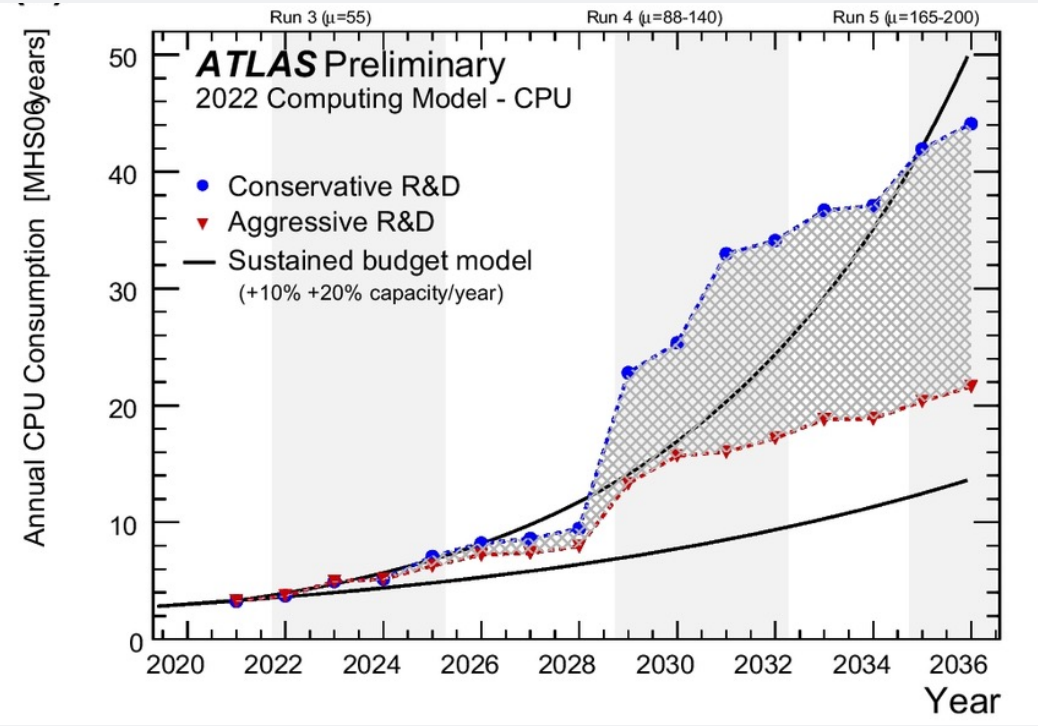
- In Section 1:
  - What does heterogeneous mean?
  - Hardware accelerators
  - Multi-core vs many cores
  - Intro to Graphic Processing Units (GPU)
  - Comparison to other accelerators
  - Examples of GPU using in research

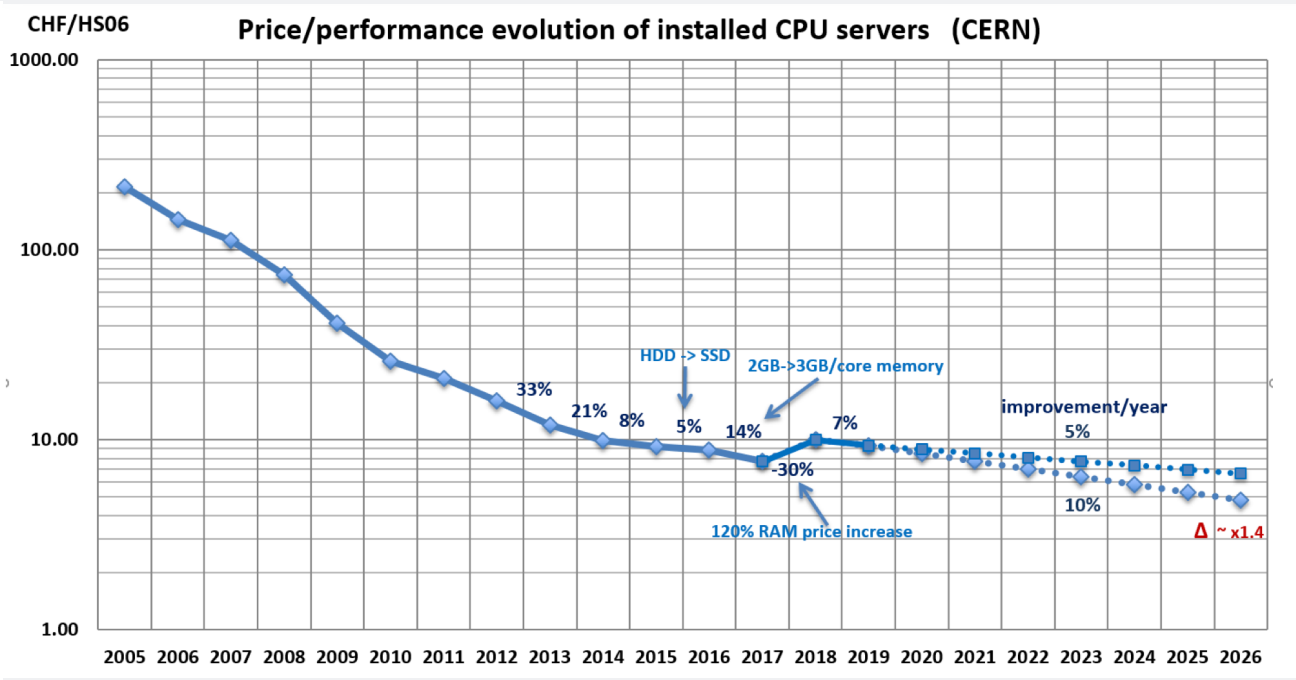
HETEROGENEOUS?

- Systems which can use multiple types of computing cores or processors based on different computer architectures:
  - Central Processing Units (CPUs)
  - Graphic Processing Units (GPUs)
  - Application-Specific Integrated Circuits (ASICs)
  - Field Programmable Gate Arrays (FPGAs)
  - Neural Processing Units (NPUs)
  - Tensor Processing Units (TPUs)
- Different processors specialized for specific purposes
- Goal is to optimise **computing performance** and **energy efficiency**

COMPUTING PERFORMANCE

- Increasing performance as a function of time not sustaible as a function of price (exponential budget)
- Example showing ATLAS experiment CPU requirements at CERN
- Assuming a constant budget per year, only-CPU model is not sustainable
- Must exploit the "power" of heterogeneous systems in scientific applications



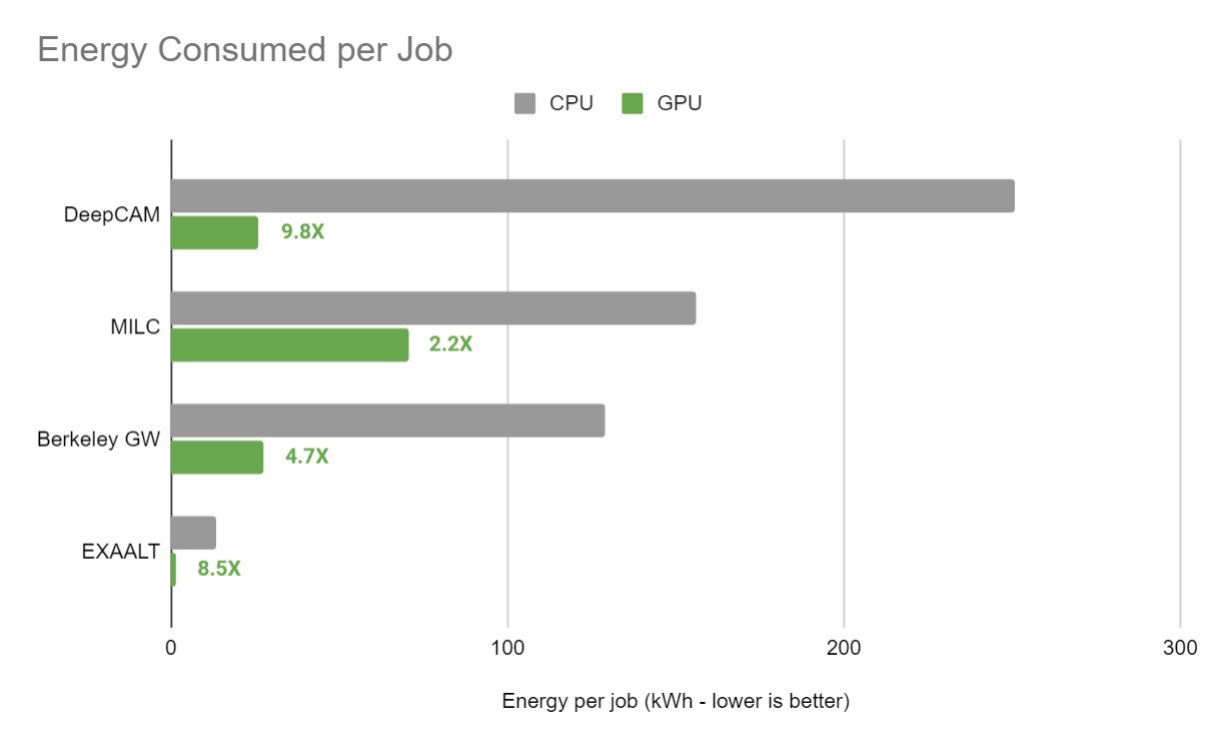


Courtesy Dr. Bernd Panzer-Steindel (CERN/IT, CTO)

J. Eschle et al. Computing and Software for Big Science, Vol7, A10

ENERGY EFFICIENCY

- More and more important to reduce electricity consumptions and environmental impacts
- Giving power to processors could be more expensive than buying them
- Heterogeneous computing can help improving energy efficiency
- We need to be careful of the definition of "power consumption" as many factors come into play (power delivered, cooling systems, average vs peak consumption, …)



Heterogeneous energy consumption comparison

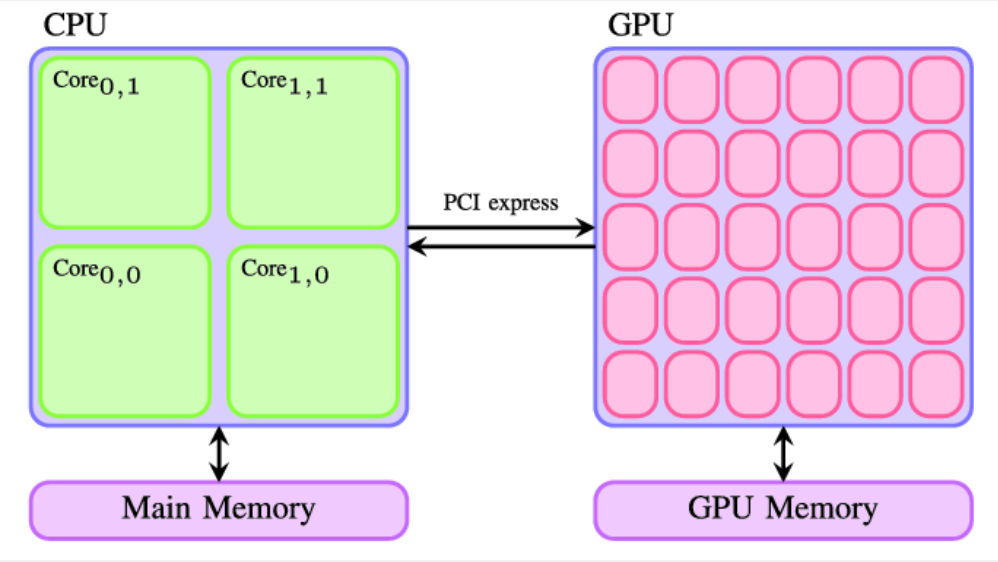
HETEROGENEOUS SYSTEMS

- Common in our daily life: video encoding and editing, graphics rendering, …

GPU accelerated rendering

HETEROGENEOUS SYSTEMS

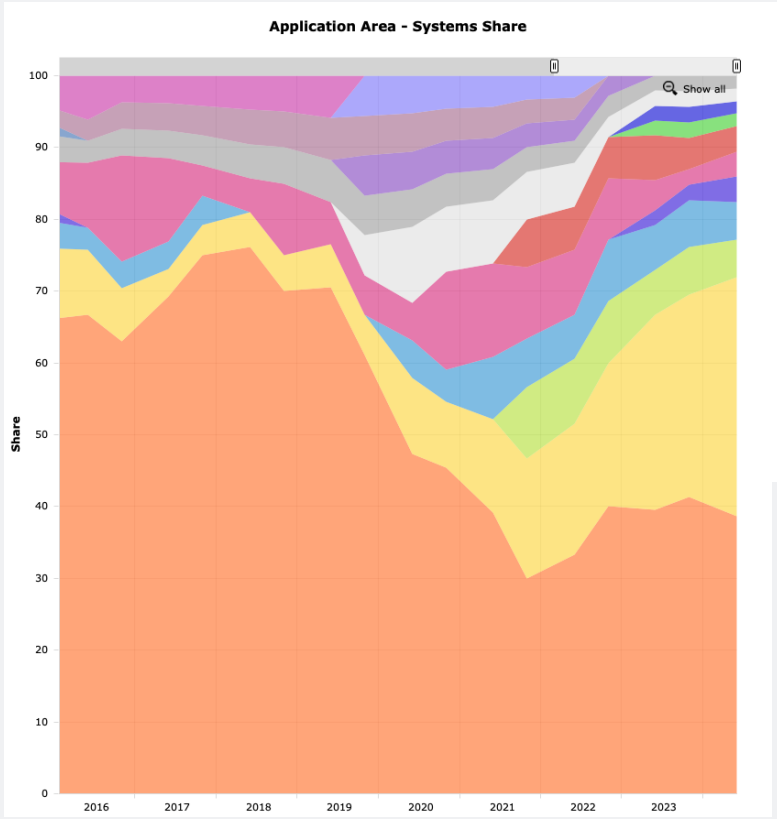
- Common in our daily life
- Used also by the top data centers in the world in many different areas
- You can take a tour in top500.org showing the top500 computing systems in the world
- Most of them use NVIDIA or AMD GPU accelerators showing the need of heterogeneous systems for top performance



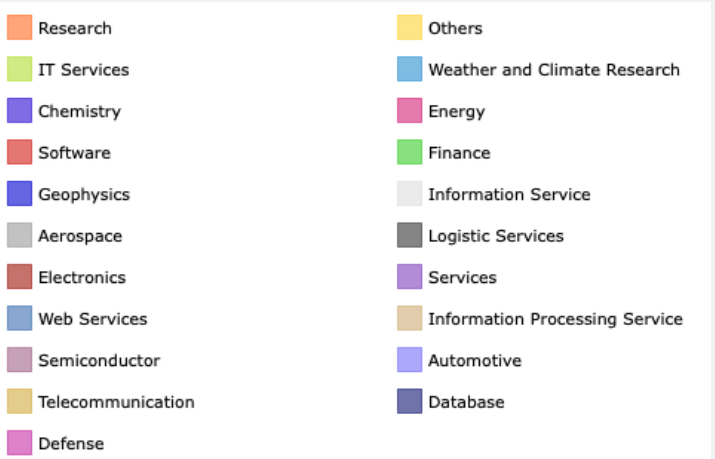
Multi-core vs many cores?

C. Feng at al. (2015). arXiv:1208.4247

HETEROGENEOUS SYSTEMS



https://www.top500.org/statistics/overtime/



- Also most of the data centers in the GREEN500 list use accelerators

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	1,305,600	270.00	353.75	5,194
7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159
9	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
10	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, Nvidia NVIDIA Corporation United States	485,888	121.40	188.65	

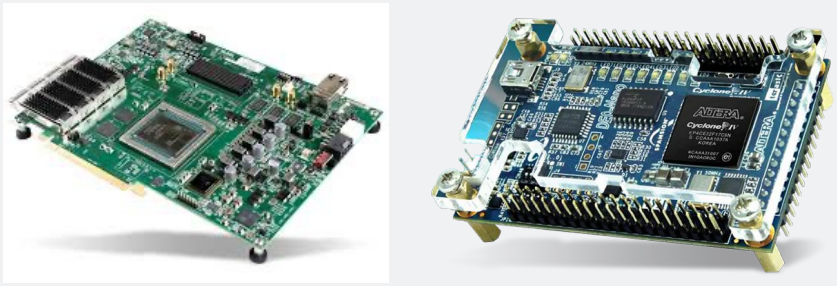
HARDWARE ACCELERATORS IN RESEARCH

- Tradeoffs between flexibility and single-task optimised performance

General purpose

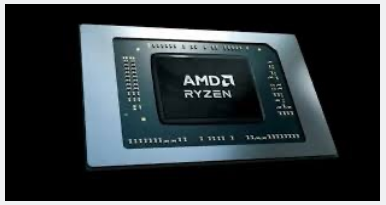
Graphic Processing Units (GPUs)
Vendors: NVIDIA, AMD, Intel

Field Programmable Gate Arrays (FPGAs)
Vendors: Xilinx, Altera



Dedicated

Neural Processing Units (NPUs)
Vendors: AMD, Intel, …
Processor specialised in AI and ML



Tensor Processing Units (TPUs)
Vendor: Google
ASIC specialised in NN machine learning

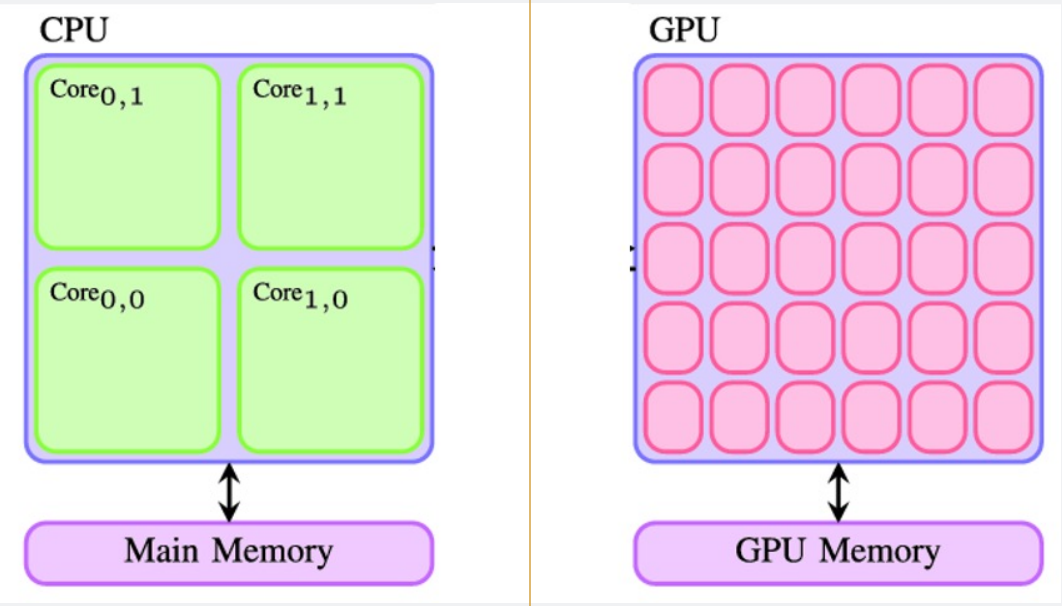
MULTI-CORE VS MANY CORES

Multi-core CPU

- O(10) cores
- Flexibility of sequential and parallel code programming
- Large caches (fast memory storage)
- Focused on single-thread high performace

GPU with many cores

- O(1000) cores
- Designed for parallel code programming
- Small caches
- Focused on operation of simpler calculations per single-thread



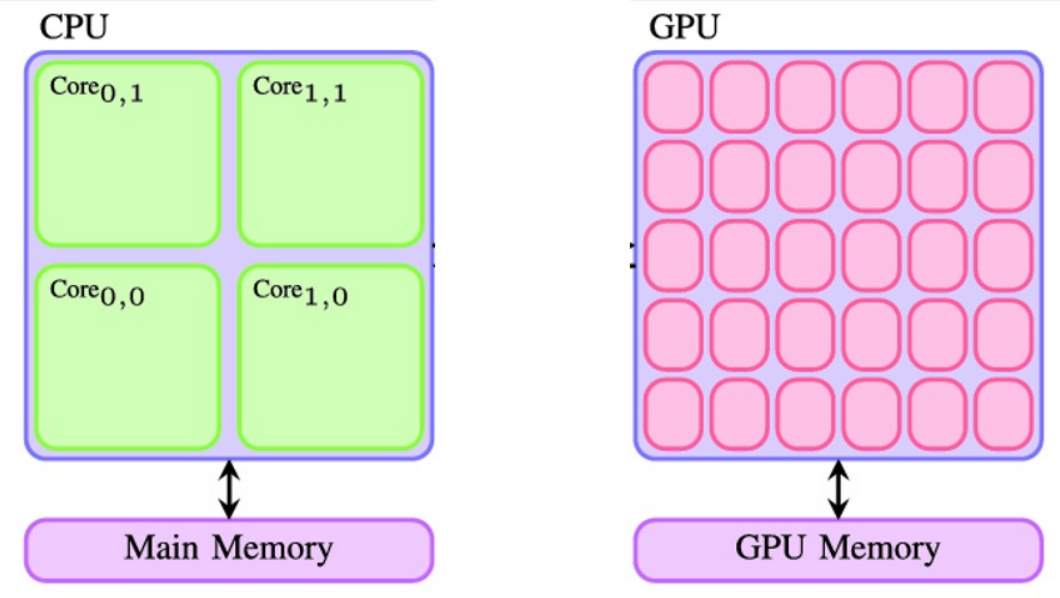
WORKLOAD: MULTI-CORE VS MANY CORES

Multi-core CPU

- Typically the main processor
- Best sequential performance
- Multi-threading optimisation needed in parallelizable problems

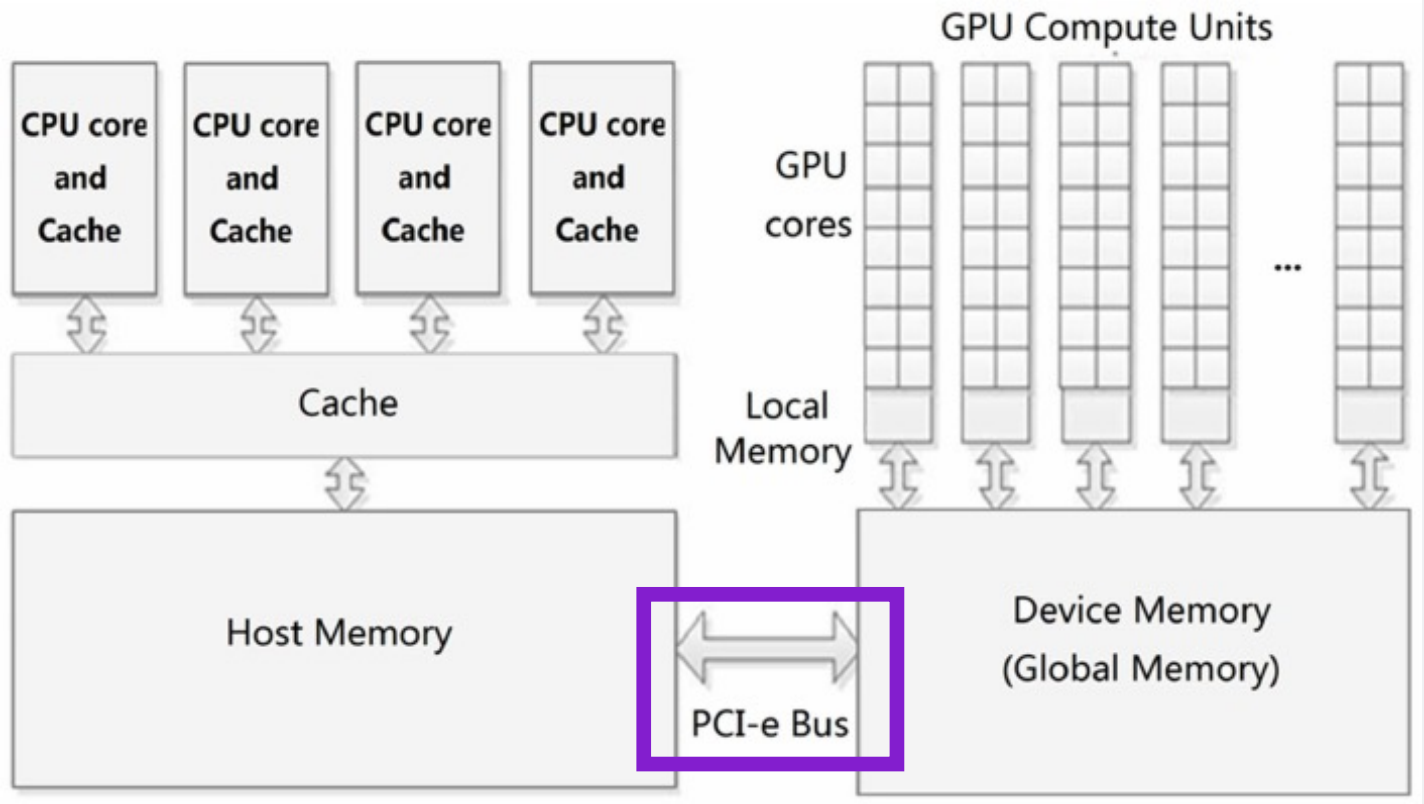
GPU with many cores

- Usually paired with a CPU
- Algorithms optimised to profit from the many cores of the accelerator
- Only highly parallelizable problems



GRAPHIC PROCESSING UNIT (GPU)

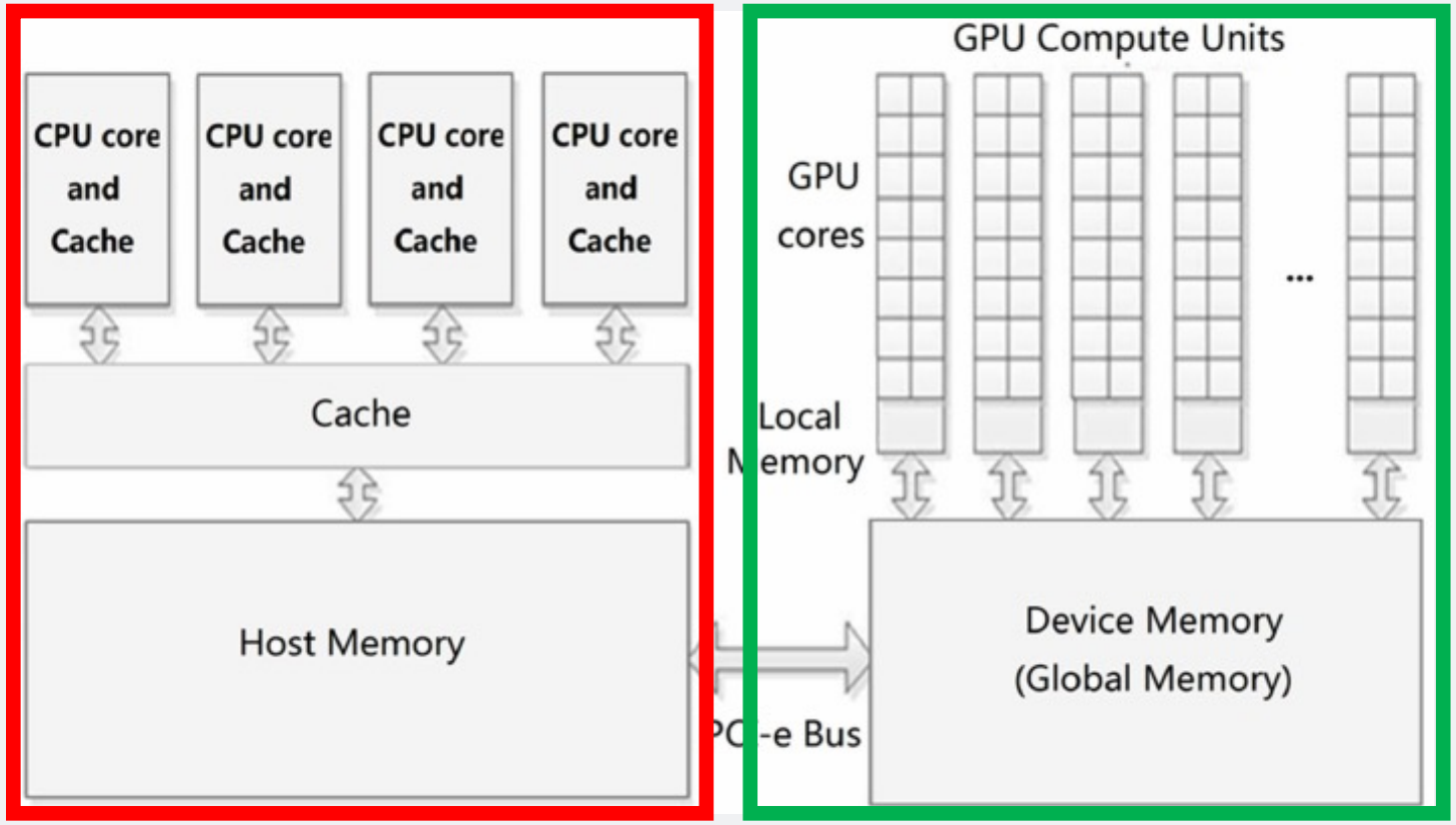
- GPU were first developed for graphics pipelines only
- Now general purpose processors (often used for AI applications)
- Programmed with high-level language
- Usually, CPU is the main processor with GPU as accelerator
- PCIe connection allows high throughput (up to 16 GB/s per lane)



PLOS ONE 8(5): e62789

HETEROGENEOUS SYSTEM

- CPU core, latency optimised (= low delay in transferring data):
  - Low number of cores
  - Complex control units
  - Large caches
- GPU accelerators, throughput optmised (= high faction of data trasferred simultaneously):
  - High number of cores
  - No complex control units
  - Small caches



PLOS ONE 8(5): e62789

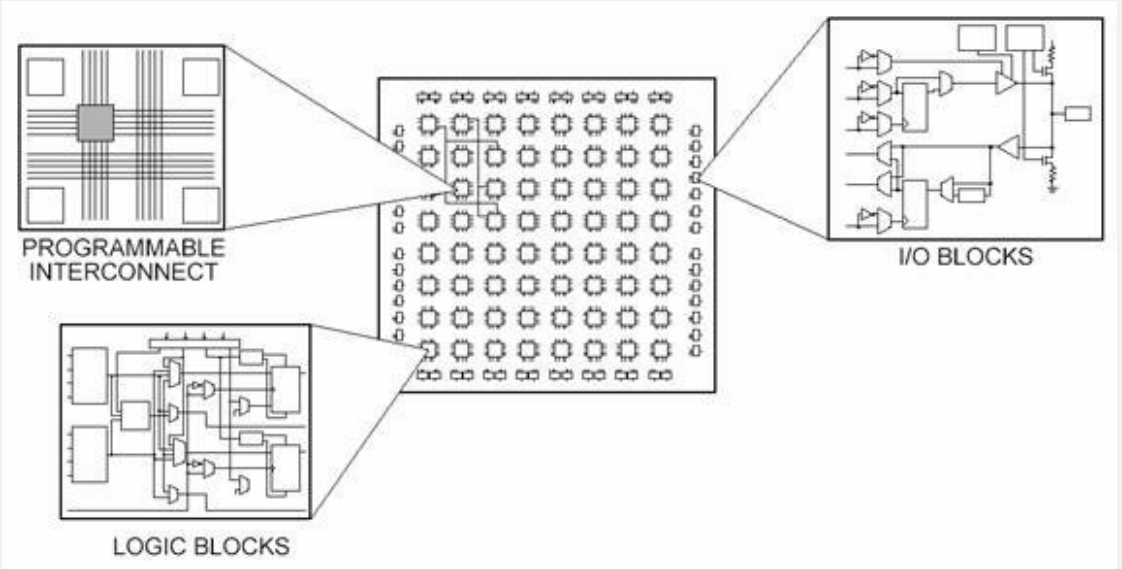
GPU VS CPU: SPECIFICATIONS EXAMPLE

- GPUs provide higher data transfer speed (bandwidth), meaning also a higher number of floating points operations per second (FLOPS)
- CPUs compute more instructions per second (frequency) exploiting a larger memory capacity

	Core count	Bandwidth	Peak Compute performance	Frequency	Memory capacity	Transistor count	Price
CPU AMD Ryzen 5 5600G	6	48 GB/s	1.7 TFLOPS	3900 MHz	64 GB	10.7 M	260 $
GPU NVIDIA RTX 3090	10496	936 GB/s	35.5 TFLOPS (single precision)	1395 MHz	24 GB	28.3 M	1500 $

FIELD PROGRAMMABLE GATE ARRAYS (FPGA)

- Thousands of logic blocks connected via programmable interconnect
- Hardware implementation of an algorithm
- Advantages:
  - Fast integer computations (low latency)
  - Does not require a CPU (any data source)
  - High bandwidth
- Disadvantages:
  - Medium floating point operations performance
  - High engineering cost
  - Not easy backward compatibility with other processors types



National Instruments

ACCELERATORS COMPARISON

	CPU	GPU	FPGA
Latency	$\mathcal{O}$ (10) µs	$\mathcal{O}$ (100) µs	Deterministic, $\mathcal{O}$ (100) ns
I/O with processor	Ethernet, USB, PCIe	PCIe, Nvlink	Connectivity to any data source via printed circuit board (PCB)
Engineering cost	Low entry level (programmable with c++, python, etc.)	Low entry level (programmable with CUDA, OpenCL, etc.)	Some high-level syntax available, traditionally VHDL, Verilog (specialized engineer)
Single precision floating point performance	$\mathcal{O}$ (10) TFLOPs	$\mathcal{O}$ (10) TFLOPs	Optimized for fixed point performance
Serial / parallel	Optimized for serial performance, increasingly using vector processing	Optimized for parallel performance	Optimized for parallel performance
Memory	$\mathcal{O}$ (100) GB RAM	$\mathcal{O}$ (10) GB	$\mathcal{O}$ (10) MB (on the FPGA itself, not the PCB)
Backward compatibility	Compatible, except for vector instruction sets	Compatible, except for specific features only available on modern GPUs	Not easily backward compatible

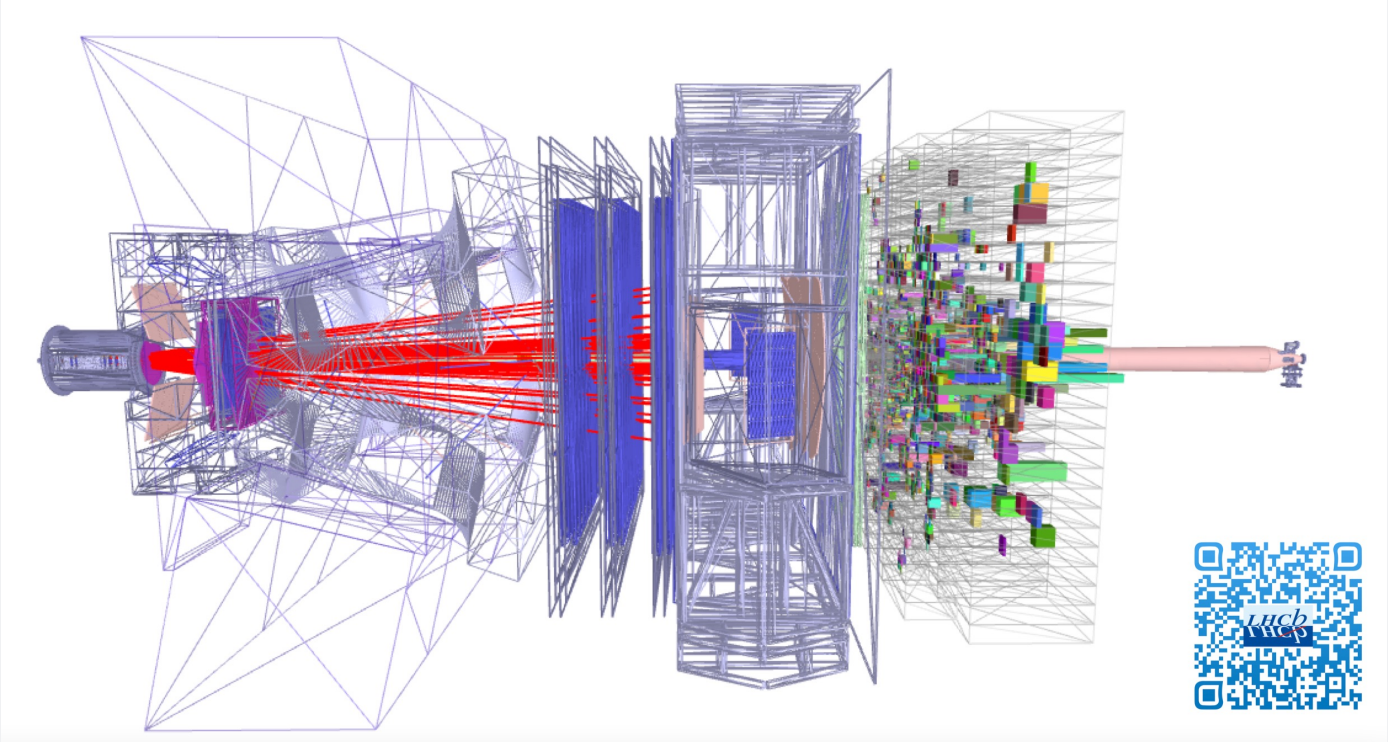
JINST 15 C06010 (2020)

CHALLENGES WITH HETEROGENEOUS COMPUTING

- Different challenges may arise when exploiting the heterogeneous computing:
  1. Instruction sets can produce results which are bit-wise not reproducible
     - Check in advance minimum required resolution (integer, floating point, …)
  2. Slow interconnects can cause bandwidth bottlenecks:
     - Try to minimize copies between devices
  3. Data layout might not be suitable for all devices architectures or memory structures:
     - Minimize transformations between data layouts
  4. Different compilers and/or programming interfaces:
     - Use programming environments for heterogeneous computing

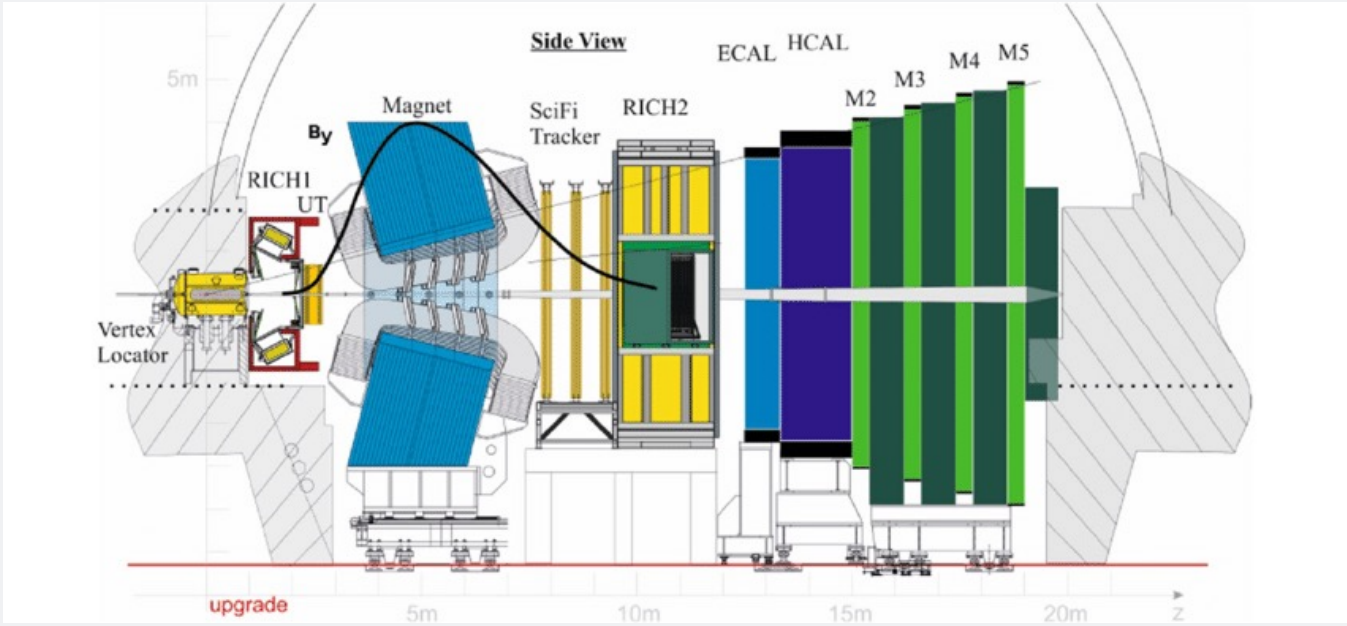
EXAMPLES OF GPU USAGE IN RESEARCH

- Trigger at high energy physics experiments
- The LHCb experiment is taking data with fully-software trigger: first-level based on GPUs (HLT1)
- Why trigger? And why GPUs?



TRIGGER: REAL TIME ANALYSIS

- At CERN, the Large Hadron Collider (LHC) provides the LHCb experiment 40 million proton-proton beams collisions per second → 4 TB of data per second
- Trigger = filtering and selection of the events (= beam-beam collisions)
- Select decay of particles containing b- and c- quark, signatures: displaced vertices, momentum, particle type
- Sub-detectors which allow tracking and particle identification, which needs to be done "live": Real-Time Analysis (RTA)



R. Aaij *et al* 2024 *JINST* **19** P05065

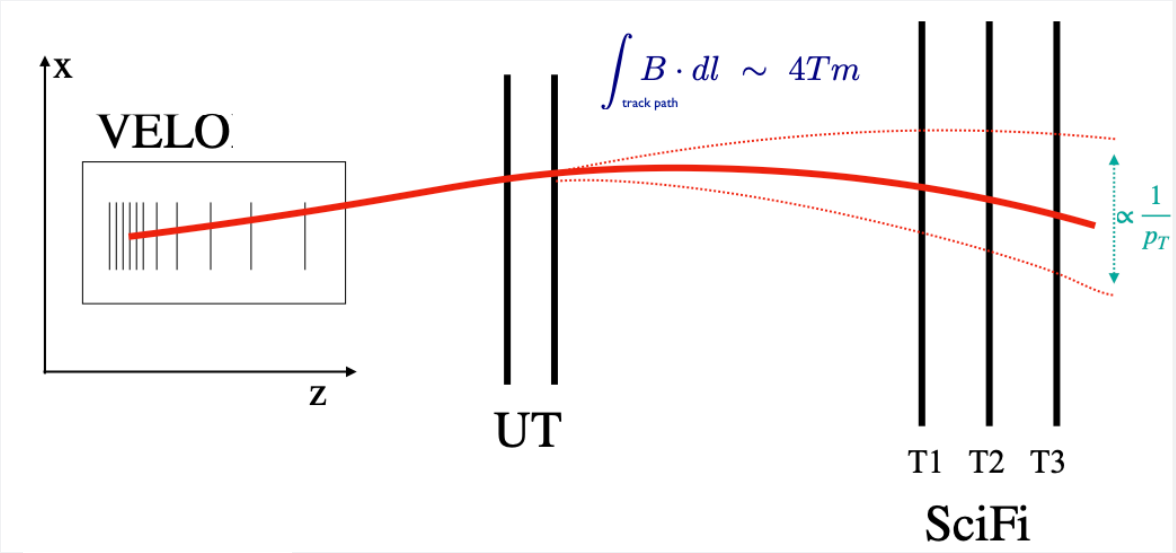
THE LHCB TRIGGER: GPUS

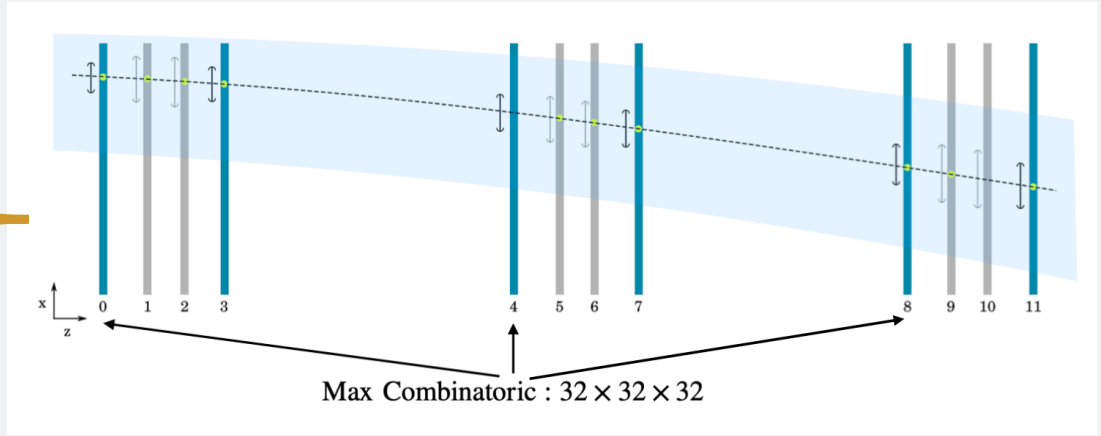
- Goal: perform the reconstruction for each of the events in the 4 TB/s and reduce it of a factor ~40 reaching 100 GB/s (HLT1)
- Architecture optimised on throughput → GPUs

The LHCb trigger	GPUs
Huge data load	High throughput Many FLOPS
Parallel problems: pp collisions (event) Within one event: tracks	Highly parallelizible
Small raw event data (~100KB)	PCIe connection -> limited I/O ~1000 events fit in GPU memory O(10)GB

RECONSTRUCTION ALGORITHM

- How to fully exploit the parallelization power of GPUs?
- Goal: reconstruct tracks traversing the whole LHCb detector, fundamental for triggering
- Parallelization levels:
  1. Over events, independent p-p collisions
  2. Over input tracks, extrapolate straight tracks in VELO+UT into the magnetic field reaching the SciFi
  3. Over hits in SciFi, meaning possible extrapolations segments





arXiv:2402.14670

MACHINE LEARNING

- The training of machine learning (ML) methods require large amounts of data to handle → high throughput of GPUs
  - Many ML methods, for example neural networks, use very parallelizible methods: matrix multiplication
  - Can be trained using reduced precision
- Artificial intelligence (AI) uses neural networks for fast training and inference from input data
- GPU performance has increased around 7000 times since 2003, also in terms of price per performance

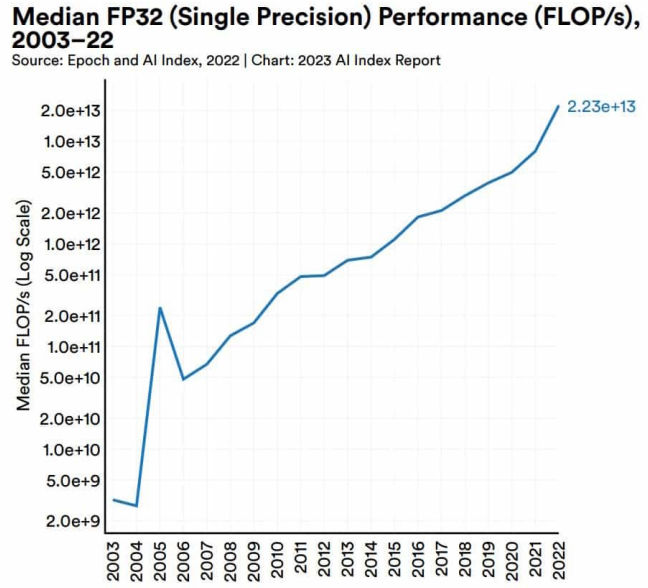
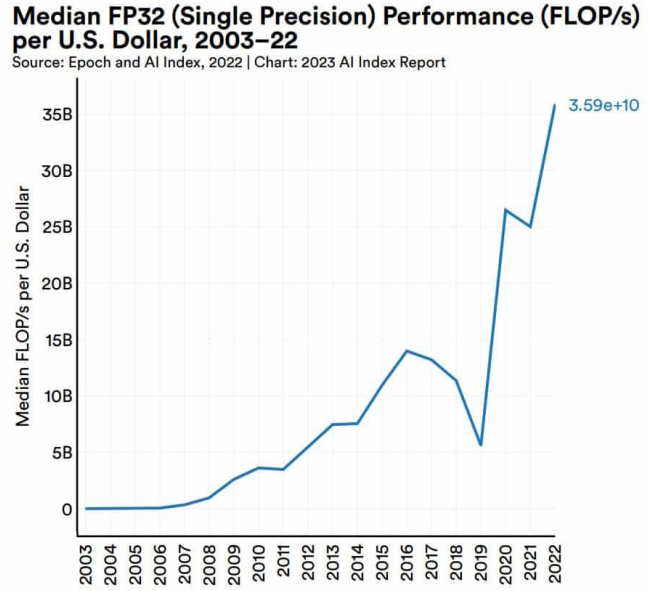


Figure 2.7.8



GPUs for AI

Figure 2.7.10

SUMMARY OF FIRST PART

- Heterogeous computing is critical to improve performance and energy efficiency of our code system
- Commonly used in our day-by-day life (video/graphics) but also by top computing clusters in the world
- Using heterogenous computing in research: General Purpose GPU
- GPUs are throughput-optimised processors vs CPU which are latency-optimised
- Examples of research applications: high energy physics trigger system, machine learning, …

END OF FIRST PART

ANY QUESTIONS?

BREAK?

LET'S DIVE INTO GPU PROGRAMMING

BUR BEFORE...

REGISTER TO THE VISPA CLUSTER:

https://vispa.physik.rwth-aachen.de

Just need username and email

OVERVIEW

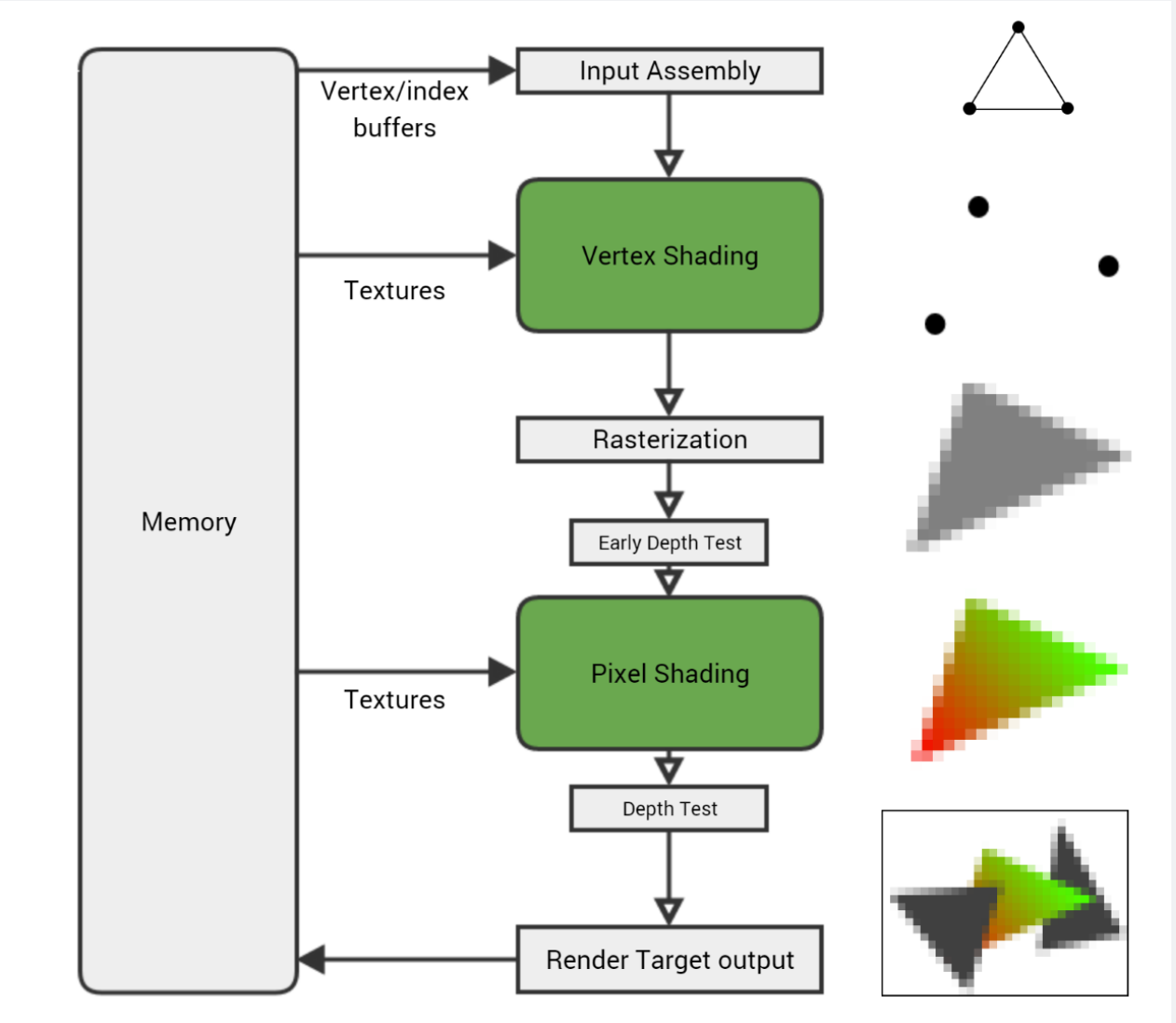
- Section 1: Heterogeneous architectures and their applications
- Section 2: GPU programming

- In Section 2:
  - GPU for graphics
  - GPU for general purpose
  - Composition and parallelization
  - Memory layout
  - Functions declaration and a first CUDA kernel
  - Parallelization in CUDA and memory management

GPU AS GRAPHIC PROCESSORS

Step by step of how to produce graphics:

1. **Vertex and index inputs**: description of the images using vertices and edges to triangles
2. **Vertex shading**: calculate the final position on the screen of each vertex
3. **Rasterization**: get pixel-by-pixel colors
4. **Pixel shading**: transform color of the pixels based on the textures (material, light, …). This is usually the most GPU expensive step
5. **Rendering**: write output to final render target



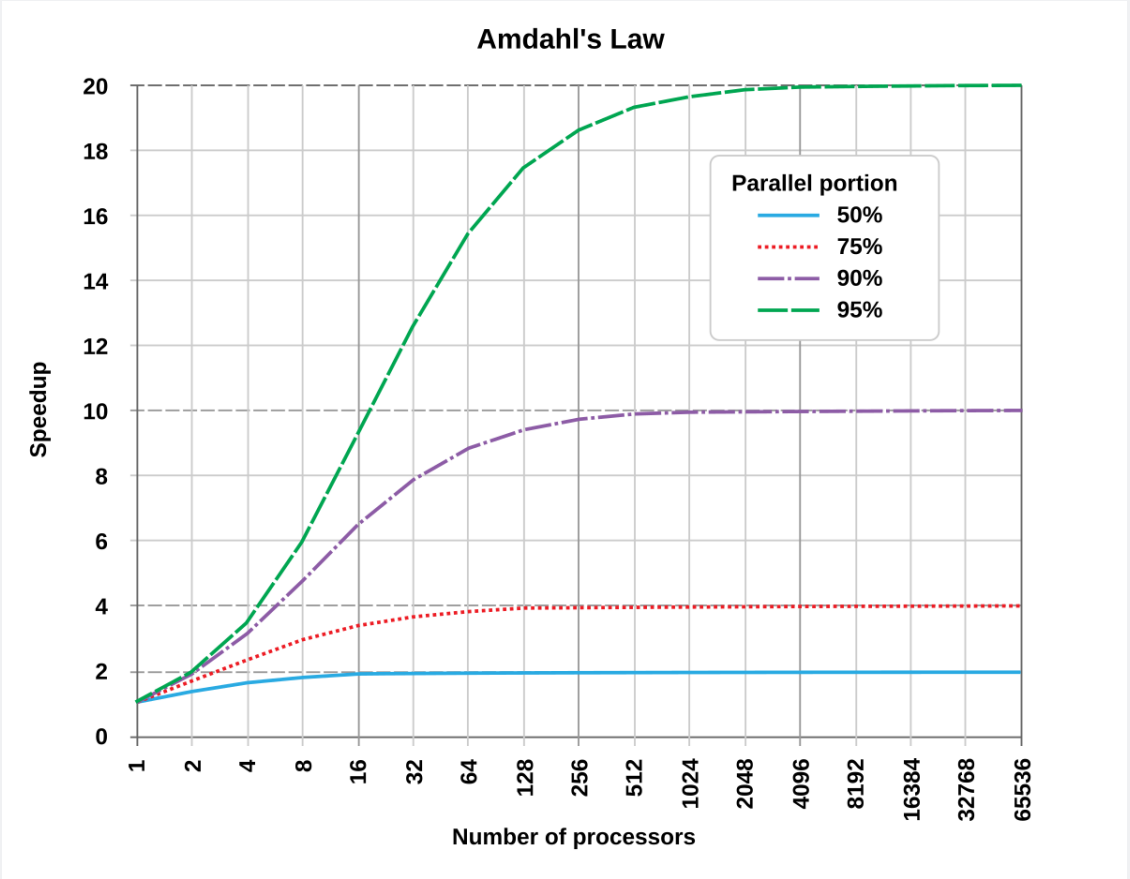
http://fragmentbuffer.com/gpu-performance-for-game-artists/

GRAPHICS REQUIREMENTS

- Graphics pipelines compute a huge amount of simple arithmetics on independent data
  - Transforming positions, get pixel colors, apply texture properties, …
- Hardware-wise:
  - Memory should be accessed simultaneously and contiguosly (no need of huge memory capacity)
  - Bandwidth and throughput far more important than latency
  - Floating-point precision needed
- GPU processors have the perfect requirements!

USING GPUS FOR GENERAL PURPOSE

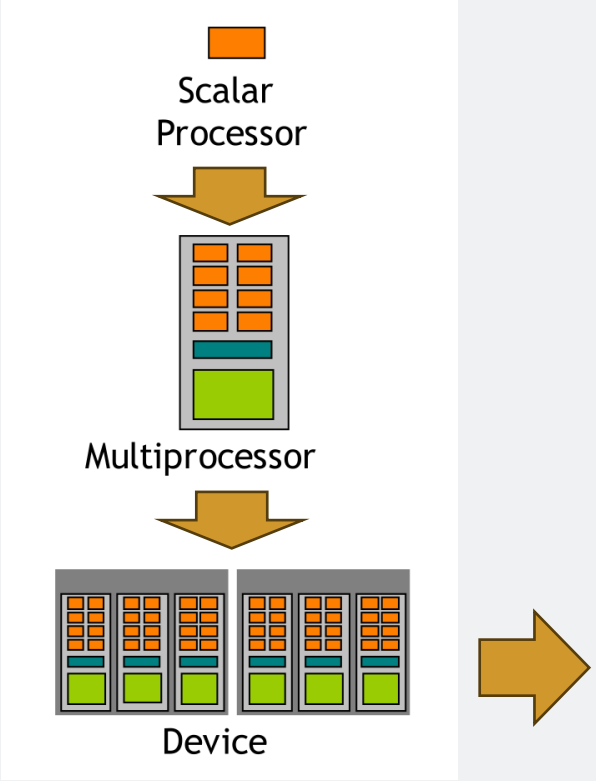
- Starting from around 2000s, with advent of programmable shaders and floating point operation support, GPU processors became popular also as General Purpose (GPGPU) systems
- When is it beneficial? **Amdahl's law**
- Speedup = 1/(S + P/N)
  - S: sequential part
  - P: parallel component
  - N: number of processors
- One most consider how much of the algorithmic problem can be parallelized
- Example: if 95% of the algorithm can be parallelized the gain could be up to 20 times

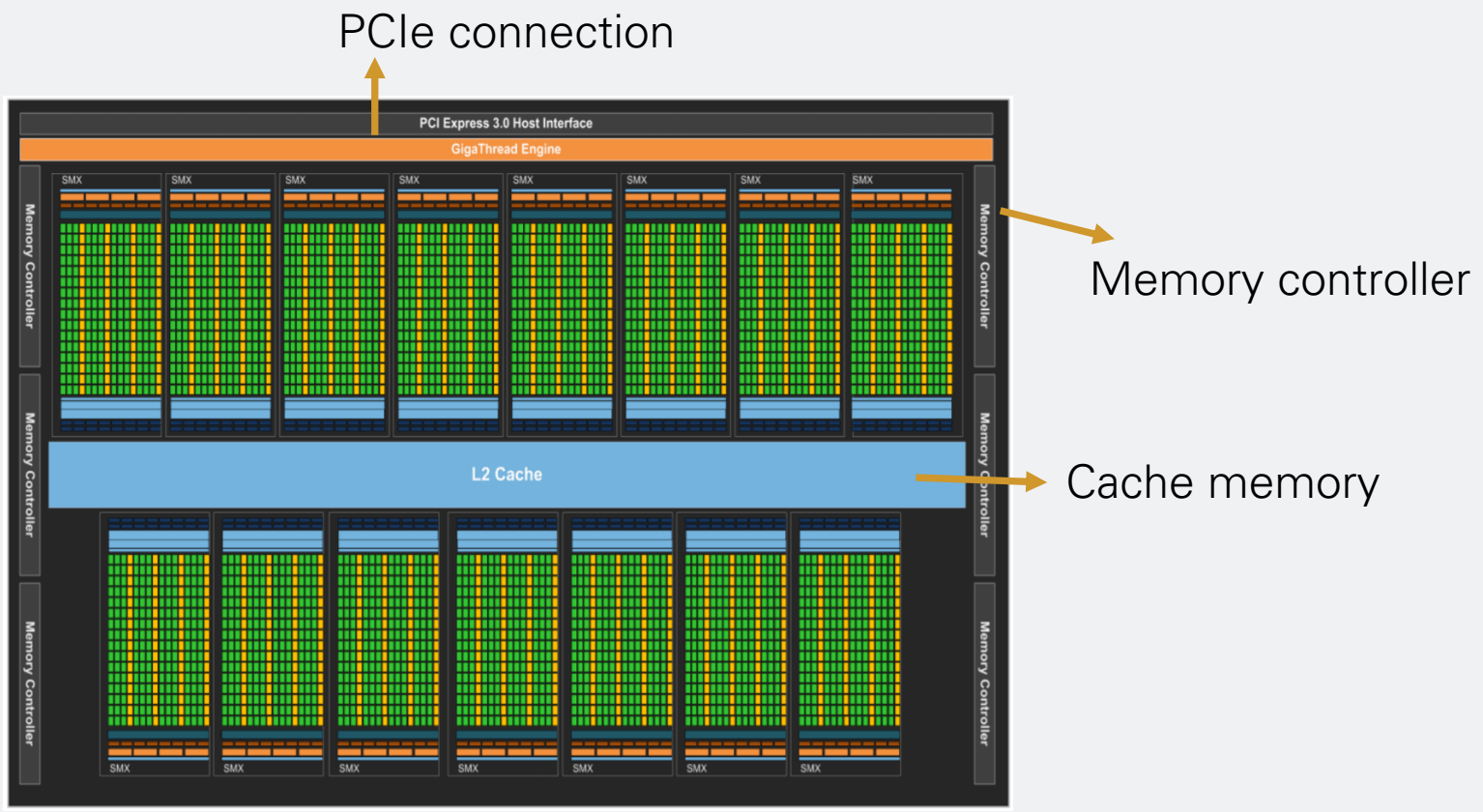


wikipedia

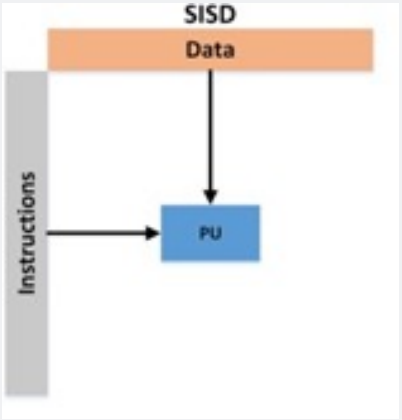
GPU COMPOSITION: HARDWARE

- GPU consists in elements which can perform Single Instruction in Multiple Threads (SIMT)

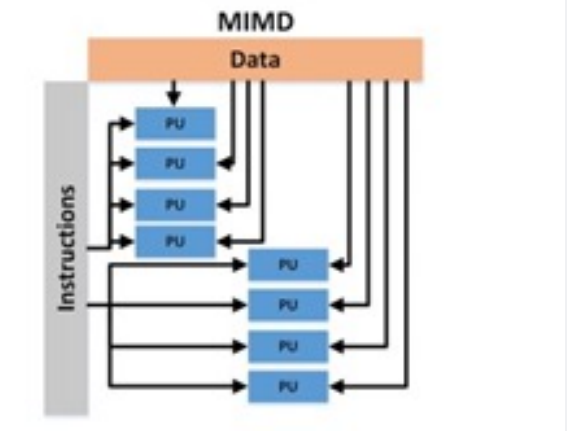




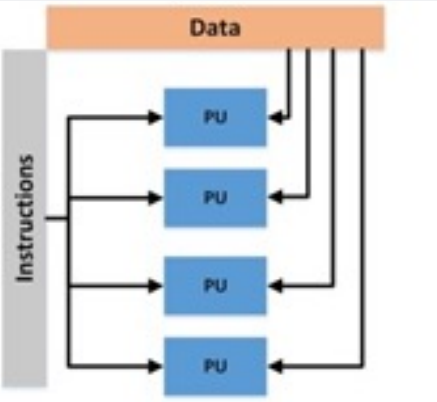
SISD, MIMD AND SIMT



- **SISD**: Single Instruction Single Data → Uniprocessors machines



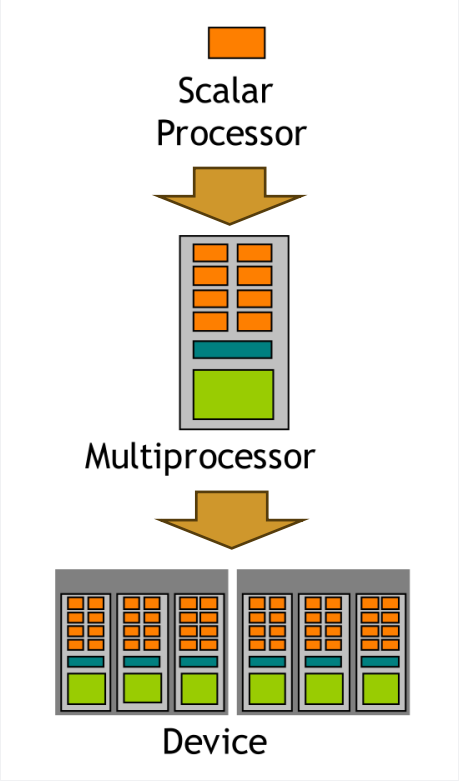
- **MIMD**: Multiple Instructions Multiple Data → Multi-core or grid processors
  - Vectorised instructions, as in modern CPUs

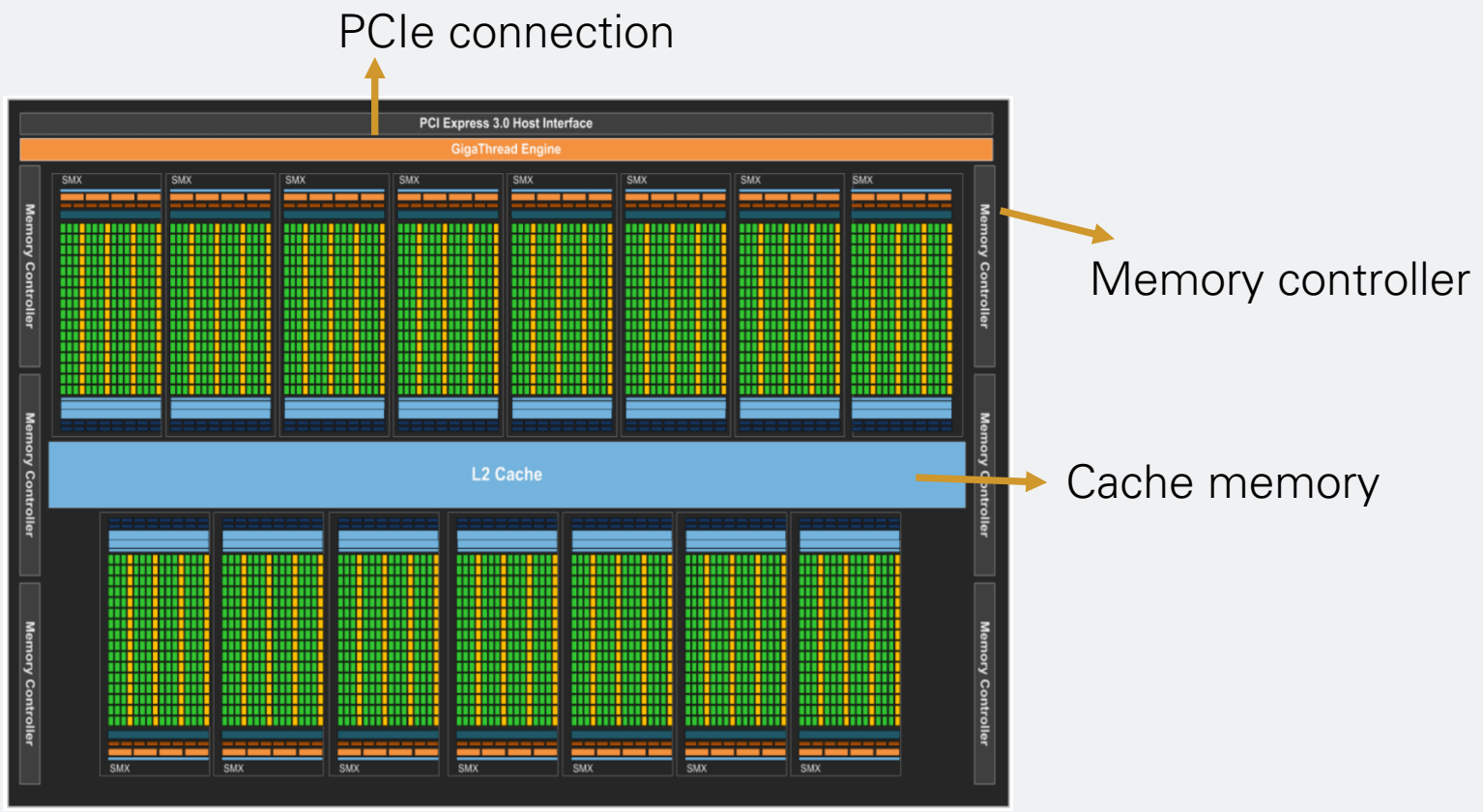


- **SIMT**: Single Instruction Multiple Threads → GPUs
  - Each thread performs the same instruction but on different data
  - Synchronization steps are needed

GPU COMPOSITION: HARDWARE

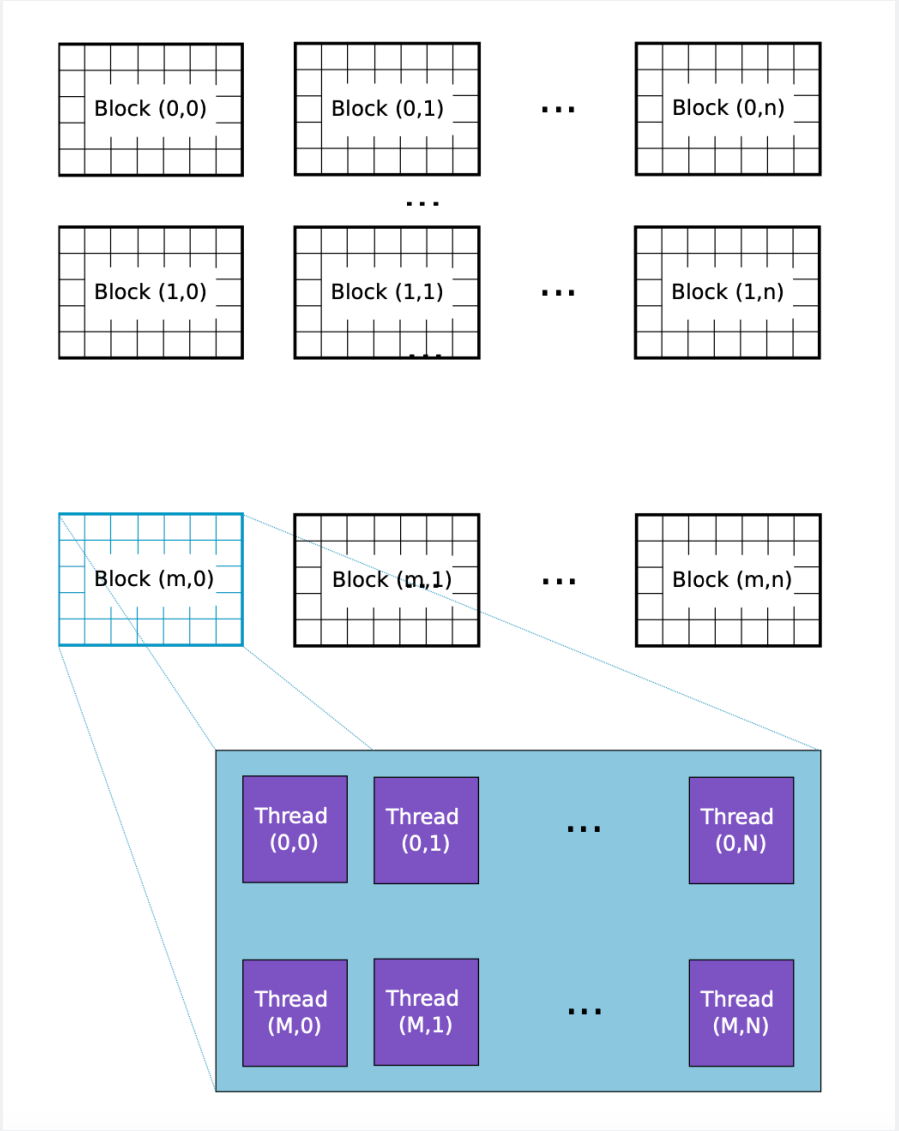
- Processors (= SIMT cores) are organised in streaming multiprocessors (SM) which compose the GPU
- How does the parallelization work? How is it assigned in the hardware?
- What kind of memory we have available?





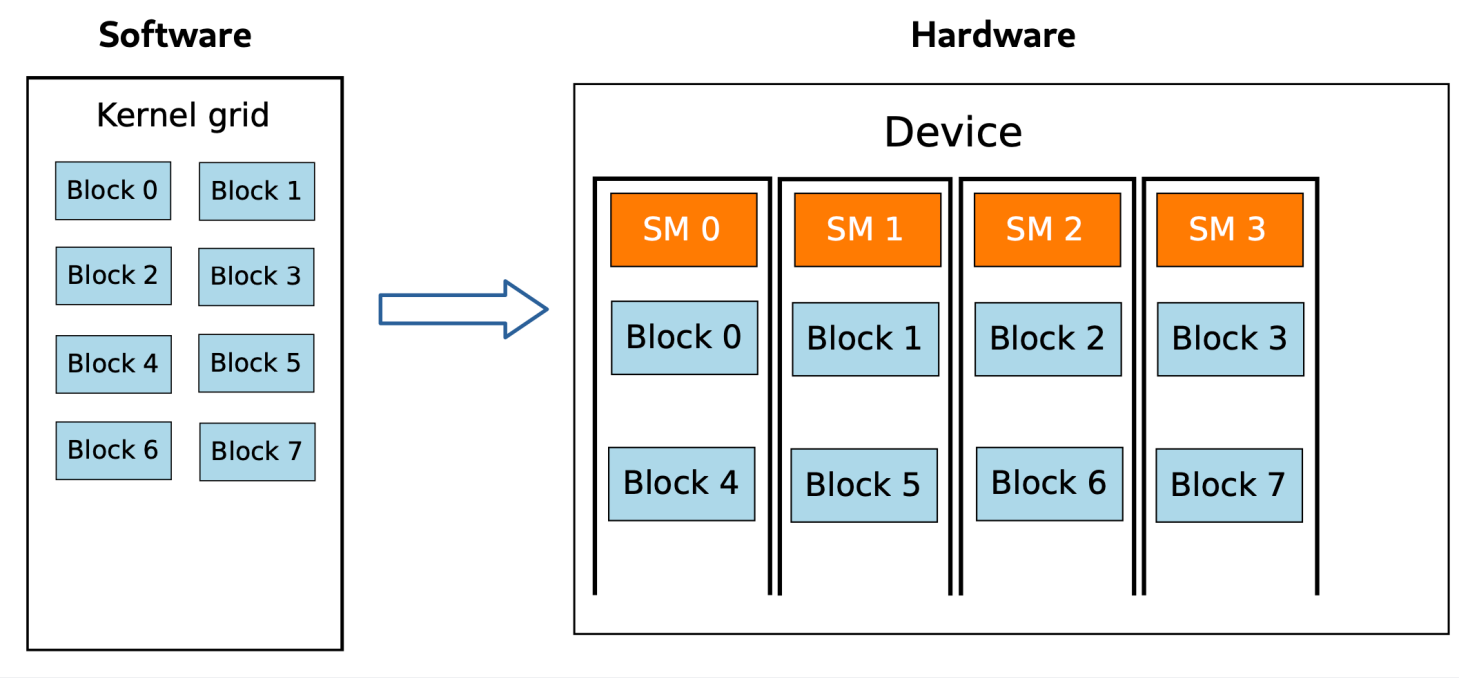
PARALLELIZATION

- The GPU code (**kernel**) is executed in many **threads**
- The total number of threads are split into **blocks** (fixed set of threads, generally maximum of 1024)
- Each thread processes the same instruction, the kernel, each one on different data
- We can go up to 3 dimensions both in blocks and threads



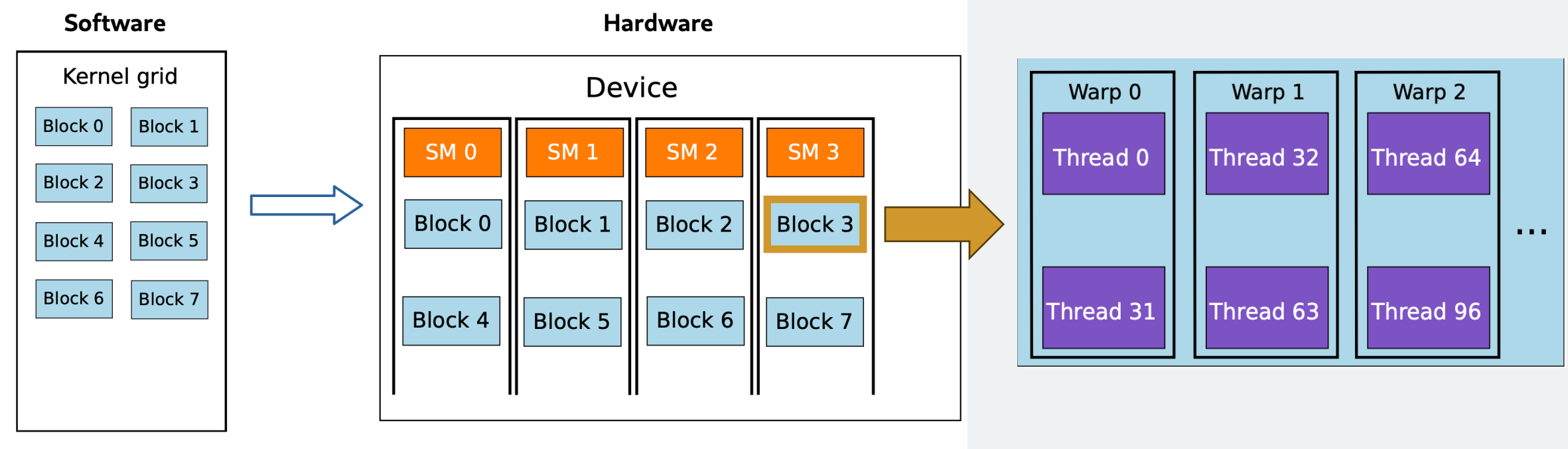
ASSIGNMENT TO STREAMING MULTIPROCESSORS

- Once the **kernel** is defined, the processes are divided into blocks a scheduled to the streaming multiprocessors (SM) of the GPU according to resource usage (memory, registers, …)
- The execution of the blocks is arbitrary



WARPS ASSIGNMENT

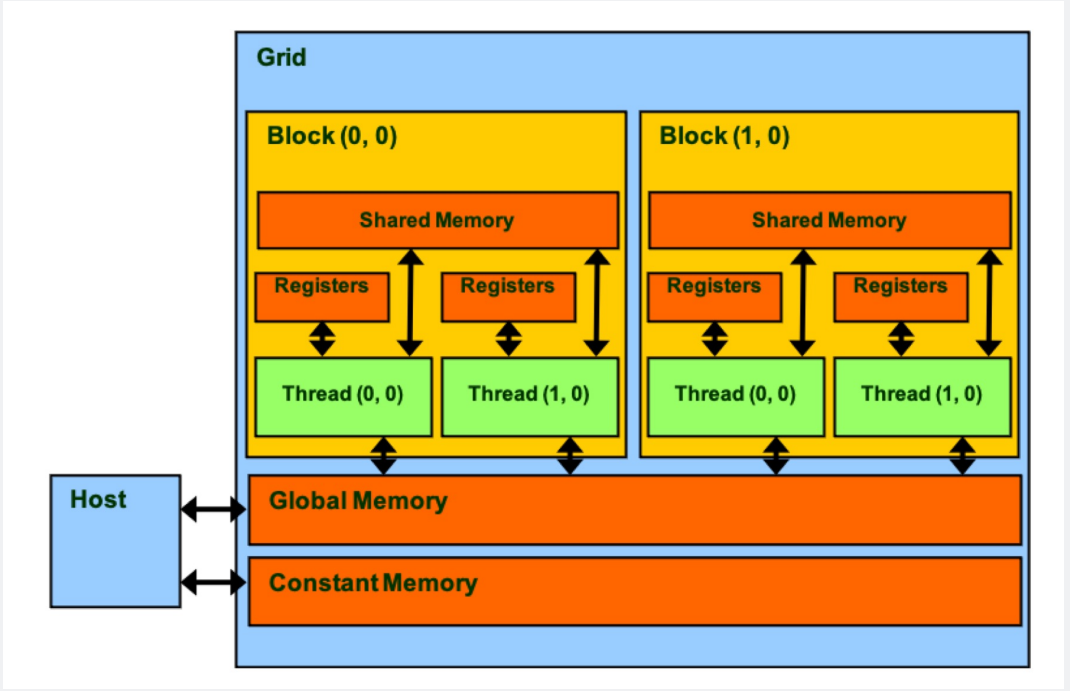
- Within a block, threads are processed in **warps** (= 32 threads in Nvidia GPUs)
- Warps are the smallest entity, meaning block size should be chosen as multiple of 32 (or warp_size)
- This ensures no threads are inherently in idle state



MEMORY LAYOUT AND USAGE

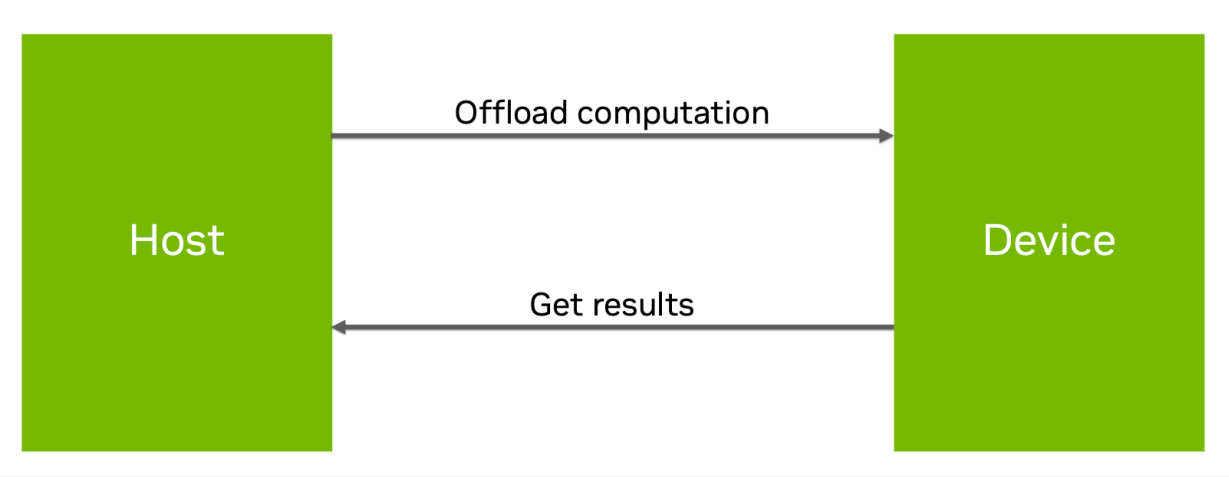
- The GPU has 3 main kind of memory:

1. **Global memory**: high latency, GBs of space
   - Main memory
   - Communication with the CPU (host)
2. **Caches**: lower latency, KBs of space
   - **Shared memory**: allows communication among threads in one block
   - **Constant memory**: read-only memory (only write from host), used to store constants
3. **Registers**: lowest latency configurable (usually 255 registers per thread)
   - Accessible only from single thread
   - All variables defined are stored in registers
   - If exceded, can result in performance penalty



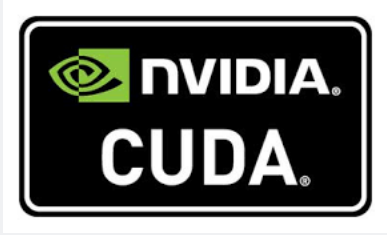
CPU-GPU COMMUNICATION

- The CPU is required to launch the applications as the **host**
- The host offloads some of the work in the GPU as the **device**
- The host takes care of the application at all times (stopping, pausing, …)
- All input data start from the **host** which populates the **global memory** and all data must return to the **host** when the process is finished
- Once in the global memory, data can be stored in constant, shared memory or registers based on the need (all their contents are flushed once the kernel function terminates)



GPU PROGRAMMING ENVIRONMENTS

- NVIDIA programming interface: CUDA
  - Works only with NVIDIA GPUs
  - Well documented, many tutorial



- AMD ROCm (HIP): open source platform
  - Support AMD and NVIDIA GPUs
  - Newer development, less documentation

- OpenCL: framework for heteorgeneous platforms
  - CPU, GPU, FPGA, …

- SYCL: C++ heterogeneous platform based on OpenCL
  - Intel GPUs

A FIRST CUDA KERNEL

- Let's have a first look at a simple CUDA kernel:
  - Identifier: __global__
  - Indices: blockIdx.x and threadIdx.x
  - No std::cout allowed in the device, using printf

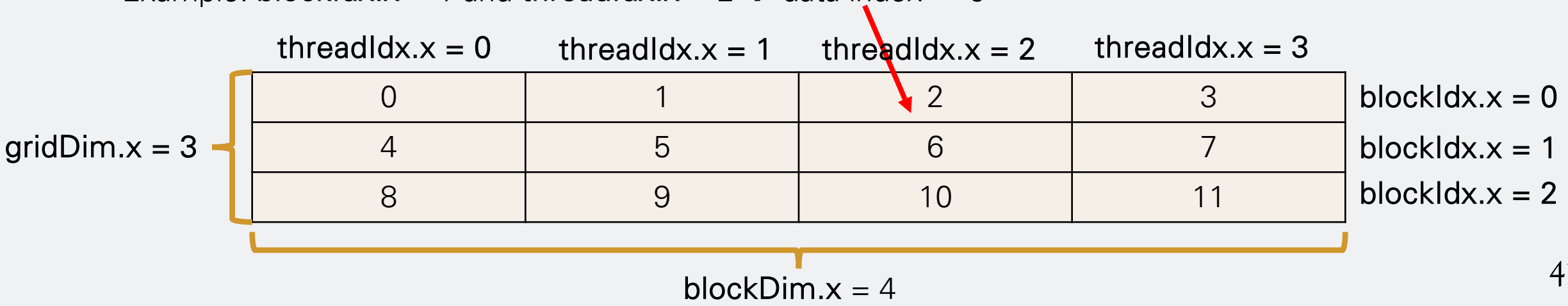
```
__global__ void hello_world_gpu() {

  /* blockIdx.x:  Accesses index of block within grid in x direction
     threadIdx.x: Accesses index of thread within block in x direction
   */
   if ( blockIdx.x < 100 && threadIdx.x < 100 )
    printf("Hello World from the GPU at block %u, thread %u \n", blockIdx.x, threadIdx.x);

}
```

FUNCTION DECLARATION

- We have different identifiers which can be used in CUDA:
  1. __global__ : function called from host and executed on device
  2. __device__: function called from device and executed on device
  3. __host__ : function called from host and executed on host

INDICES AND PARALLELIZATION

- Inside a CUDA kernel, **indices** help identify single threads
- One-dimensional example of 3 blocks with 4 threads each:
  - **gridDim.x** = 3, number of blocks in the grid
  - **blockIdx.x** identifies the current block number
  - **blockDim.x** = 4, refers to the number of threads in the block
  - **threadIdx.x** identifies the current thread number
- This means that the formula **blockIdx.x * blockDim.x + threadIdx.x** uniquely identifies one thread
- Example: **blockIdx.x** = 1 and **threadIdx.x** = 2 → data index = 6



INDICES CONFIGURATIONS

- Grid and block sizes can be defined up to 3 dimensions, which can help in the parallelization process
- This mean we can have x, y, z:
  - gridDim.x, gridDim.y, gridDim.z
  - blockIdx.x, blockIdx.y , blockIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - threadIdx.x, threadIdx.y, threadIdx.z
- The maximum number of threads per block is 1024, which cannot be exceded (multiplication of the 3 dimensions)
- The maximum number of blocks varies per hardware, usually 65535 per dimension

RUNNING A FIRST CUDA KERNEL

- We need to define **grid size** and **block size**
- **dim3** is a CUDA specific variable taking up to 3 input variables defining the sizes in 3 dimensions
- **cudaDeviceSynchronize()** waits for all requested tasks on device to be finished (here waiting for the printf to print out values)

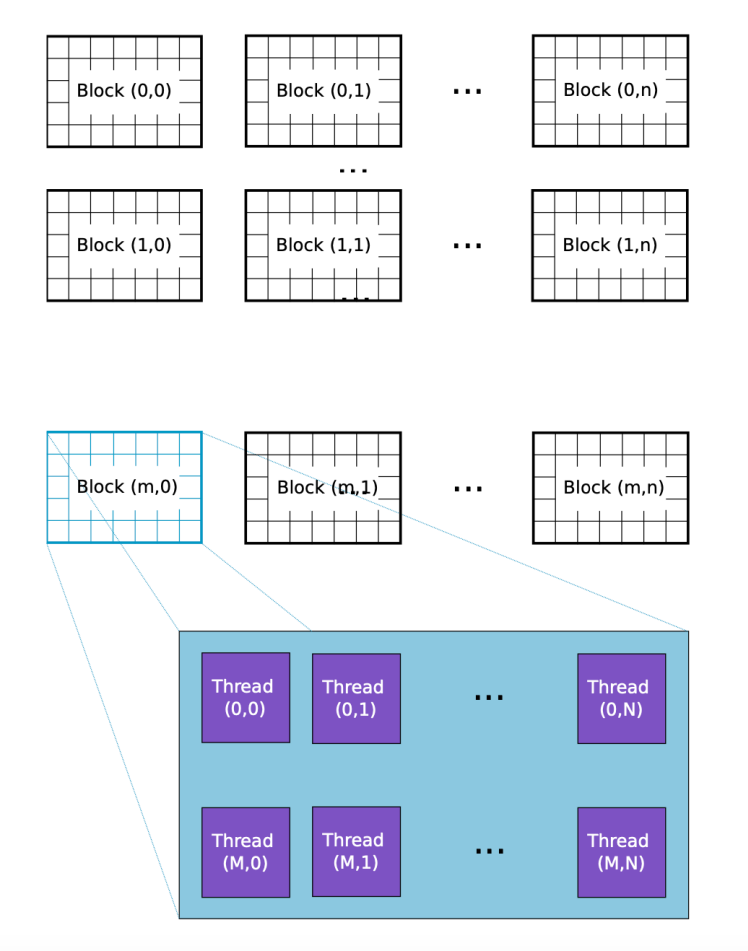
```
dim3 grid_dim(n_blocks);
dim3 block_dim(n_threads);

hello_world_gpu<<<grid_dim, block_dim>>>();

cudaDeviceSynchronize();
```

Any parameters to be passed in the kernel can be added in ()

SYNCHRONIZATION

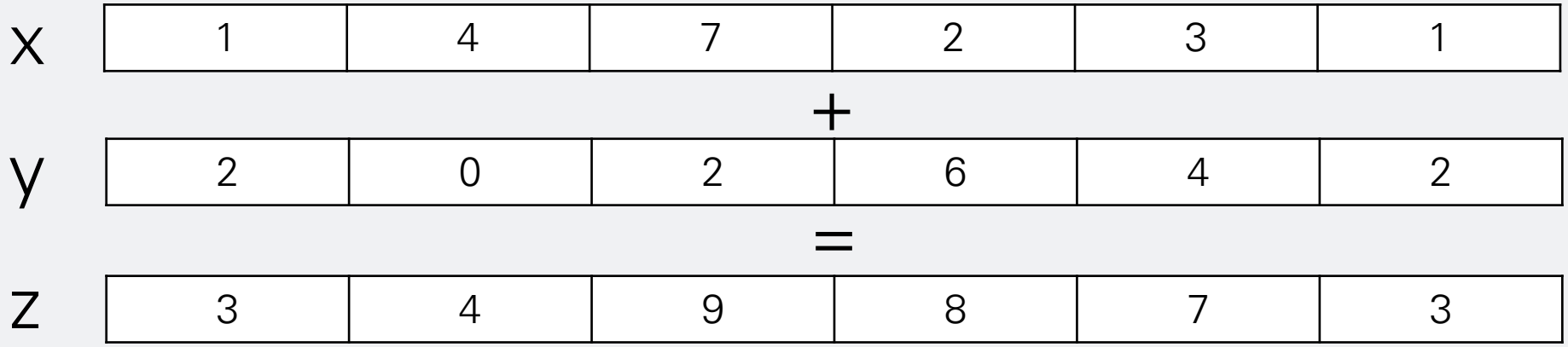
- The execution of **blocks** and **threads** is arbitrary
- If we want to ensure all work has finished, synchronization is needed
- It can be done at grid- and block-level:
  - **cudaDeviceSynchronize()** waits for all work on the device to be finished, meaning all blocks and also memory copies
  - **__syncthreads()** waits for all threads inside one block to finish their work, can be written within the kernel code (for example when having 2 consecutive loops)

```
for (int i = threadIdx.x; i < N+1; i++) {
     variable[threadIdx.x] = ...
}

__syncthreads();

for (int i = threadIdx.x; i < N+1; i++) {
     Use variable[threadIdx.x]
}
```



FOR LOOP AND PARALLELIZATION: EXAMPLE

- Example: writing an vector addition kernel, with **x**, **y** and **z** of size **N**



- How would you do it in CPU?

```
void vector_addition_cpu(int *X, int *Y, int *Z, int N) {
    for (int i = 0; i < N; i++) {
        Z[i] = X[i] + Y[i];
    }
}
```

FOR LOOP AND PARALLELIZATION: EXAMPLE

- Running on a single block using size N as number of threads
- Making the loop in this way forces the block dimension to be N, otherwise we lead to incorrect results or out bounds accesses→ how to avoid this?

```
__global__ void vector_addition_gpu(int *X, int *Y, int *Z) {
    const int i = threadIdx.x;
    Z[i] = X[i] + Y[i];
}

int main(){
    ...
    vector_addition_gpu<<<1, N>>>( X, Y, Z);
    ...
}
```

Index value different for each of the N threads in the block

Grid dimension 1 and block dimension N

FOR LOOP AND PARALLELIZATION: EXAMPLE

- Making a **block-dimension strided loop:** block-dimension can be any number **n**
  - **Stride = blockDim.x = n**
  - If n >= N: the loop will be the same as previous slide
  - If n < N: some or all threads will make more than 1 iteration

```
__global__ void vector_addition_gpu(int *X, int *Y, int *Z, int N) {
    for (int i = threadIdx.x; i < N; i += blockDim.x) {
    Z[i] = X[i] + Y[i];
  }
}

int main(){
    ...
    vector_addition_gpu<<<1, n>>>( X, Y, Z, N);
    ...
}
```

Block-dimension strided

Grid dimension 1 and block dimension n

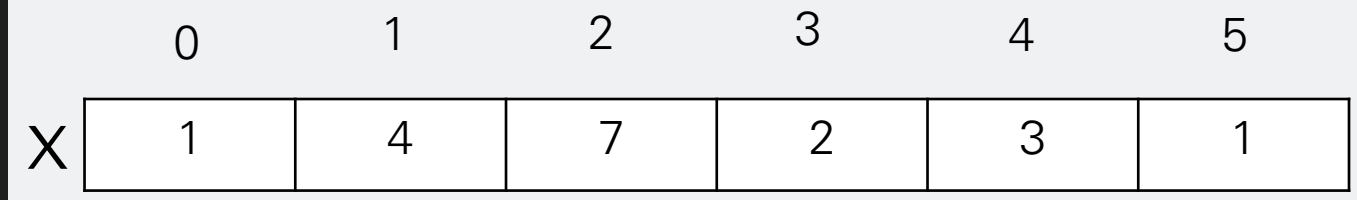
FOR LOOP AND PARALLELIZATION: EXAMPLE

- Making a **block-dimension strided loop:** block-dimension can be any number **n**

```
    for (int i = threadIdx.x; i < N; i += blockDim.x) {
    Z[i] = X[i] + Y[i];
  }
```

- Example: N = 6
  - If n = 8, first 6 go in each thread, last 2 do not satisfy i < N in loop
  - If n = blockDim.x = 4, first 4 go in first iteration of the loop (4 threads), then 2 calculation go in second iteration i+= blockDim.x

```
__global__ void vector_addition_gpu(int *X, int *Y, int *Z, int N) {
    for (int i = threadIdx.x; i < N; i += blockDim.x) {
    Z[i] = X[i] + Y[i];
  }
}

int main(){
    ...
    vector_addition_gpu<<<1, n>>>( X, Y, Z, N);
    ...
}
```



FOR LOOP AND PARALLELIZATION: EXAMPLE

- Making a **grid- and block-dimension strided loop:** with grid-dimension **m** and block-dimension **n**
  - Stride = blockDim.x * gridDim.x
  - Start = threadIdx.x + blockIdx.x * blockDim.x
  - Iterating in this way in all threads across all blocks, profiting from maximum parallelization

```
__global__ void vector_addition_gpu(int *X, int *Y, int *Z, int N) {
    for (int i = threadIdx.x + blockIdx.x * blockDim.x; i < N; i += blockDim.x * gridDim.x) {
    Z[i] = X[i] + Y[i];
  }
}

int main(){
    ...
    vector_addition_gpu<<<m, n>>>( X, Y, Z, N);
    ...
}
```

Grid dimension m and block dimension n

GLOBAL MEMORY MANAGEMENT

- Vector addition example, how to handle memory for our vectors from host to device?

```
int main(){
    int *x_d, *y_d, *z_d;
    cudaMalloc( (void**)&x_d, size * sizeof(int) );
    cudaMalloc( (void**)&y_d, size * sizeof(int) );
    cudaMalloc( (void**)&x_d, size * sizeof(int) );

    cudaMemcpy( x_d, X, size * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( y_d, Y, size * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( z_d, Z, size * sizeof(int), cudaMemcpyHostToDevice );

    vector_addition_gpu<<<m, n>>>( X, Y, Z, N);

    cudaDeviceSynchronize();

    cudaMemcpy( Z, z_d, size * sizeof(int), cudaMemcpyDeviceToHost );

    cudaFree( x_d );
    cudaFree( y_d );
    cudaFree( z_d );
}
```

1. Allocate global memory in device of size N with **cudaMalloc**
2. Populate global memory with data using **cudaMemcpy: cudaMemcpyHostToDevice**
3. Run kernel
4. Synchronize after kernel completion
5. Read inputs back in host: **cudaMemcpyDeviceToHost**
6. Pointer to global memory to be freed with **cudaFree**

A WORD ON SHARED MEMORY

- It is useful to define allocate shared memory from the kernel for much faster data usage, knowing its limited size
- If its size is known in advance, it is better to allocate the correct size:
  - `__shared__ float variable_sh[N];`
- Otherwise shared memory could also be allocated dynamically, but size must be known in the host by passing an additional argument in the kernel call
  - `__shared__ float variable_sh[];`
  - `kernel_name<<m,n, N * sizeof(float)>>();`

ATOMIC OPERATIONS AND RACE CONDITIONS

- We need to be cautious when modifying a value in memory and reading it again in different threads:
  - Timing of threads can be different
  - Three main operations: read, modify and write
- Use atomic operations:
  - Makes the read-write-modify as a single operation = cannot be interrupted
  - atomicAdd(), atomicSub(), …
  - Usual use-cases: counting elements, searching elements in array, histogramming, …

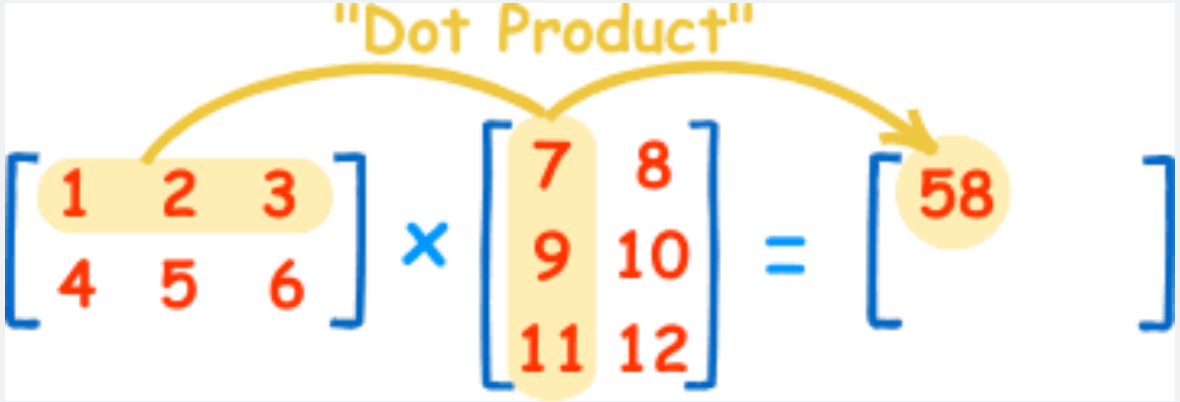
SATURATING GPU: DEBUGGING

- We need to avoid saturating **shared memory** (O(KBs) and **registers** (255 per thread max) which can cause loss of performance
- Several ways to debug and profile (dynamic program analysis) GPU code:
  - **nvprof**: profiler built-in CUDA
  - **cuda-gdb**: command line debugger based of gdb
  - **Nvidia nsight (ncu):** debugger and profiler implemented in VS Code and usable from command line
  - **nsys**: command line profiler which produce analytics

```
seed_xz::seed_xz(seed_xz::Parameters, const char *) (500, 1, 1)x(128, 1, 1), Context 1, Stream 13, Device 0, CC 8.6
  Section: GPU Speed Of Light Throughput
  ----------------------- ------------- ------------
  Metric Name               Metric Unit Metric Value
  ----------------------- ------------- ------------
  DRAM Frequency          cycle/nsecond         7.58
  SM Frequency            cycle/nsecond         1.17
  Elapsed Cycles                  cycle    4,092,774
  Memory Throughput                   %         9.86
  DRAM Throughput                     %         1.40
  Duration                       msecond         3.50
  L1/TEX Cache Throughput             %        26.88
  L2 Cache Throughput                 %         6.06
  SM Active Cycles                cycle 1,501,520.95
  Compute (SM) Throughput             %         9.86
  ----------------------- ------------- ------------

  WRN   This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance
        of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate
        latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

  Section: Launch Statistics
  -------------------------------- --------------- ---------------
  Metric Name                          Metric Unit    Metric Value
  -------------------------------- --------------- ---------------
  Block Size                                                   128
  Function Cache Configuration                      CachePreferNone
  Grid Size                                                    500
  Registers Per Thread             register/thread             118
  Shared Memory Configuration Size           Kbyte           65.54
  Driver Shared Memory Per Block        Kbyte/block            1.02
  Dynamic Shared Memory Per Block       byte/block               0
  Static Shared Memory Per Block        Kbyte/block            7.22
  Threads                                    thread          64,000
  Waves Per SM                                                1.95
  -------------------------------- --------------- ---------------

  WRN   A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the
        target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical
        occupancy of the kernel. This kernel launch results in 1 full waves and a partial wave of 244 thread blocks.
        Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for
        up to 50.0% of the total kernel runtime with a lower occupancy of 36.3%. Try launching a grid with no
        partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for
        a grid. See the Hardware Model
        (https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-hw-model) description for more
        details on launch configurations.

  Section: Occupancy
  ------------------------------- ----------- ------------
  Metric Name                     Metric Unit Metric Value
  ------------------------------- ----------- ------------
  Block Limit SM                        block           16
  Block Limit Registers                 block            4
  Block Limit Shared Mem                block            7
  Block Limit Warps                     block           12
  Theoretical Active Warps per SM        warp           16
  Theoretical Occupancy                     %        33.33
  Achieved Occupancy                        %        21.23
  Achieved Active Warps Per SM           warp        10.19
  ------------------------------- ----------- ------------

  WRN   This kernel's theoretical occupancy (33.3%) is limited by the number of required registers. The difference
        between calculated theoretical (33.3%) and measured achieved occupancy (21.2%) can be the result of warp
        scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between
        warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide
        (https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy) for more details on
        optimizing occupancy.
```

OTHER EXAMPLE: MATRIX MULTIPLICATION

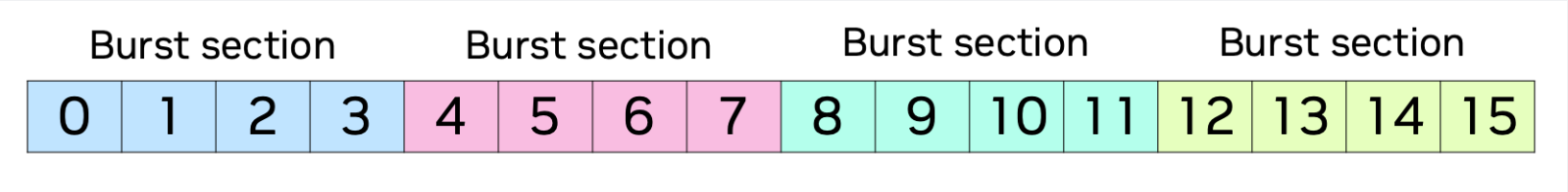
- We quickly introduce tiling and coalasced memory accesses



TILING

- Tiled data processing: dividing large datasets into many tiles which are processed at one time
- It is useful when data have similar patterns, such that threads can access memory in a tiled way
- Typical tiled-process:
  1. Load tile from global to shared memory
  2. Synchronize
  3. Multiple threads access the data in shared memory
  4. Synchronize
  5. Move to the next tile
- Example: multiply two arrays of any given size by dividing it into tiles

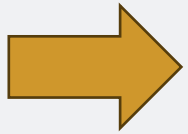
COALESCED MEMORY ACCESS



- Global memory is organized into **bursts sections**, each cell representing a byte
- If threads make a memory request under the same **burst section,** the access is **coalesced**
- Non-coalesced memory access can significantly affect performance
- **Advice:**
  - Access index to an array X should have a part depedent and indendent of threadIdx.x
  - Example: X[**x0 + threadIdx.x**] with x0 independent of thread index

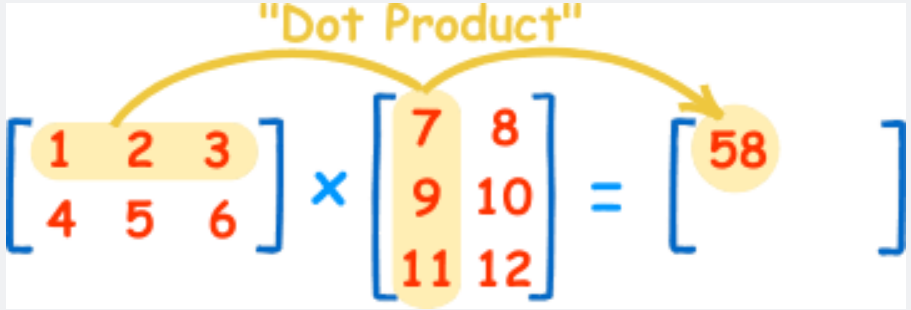
OTHER EXAMPLE: MATRIX MULTIPLICATION

X00	X01	X02
X10	X11	X12
X20	X21	X22



X00	X01	X02	X10	X11	X12	X20	X21	X22

- Advice: always store higher order arrays into 1D array
- Matrix multiplication becomes a 1D array multiplication:
  - X of size m x n
  - Y of size n x k
  - Result is Z of size m x k



OTHER EXAMPLE: MATRIX MULTIPLICATION

- Matrix multiplication becomes a 1D array multiplication:
  - X of size m x n
  - Y of size n x k
  - Result is Z of size m x k

X00	X01	X02	X10	X11	X12	X20	X21	X22

Y00	Y01	Y02	Y10	Y11	Y12	Y20	Y21	Y22

- With this method access to Y are coalesced while X accesses are not
- This can be improved using shared memory

```
__global__ void multiply_matrices(const float *X, const float *Y, float *Z, int m, int n, int k) {
  for (int row = threadIdx.x; row < n; row += blockDim.x) {
    for (int col = threadIdx.y; col < k; col += blockDim.y) {
      float element = 0.f;
      for (int i = 0; i < n; i++) {
        element += X[row * m + i] * Y[i * k + col];
      }
      Z[row * size + col] = element;
    }
  }
}
```

OTHER EXAMPLE: MATRIX MULTIPLICATION

- This can be improved using shared memory, by preloading all elements of X and Y
- Only using coalesced accesses as all threads can access shared memory
- Example on how to load shared memory and using function of the previous slide to do 16 x 16 matrix multiplication

```
__global__ void shared_matrix_multiply_16_16(float* X, float* Y, float* Z) {
  __shared__ float shared_X [256];
  __shared__ float shared_Y [256];
  // Coalesced loads
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
    shared_X[i] = X[i];
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
    shared_Y[i] = Y[i];
  __syncthreads();
  // Now shared_X and shared_Y are populated and can be used
  // instead of the original arrays to perform the multiplication
  multiply_arrays(shared_X , shared_Y , Z, 16, 16, 16);
}
```

STREAM AND PIPELINES

- A **stream** is a sequence of commands to be executed in order, example, kernel invocations, memory transmissions and allocations, synchronizations, …
- Any instructions run in a stream must complete before the next instruction is issued
- CUDA uses a default stream
- Non-default stream can be defined, but the default one will always have priority
- GPUs can actually perform in this way data transmissions while executing kernels = **pipeline**
- Typical pipeline with 3 streams:
  - Use SMs to perform computations
  - Transfer data from host to device
  - Transfer data from device to host (using **cudaMemcpyAsync** to transfer data asynchrounsly in a non-default stream)

SUMMARY

- Starting from year ~2000, GPUs were starting to be used for general purpose
- It is important to understand how and how much of our algorithm can be parallelized
- GPUs are single instruction multiple threads (SIMT) devices
- Parallelization organised in blocks and threads
- The CPU-GPU communication is fundamental as the CPU is in charge of all time of the process and can offload some work to the GPU
- It is important to know and understand the memory layout of a GPU to achieve the best performance and use them in the best way

RESOURCES USED IN THIS TALK

- Most of what I learned comes from Dorothea vom Bruch (physics researcher at CPPM in Marseille) and Daniel Campora (now Nvidia engineer)
- They do also amazing lectures, from which I took all the inspiration: tCSC2023 and tSCS2024
- Where to find material:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide
  - D. B. Kirk, W. w. Hwu: "Programming Massively Parallel Processors"
  - J. Sanders, E. Kandrot: "CUDA by Example: An Introduction to General-Purpose GPU Programming"
  - N. Wilt: "The CUDA Handbook: A Comprehensive Guide to GPU Programming"