Technische Universiteit Eindhoven University of Technology

GPU Computing: Development and Analysis

Part 1

Anton Wijs
Muhammad Osama

Marieke Huisman
Sebastiaan Joosten

Where innovation starts

NLeSC GPU Course

Rob van Nieuwpoort & Ben van Werkhoven

netherlands eScience center

by SURF & NWO

Who are we?

- Anton Wijs
  - Assistant professor, Software Engineering & Technology, TU Eindhoven
  - Developing and integrating formal methods for model driven software engineering
    - Verification of model transformations
    - Automatic generation of (correct) parallel software
  - Accelerating formal methods with multi-/many-threading

- Muhammad Osama
  - PhD student, Software Engineering & Technology, TU Eindhoven
  - GEARS: GPU Enabled Accelerated Reasoning about System designs
    - GPU Accelerated SAT solving

Schedule GPU Computing

- Tuesday 12 June
  - Afternoon: Intro to GPU computing
- Wednesday 13 June
  - Morning / Afternoon: Formal verification of GPU software
  - Afternoon: Optimised GPU computing (to perform model checking)

Schedule of this afternoon

- 13:30 – 14:00 Introduction to GPU Computing
- 14:00 – 14:30 High-level intro to CUDA Programming Model
- 14:30 – 15:00 1st Hands-on Session

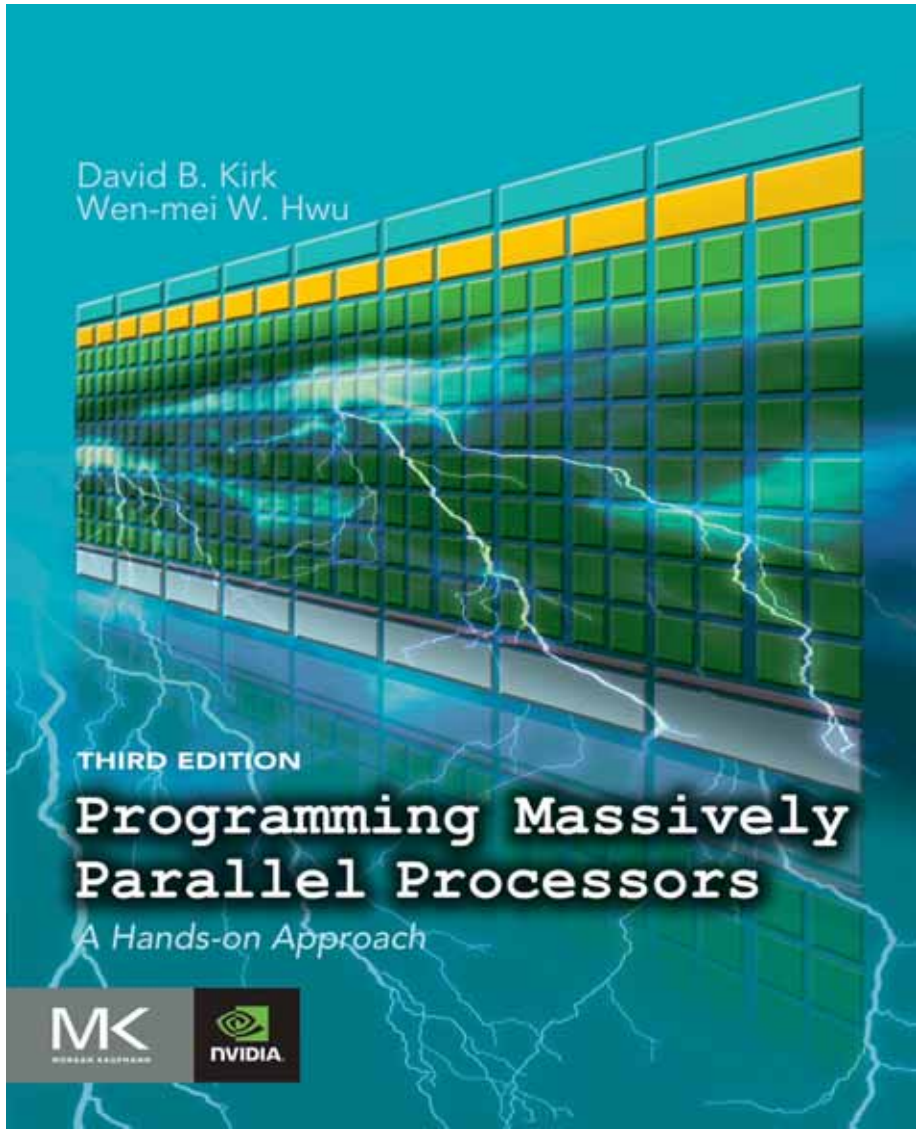
- 15:00 – 15:15 Coffee break

- 15:15 – 15:30 Solution to first Hands-on Session
- 15:30 – 16:15 CUDA Programming model Part 2 with 2nd Hands-on Session
- 16:15 – 16:40 CUDA Program execution

Before we start

- You can already do the following:
  - Install VirtualBox (virtualbox.org)

  - Download VM file:
    - scp gpuser@131.155.68.95:GPUtutorial.ova .
      - in terminal (Linux/Mac) or with WinSCP (Windows)
      - Password: cuda2018
    - https://tinyurl.com/y9j5pcwt (10 GB)
    - Or copy from USB stick



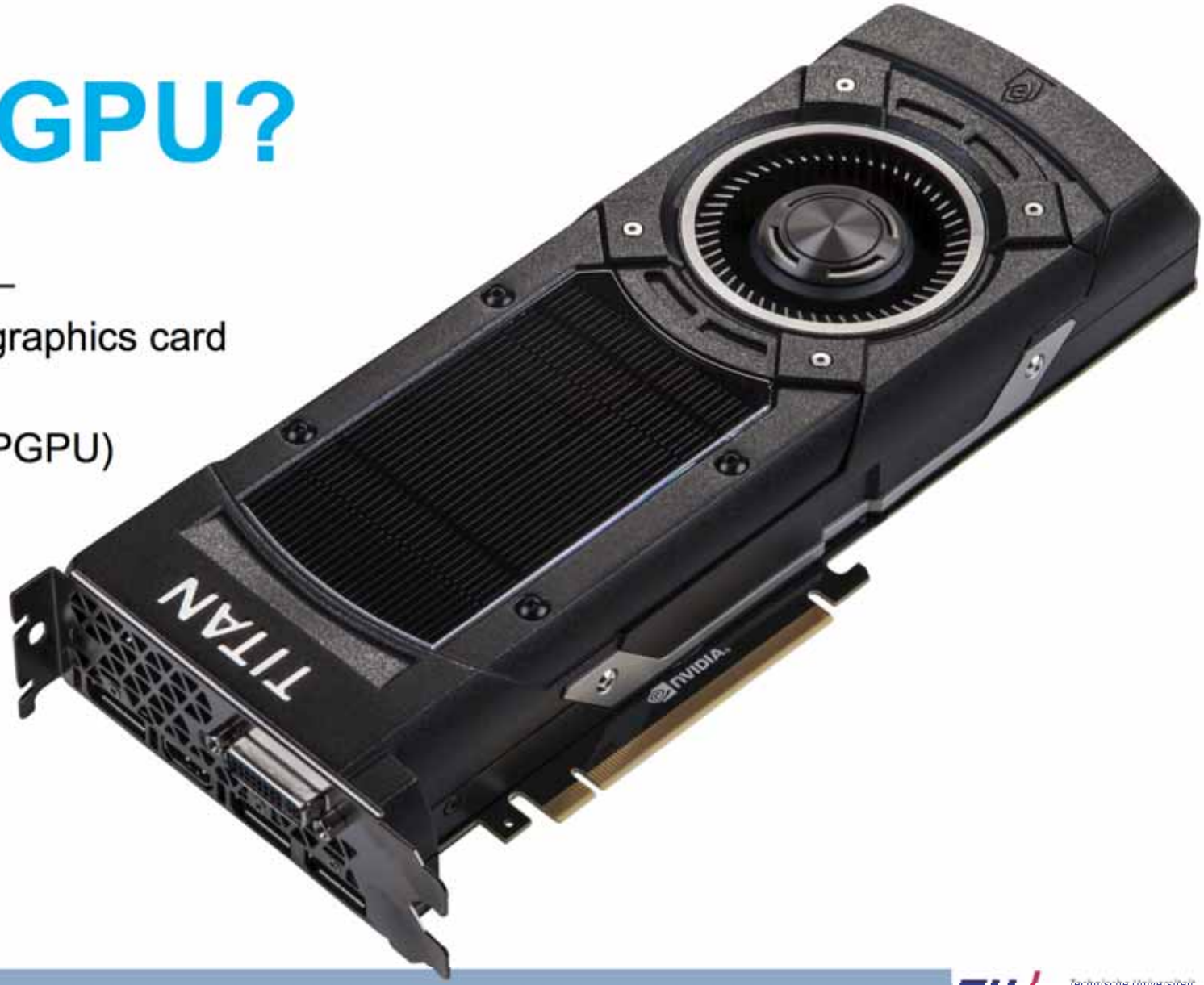
We will cover approx. first five chapters

Introduction to GPU Computing

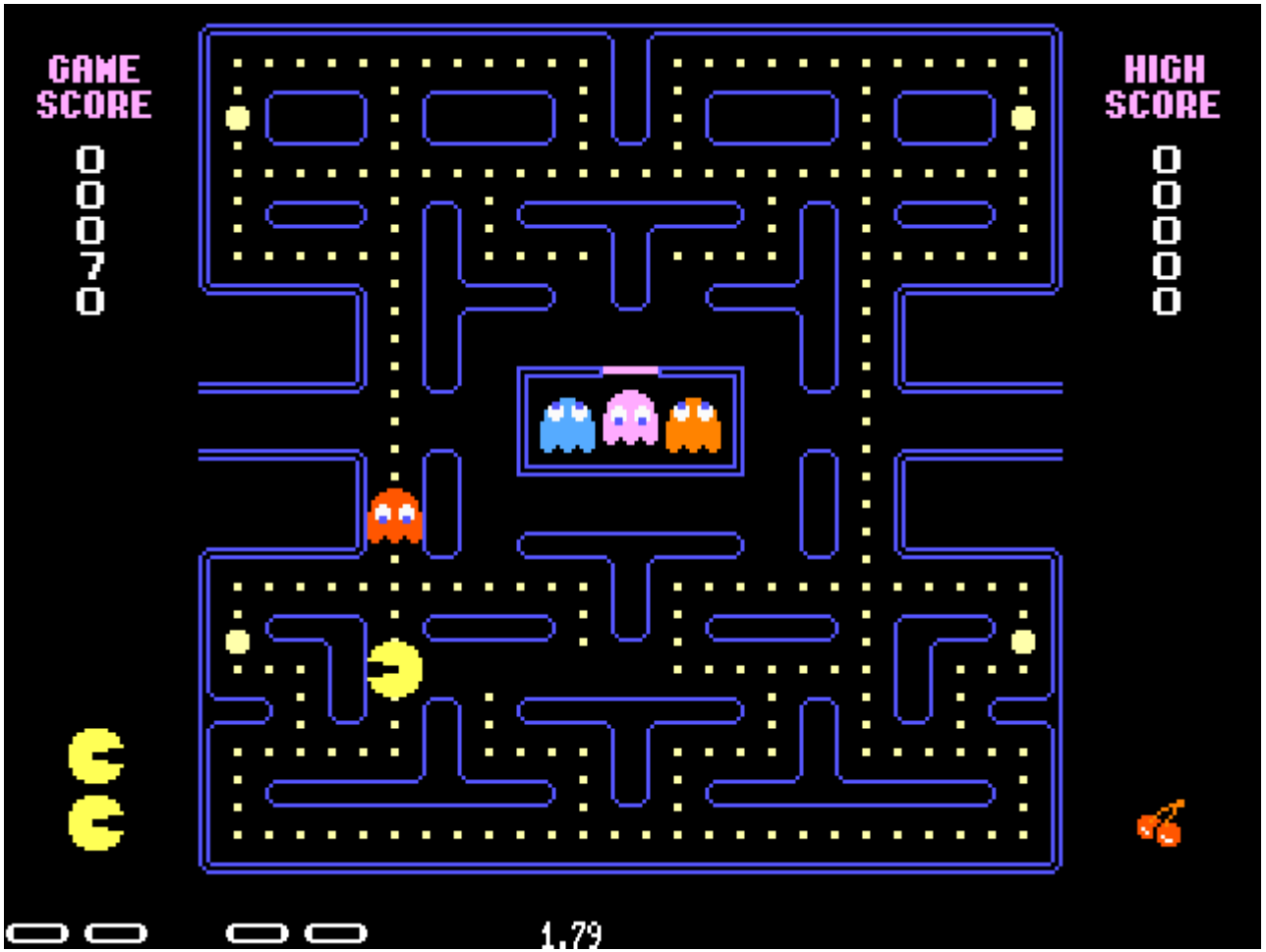
What is a GPU?

- Graphics Processing Unit –
  The computer chip on a graphics card

- *General Purpose* GPU (GPGPU)



Graphics in 1980

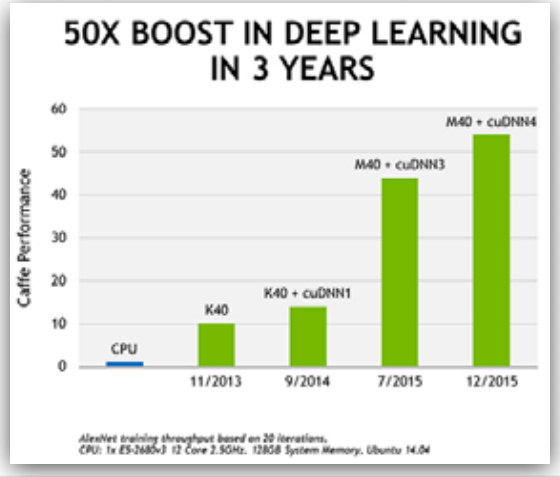


Graphics in 2000

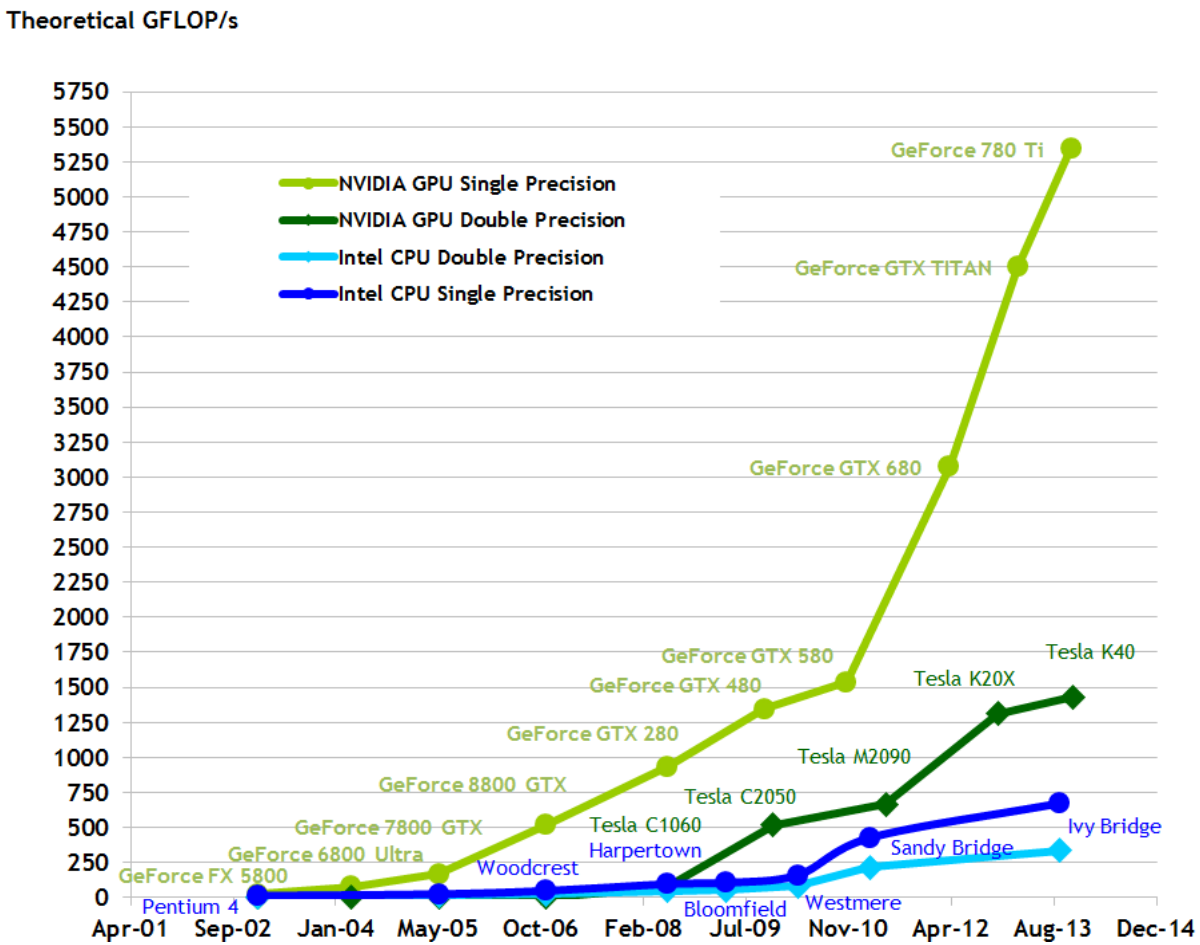
Graphics now

General Purpose Computing

- Graphics processing units (GPUs)
  - Numerical simulation, media processing, medical imaging, machine learning, …
- *Communications of the ACM 59(9):14-16 (sep.'16)*
  - "GPUs are a gateway to the future of computing"
  - Example: **deep learning**
    - 2011-12: GPUs dramatically increase performance

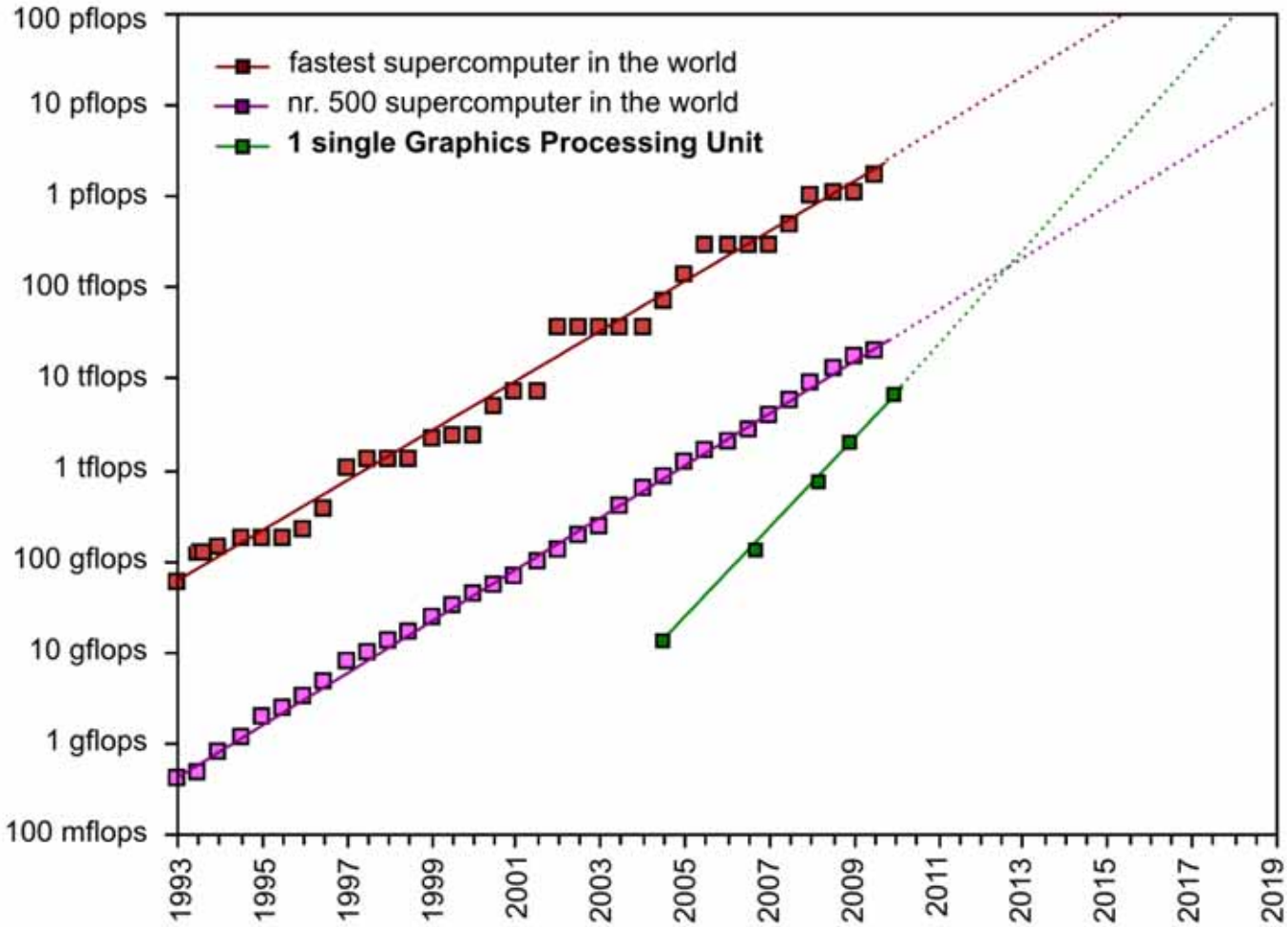


Compute performance



(According to Nvidia)

GPUs vs supercomputers ?

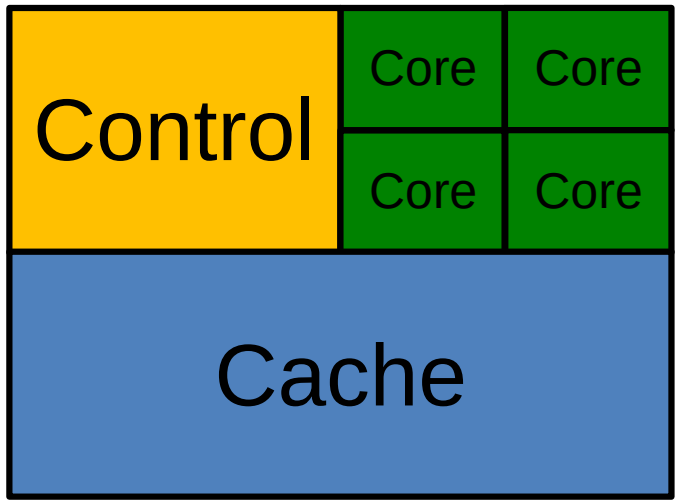


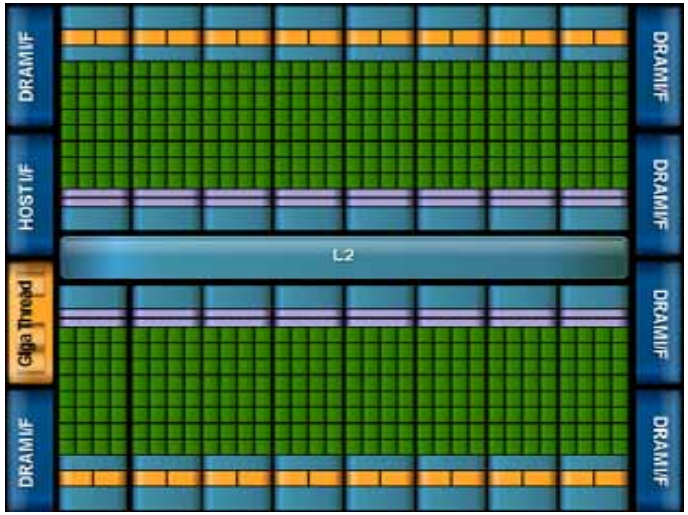
Oak Ridge's Titan

- Number 3 in top500 list: 27.113 pflops peak, 8.2 MW power
- 18.688 AMD Opteron processors x 16 cores = 299.008 cores
- 18.688 Nvidia Tesla K20X GPUs x 2688 cores = 50.233.344 cores

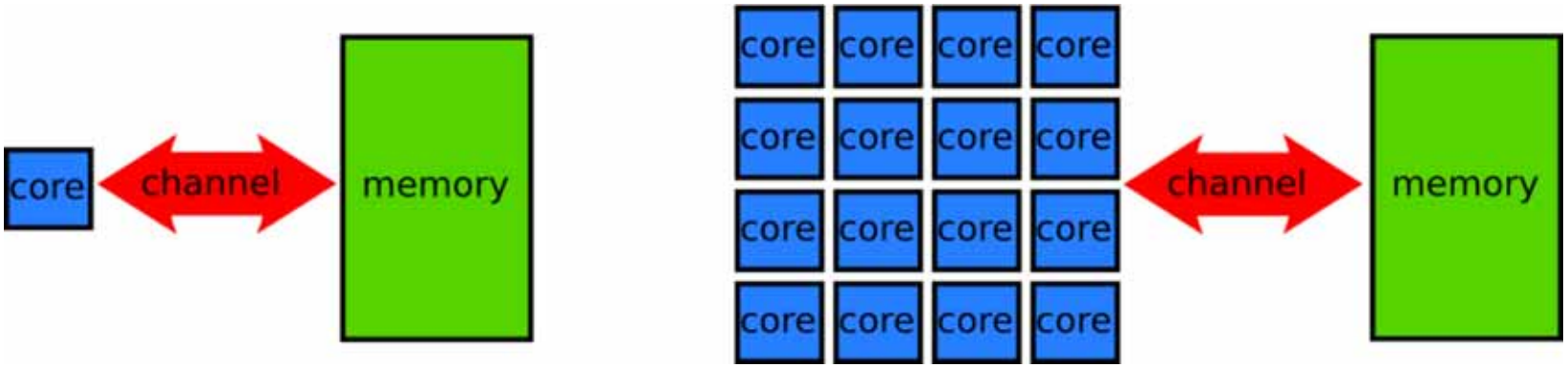
CPU vs GPU Hardware

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - Big on-chip caches
  - Sophisticated control logic
- GPU: maximize throughput of all threads
  - Multithreading can hide latency, so no big caches
  - Control logic
    - Much simpler
    - Less: share control logic across many threads





It's all about the memory

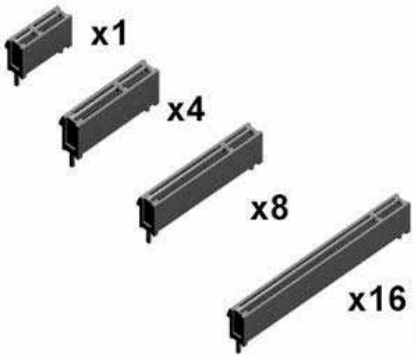


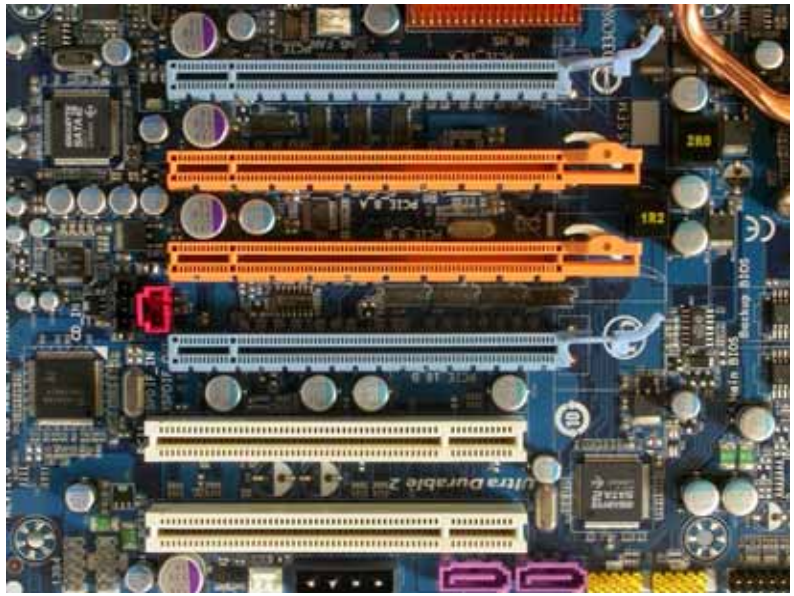
Many-core architectures

From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — largely because of issues with congestion in supplying instructions and data to the many processors.”

Integration into host system

- PCI-e 3.0 achieves about 16 GB/s
- Comparison: GPU device memory bandwidth is 320 GB/s for GTX1080





Why GPUs?

- Performance
  - Large scale parallelism
- Power Efficiency
  - Use transistors more efficiently
  - #1 in green 500 uses NVIDIA Tesla P100
- Price (GPUs)
  - Huge market
  - Mass production, economy of scale
  - Gamers pay for our HPC needs!

When to use GPU Computing?

- When:
  - Thousands or even millions of elements that can be processed in parallel

- Very efficient for algorithms that:
  - have high arithmetic intensity (lots of computations per element)
  - have regular data access patterns
  - do not have a lot of data dependencies between elements
  - do the same set of instructions for all elements

A high-level intro to the CUDA Programming Model

CUDA Programming Model

Before we start:

- I'm going to explain the CUDA Programming model
- I'll try to avoid talking about the hardware as much as possible
- For the moment, make no assumptions about the backend or how the program is executed by the hardware
- I will be using the term 'thread' a lot, this stands for 'thread of execution' and should be seen as a parallel programming concept. Do not compare them to CPU threads.

CUDA Programming Model

- The CUDA programming model separates a program into a *host* (CPU) and a *device* (GPU) part.

- The host part: allocates memory and transfers data between host and device memory, and starts GPU functions

- The device part consists of functions that will execute on the GPU, which are called *kernels*
- Kernels are executed by huge amounts of threads at the same time
- The data-parallel workload is divided among these threads
- The CUDA programming model allows you to code for each thread individually

Data management

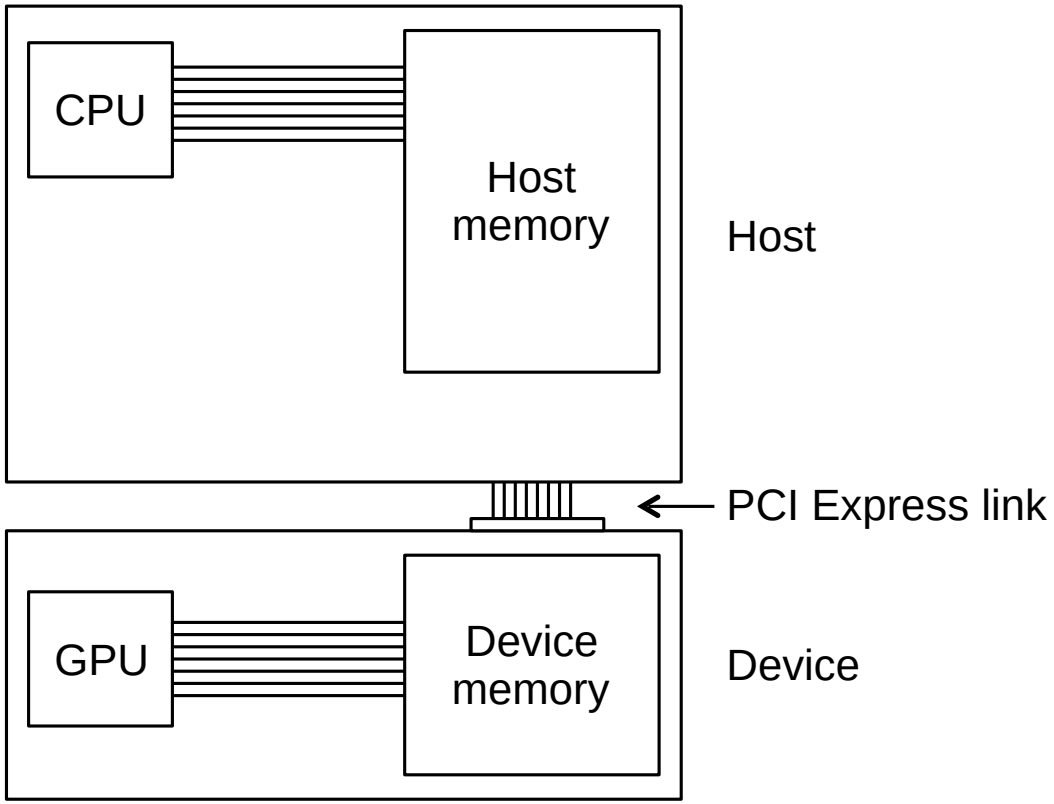
- The GPU is located on a separate device
- The host program manages the allocation and freeing of GPU memory

  C:
  - `cudaMalloc()`
  - `cudaFree()`

  Python:
  - `mem_alloc()`
- Host program also copies data between different physical memories

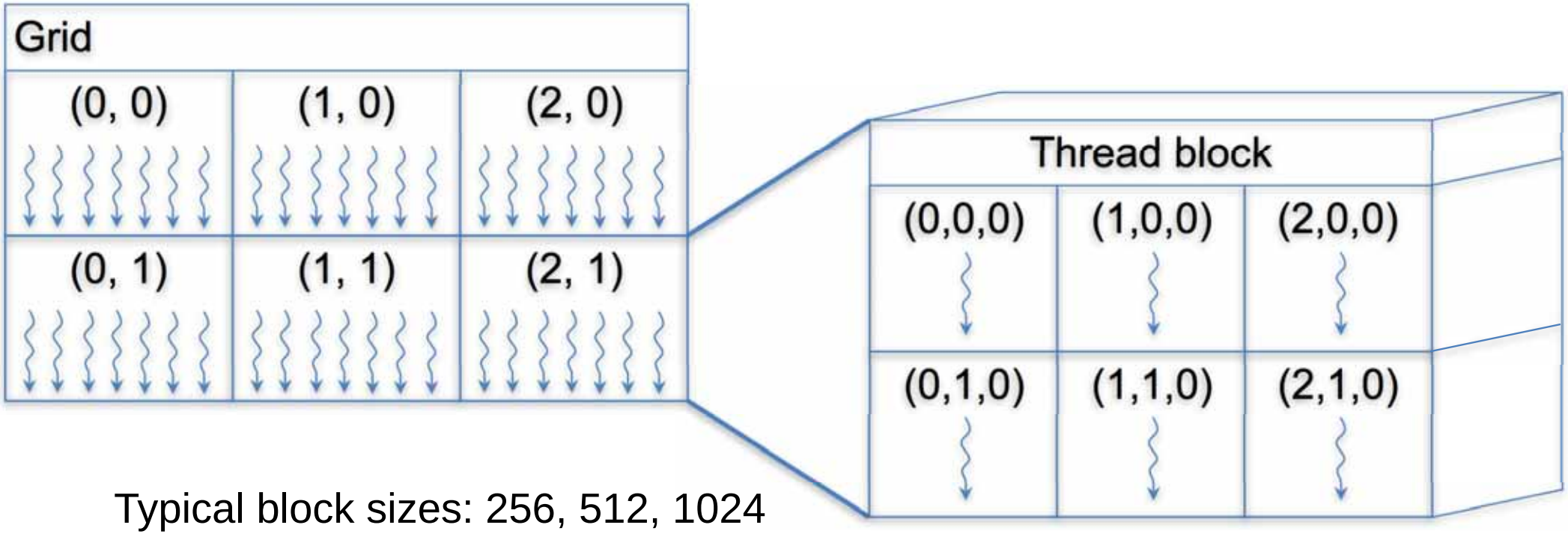
  C:
  - `cudaMemcpy()`

  Python:
  - `memcpy_htod() or memcpy_dtoh()`



Thread Hierarchy

- Kernels are executed in parallel by possibly millions of threads, so it makes sense to try to organize them in some manner



Typical block sizes: 256, 512, 1024

Threads

- In the CUDA programming model a thread is the most fine-grained entity that performs computations
- Threads direct themselves to different parts of memory using their built-in variables `threadIdx.x, y, z` (thread index *within* the thread block)
- Example:

```
for (i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

Create a single thread block of N threads:

```
i = threadIdx.x;
c[i] = a[i] + b[i];
```

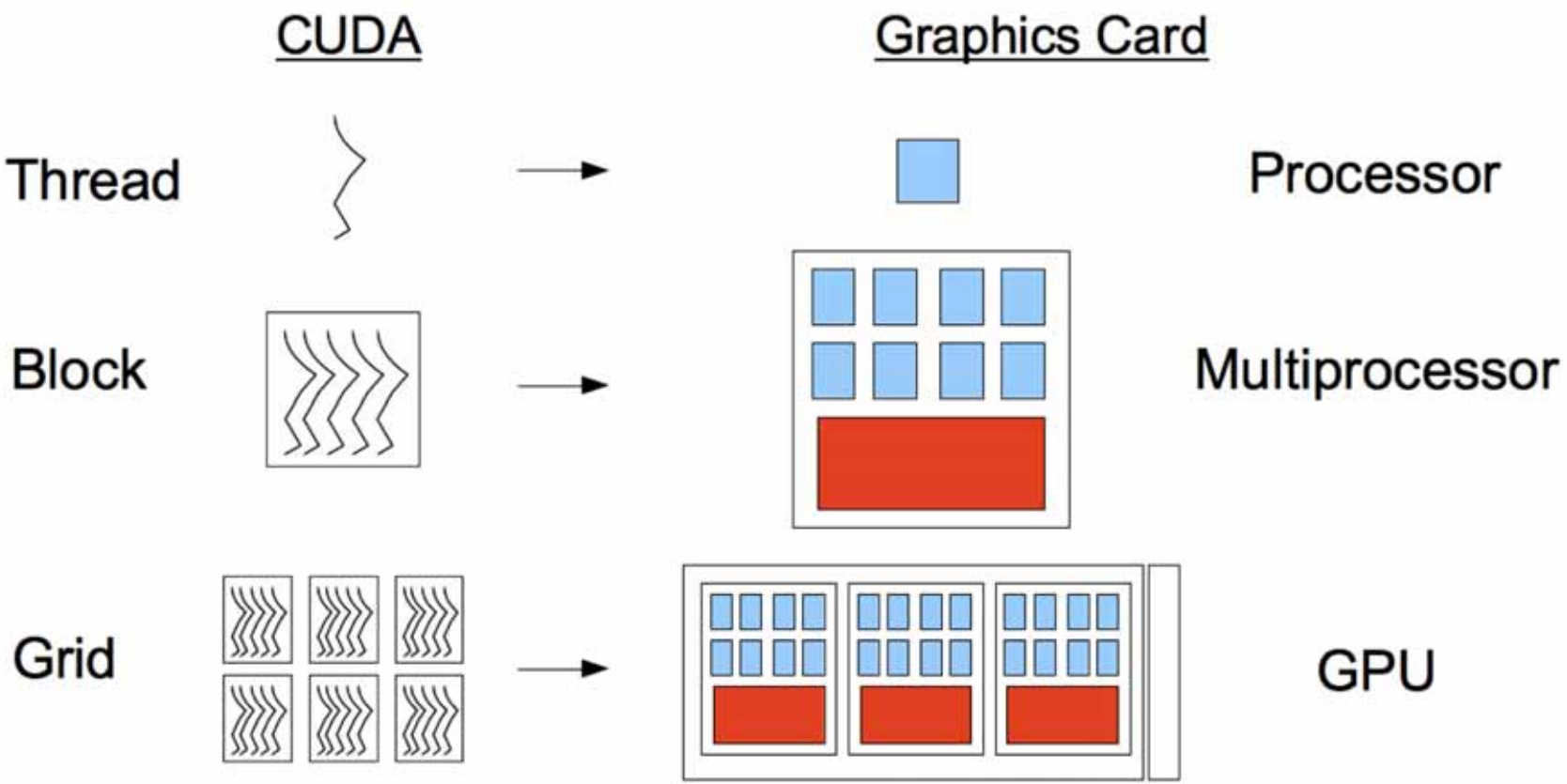
- Effectively the loop is ‘unrolled’ and spread across N threads

Single Instruction Multiple Data (SIMD) principle

Thread blocks

- Threads are grouped in thread blocks, allowing you to work on problems larger than the maximum thread block size
- Thread blocks are also numbered, using the built-in variables `blockIdx.x, y` containing the index of each block within the grid.
- Total number of threads created is always a multiple of the thread block size, possibly not exactly equal to the problem size
- Other built-in variables are used to describe the thread block dimensions `blockDim.x, y, z` and grid dimensions `gridDim.x, y`

Mapping to hardware



Starting a kernel

- The host program sets the number of threads and thread blocks when it launches the kernel

- ```
  //create variables to hold grid and thread block dimensions
  dim3 threads(x, y, z)
  dim3 grid(x, y)

  //launch the kernel
  vector_add<<<grid, threads>>>(c, a, b);

  //wait for the kernel to complete
  cudaDeviceSynchronize();
  ```
  ```

CUDA function declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Setup hands-on session

- You can already do the following:
  - Install VirtualBox (virtualbox.org)

  - Download VM file:
    - scp gpuser@131.155.68.95:GPUtutorial.ova .
      - in terminal (Linux/Mac) or with WinSCP (Windows)
      - Password: cuda2018
    - https://tinyurl.com/y9j5pcwt (10 GB)
    - Or copy from USB stick

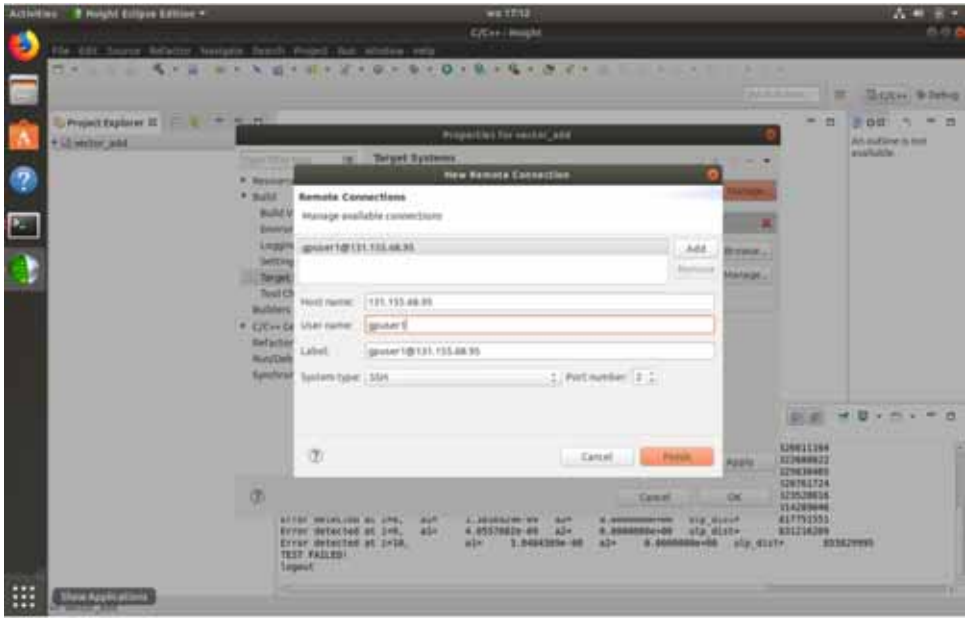
Setup hands-on session

- Import file as Appliance in VirtualBox
- Start the machine
- Login name: *gpuser*
- Login password: *cuda2018*

- Launch **NSight**

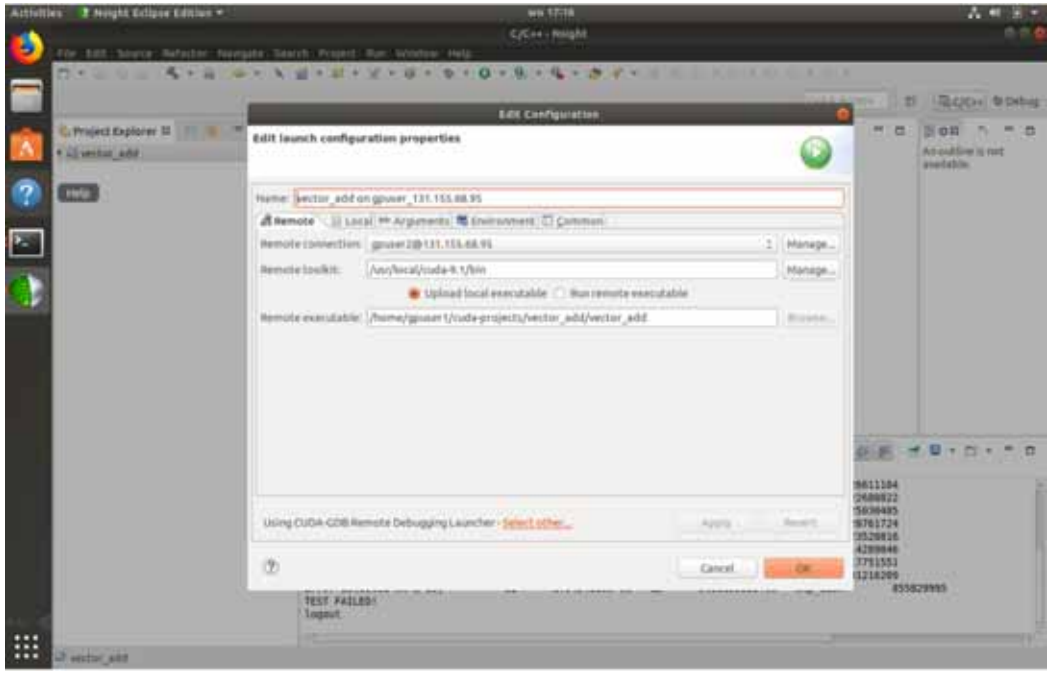
First hands-on session

- Start with project *vector_add* in left pane
- Configure: right click *vector_add* -> Properties; go to Build -> Target Systems -> Manage
- Also update Project Path



First hands-on session

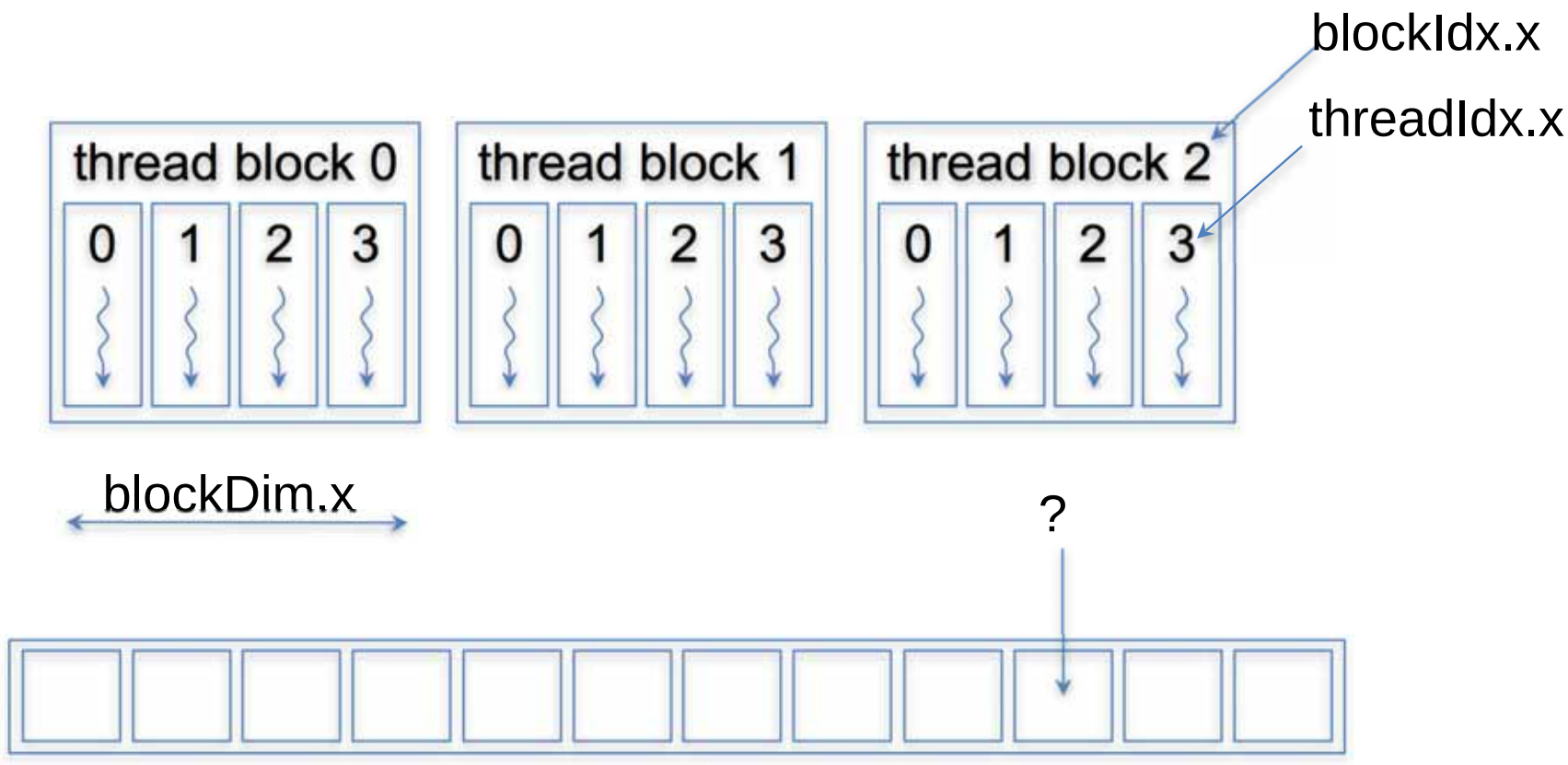
- Configure: go to Run/Debug Settings. Click the configuration -> Edit. Select the remote connection, and set Remote executable folder
- Do these steps for the four projects in the left pane, and **restart Nsight**



1st Hands-on Session

- Make sure you understand everything in the code, and complete the exercise!
- **Hints:**
  - Look at how the kernel is launched in the host program
  - `threadIdx.x` is the thread index within the thread block
  - `blockIdx.x` is the block index within the grid
  - `blockDim.x` is the dimension of the thread block

Hint


blockIdx.x
threadIdx.x
thread block 0
0
1
2
3
thread block 1
0
1
2
3
thread block 2
0
1
2
3
blockDim.x
?

Solution

- CPU implementation:

```
for (i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

- GPU implementation:

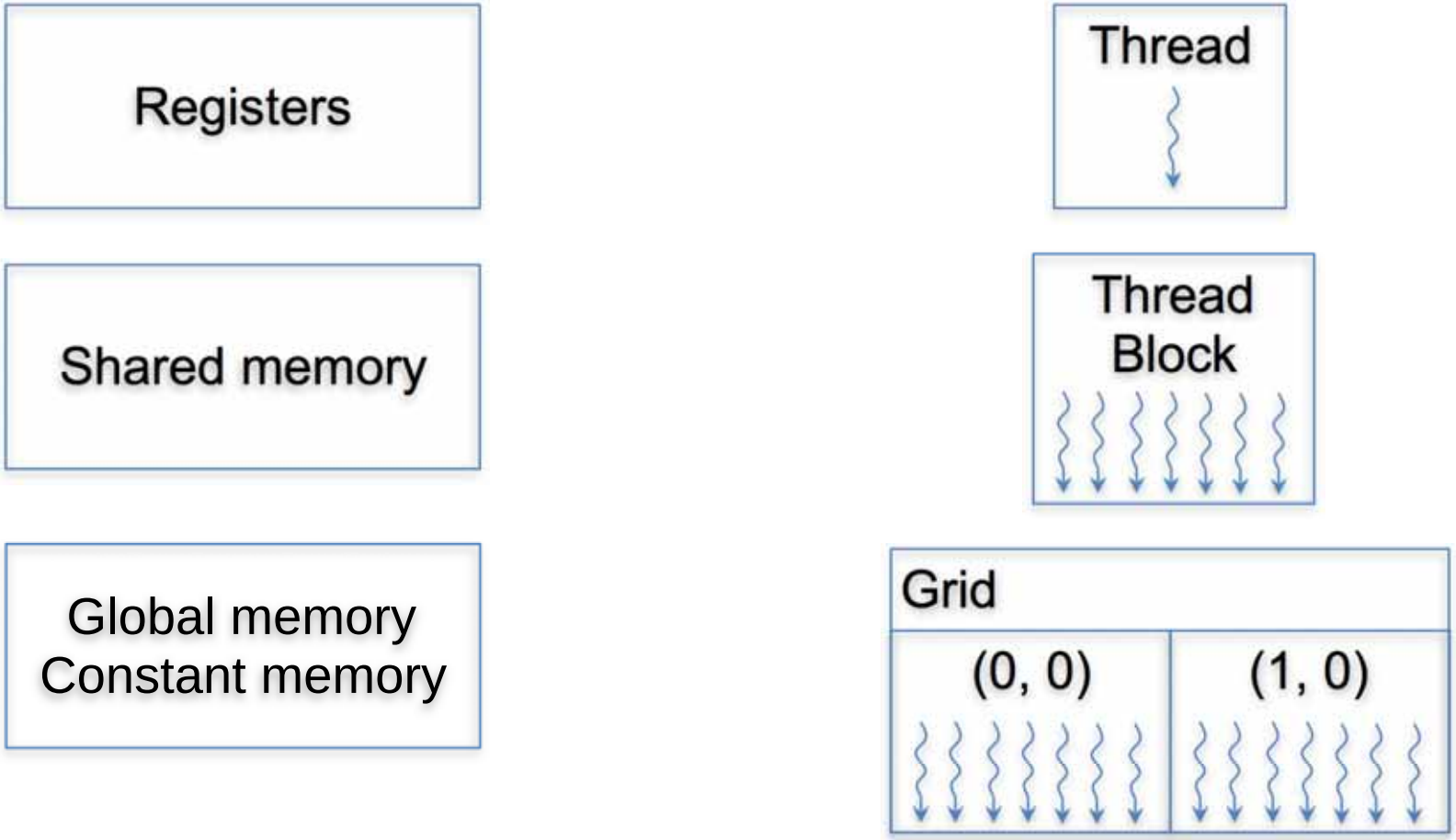
  Create a N threads using multiple thread blocks:

```
i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<N) {
    c[i] = a[i] + b[i];
}
```

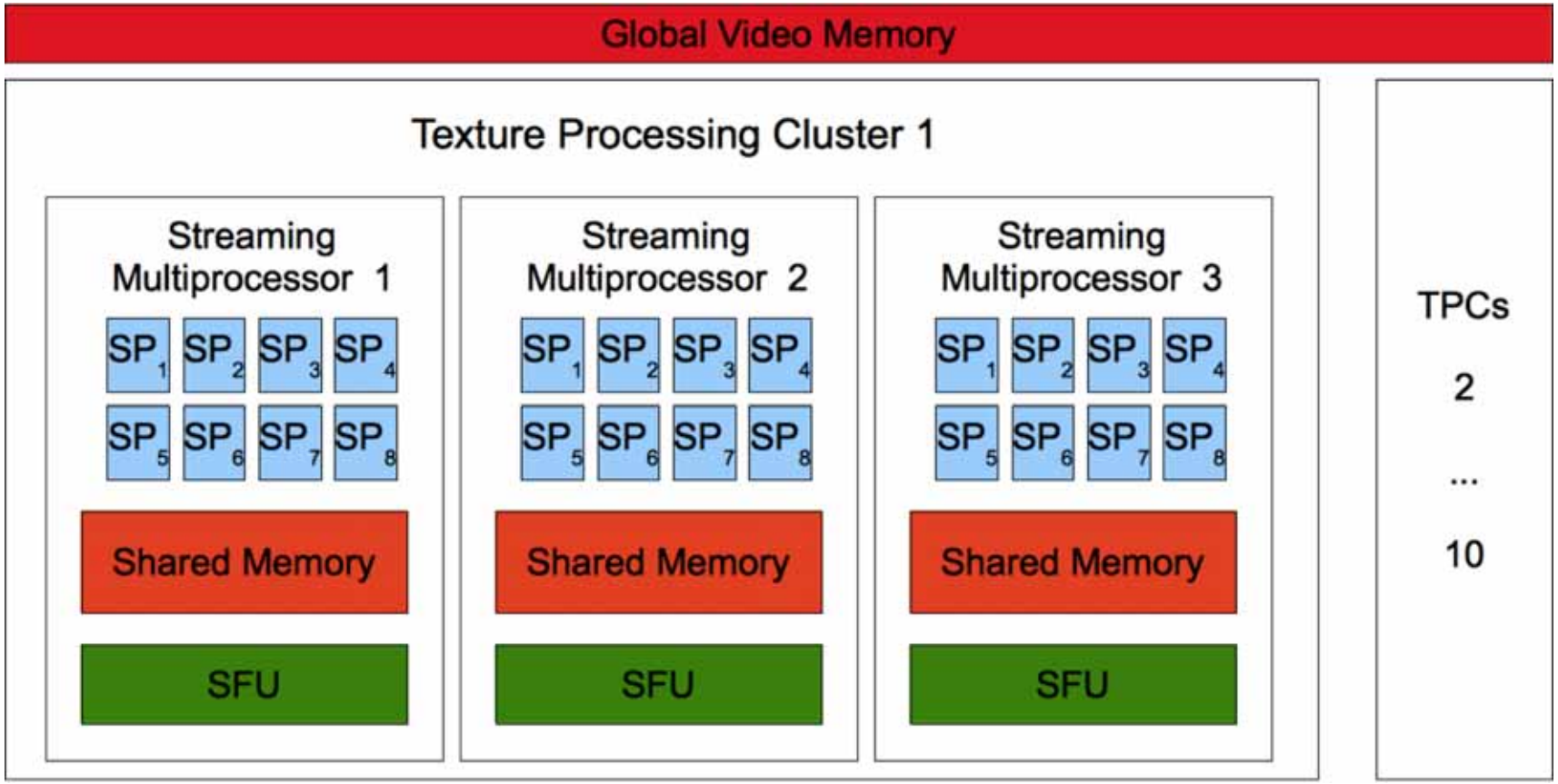
Single Instruction Multiple Data (SIMD) principle

CUDA Programming model
Part 2

CUDA memory hierarchy



Hardware overview



Memory space: Registers

- Example:

```
__global__ void matmul_kernel(float *C, float *A, float *B) {
    int tx = threadIdx.x;      //local variable in registers
    float local_sum[4];        //small compile-time sized array in registers
```

- Registers
  - Thread-local scalars or small constant size arrays are stored as registers
  - Implicit in the programming model
  - Behavior is very similar to normal local variables
  - Not persistent, after the kernel has finished, values in registers are lost

Memory space: Global

- Example:

```
__global__ void matmul_kernel( float *C,  //C points to global memory
                               float *A,  //A points to global memory
                               float *B)  //B points to global memory
{
```

- Global memory
  - Allocated by the host program using `cudaMalloc()`
  - Initialized by the host program using `cudaMemcpy()` or previous kernels
  - Persistent, the values in global memory remain across kernel invocations
  - Not coherent, writes by other threads will not be visible until kernel has finished

Memory space: Constant

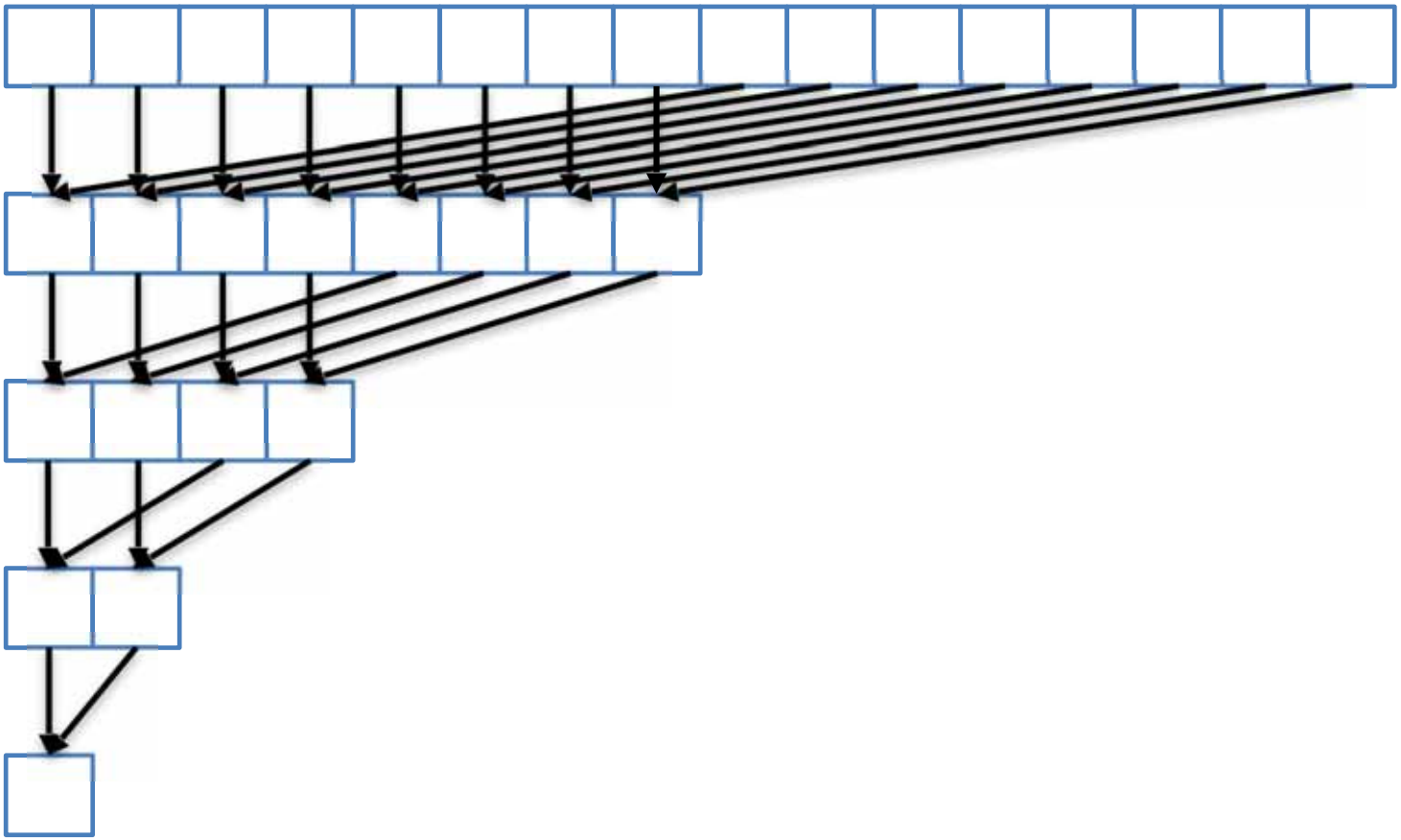
```
__constant__ float filter[filter_width * filter_height]; //initialized by a host function
__global__ void convolution_kernel(float *output, float *input) {
   ...
   for (j = 0; j < filter_height; j++) {
      for (i = 0; i < filter_width; i++) {
         sum += input[y + j][x + i] *
               filter[j * filter_width + i]; //index j and i do not depend on threadIdx (x and y)
      }
   }
```

- Constant memory
  - Statically defined by the host program using `__constant__` qualifier
  - Defined as a global variable
  - Initialized by the host program using `cudaMemcpyToSymbol()`
  - Read-only to the GPU, cannot be accessed directly by the host
  - Values are cached in a special cache optimized for broadcast access by multiple threads simultaneously, access should not depend on `threadIdx`

2nd Hands-on Session

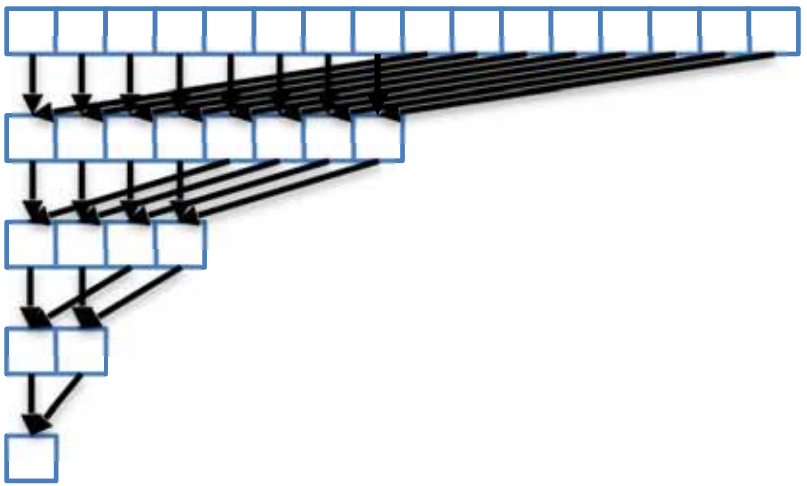
- Go to project *reduction*, look at the source files
- Make sure you understand everything in the code
- **Task:**
  - Implement the kernel to perform a single iteration of parallel reduction
- **Hints:**
  - It is assumed that enough threads are launched such that each thread only needs to compute the sum of two elements in the input array
  - In each iteration, an array of size n is reduced into an array of size n/2
  - Each thread stores it result at a designated position in the output array

Hint – Parallel Summation



Global synchronisation

- CUDA has no mechanism to indicate global synchronisation of all threads across the grid
- Instead, enforce synchronisation points by breaking down computation into multiple kernel launches



Kernel launch 0

Kernel launch 1

Kernel launch 2

Kernel launch 3

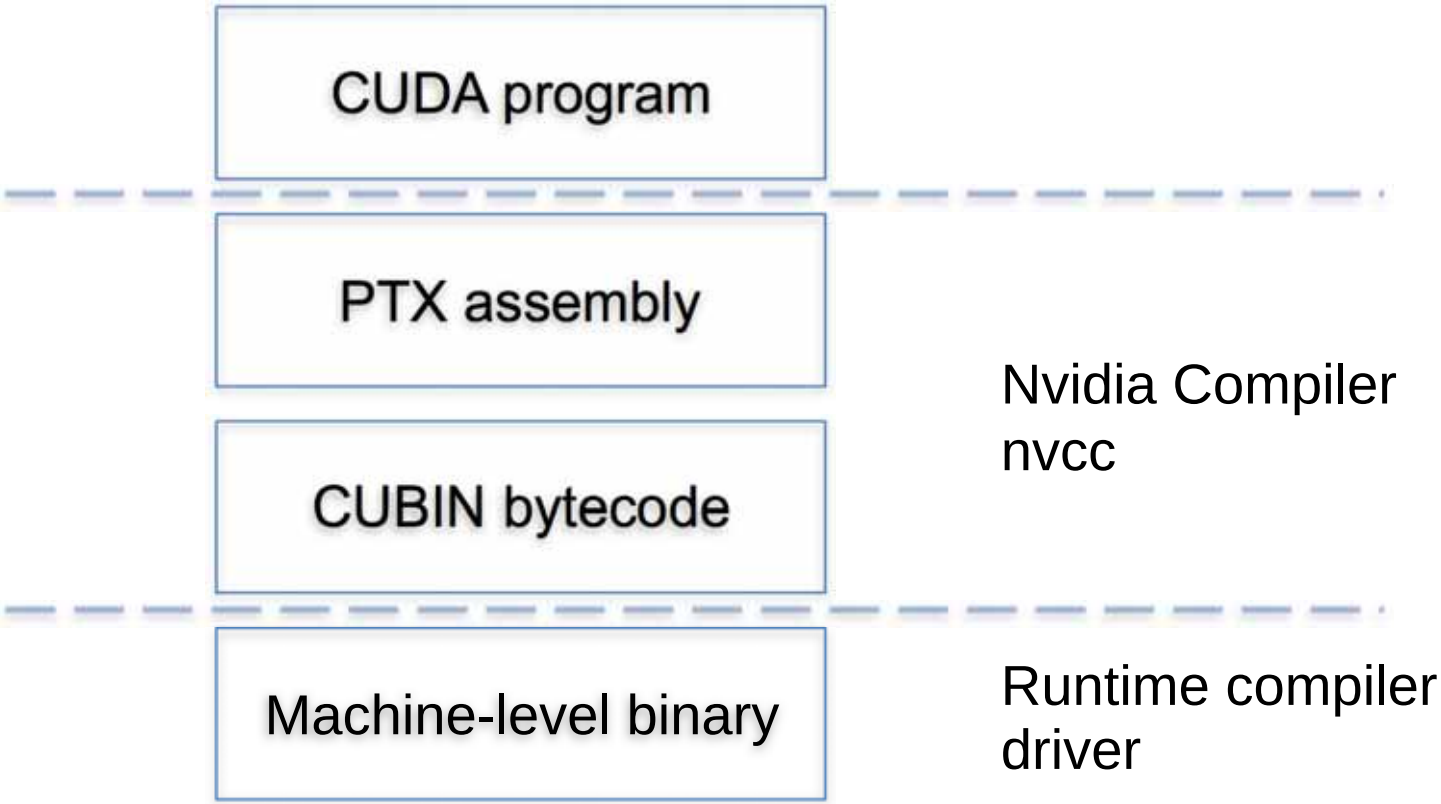
Kernel launch 4

Barrier synchronisation

- Two forms:
  - **Global synchronisation:** achieved between kernel launches
  - **Intra-block synchronisation:** Contrary to global synchronisation, CUDA does provide a mechanism to synchronise all threads in the same block
    - `__syncthreads()`
    - All threads in the same block must reach the `__syncthreads()` before any of them can move on
    - Best used to split up computation of each block in several phases
    - Tightly linked to use of (block-local) **shared memory**, which we will address tomorrow afternoon

CUDA Program execution

Compilation



Translation table

CUDA	OpenCL	OpenACC	OpenMP 4
Grid	NDRange	compute region	parallel region
Thread block	Work group	Gang	Team
Warp	CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE	Worker	SIMD Chunk
Thread	Work item	Vector	Thread or SIMD

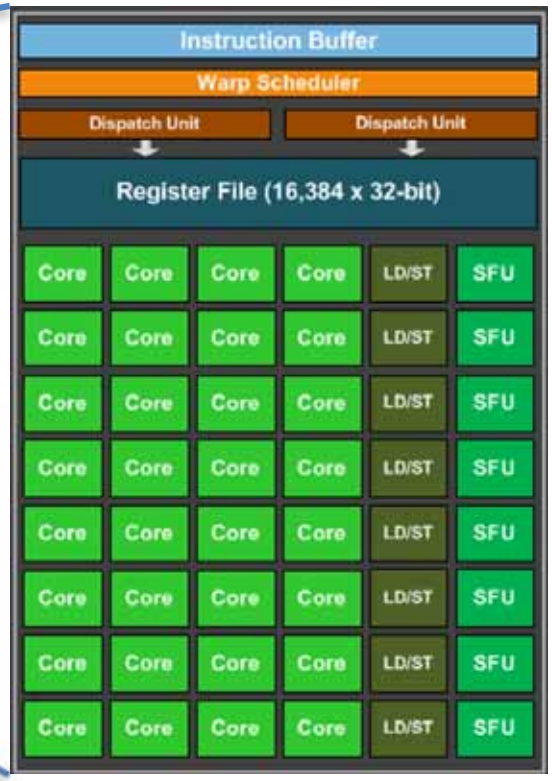
- Note that for the mapping is actually implementation dependent for the open standards and may differ across computing platforms
- Not too sure about the OpenMP 4 naming scheme, please correct me if wrong

How threads are executed

- Remember: all threads in a CUDA kernel execute the exact same program
- Threads are actually executed in groups of (32) threads called *warps* (more on this tomorrow afternoon)
- Threads within a warp all execute one common instruction simultaneously
- The context of each thread is stored separately, as such the GPU stores the context of all currently active threads
- The GPU can switch between warps even after executing only 1 instruction, effectively hiding the long latency of instructions such as memory loads

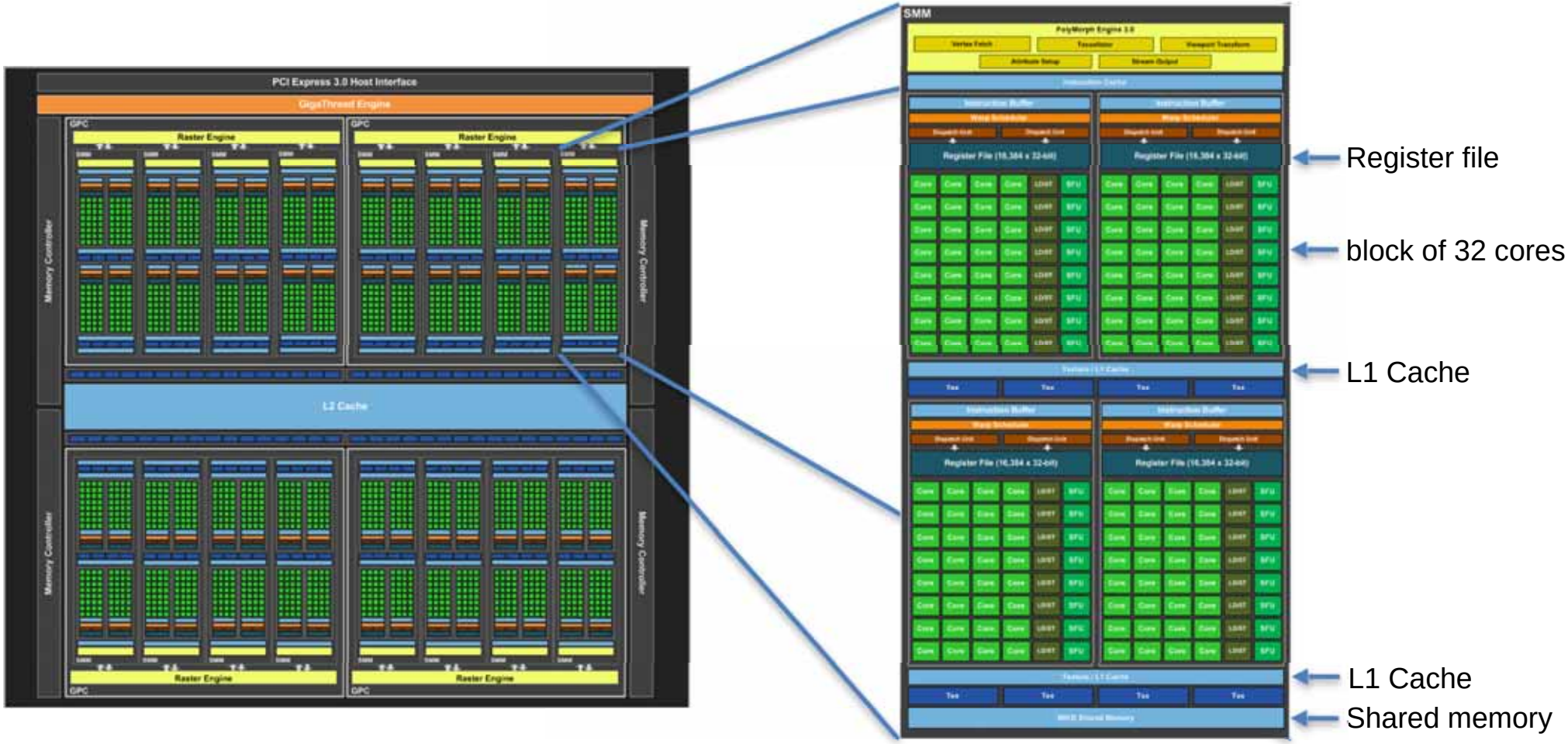
Maxwell Architecture

Streaming multiprocessor (SM)



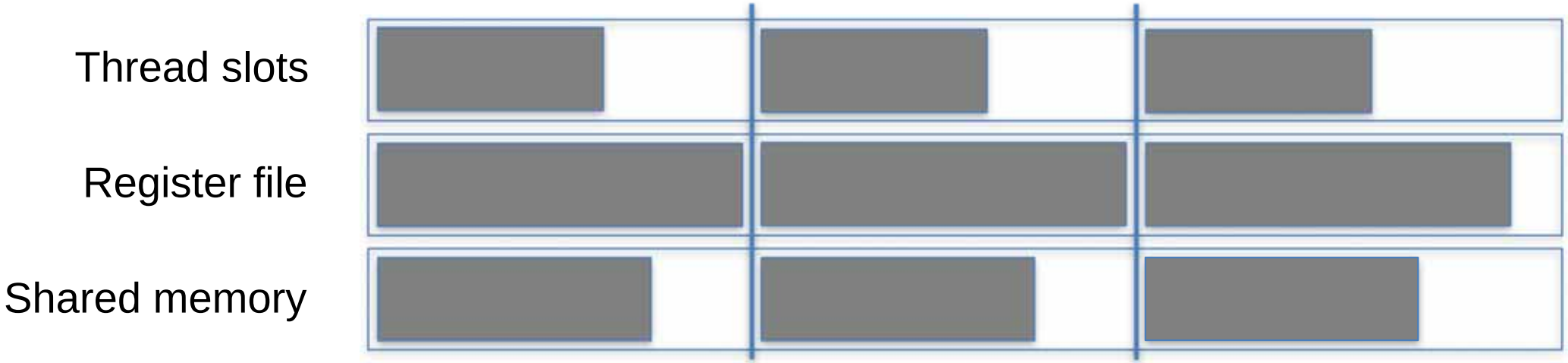
32 core block

Maxwell Architecture



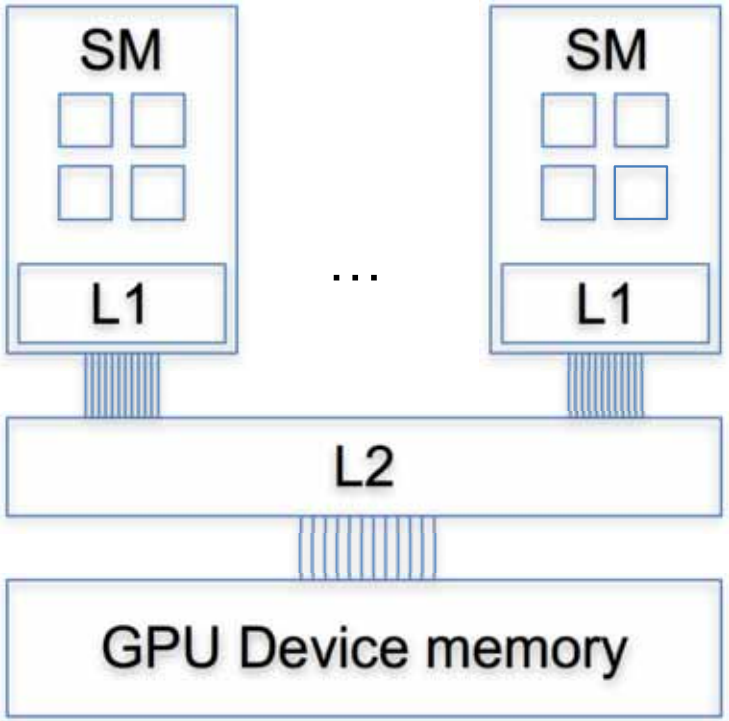
Resource partitioning

- The GPU consists of several (1 to 56) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Thread and Thread Block slots, Register file, and Shared memory
- SM Resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy*



Global Memory access

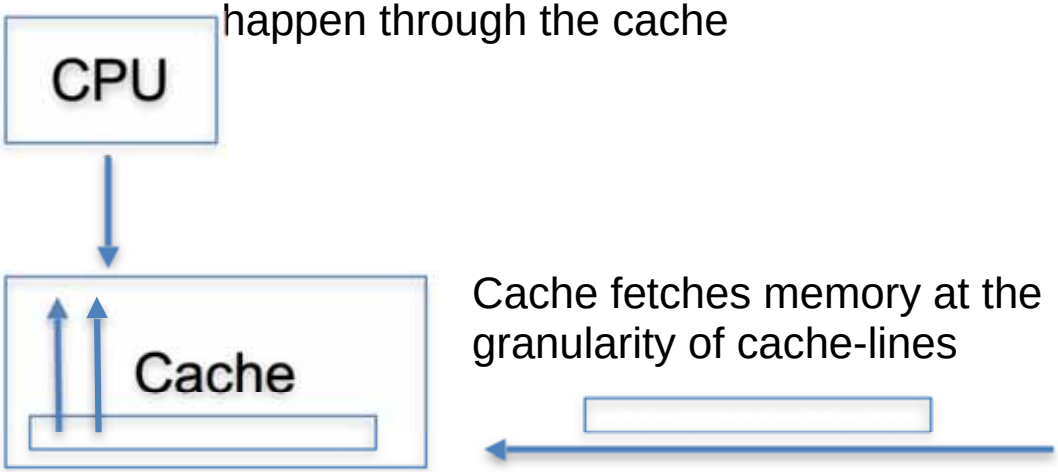
- Global memory is cached at L2, and for some GPUs also in L1
- When a thread reads a value from global memory, think about:
  - The total number of values that are accessed by the warp that the thread belongs to
  - The cache line length and the number of cache lines that those values will belong to
  - Alignment of the data accesses to that of the cache lines



Cached memory access

- The memory hierarchy is optimized for certain access patterns

All memory accesses happen through the cache



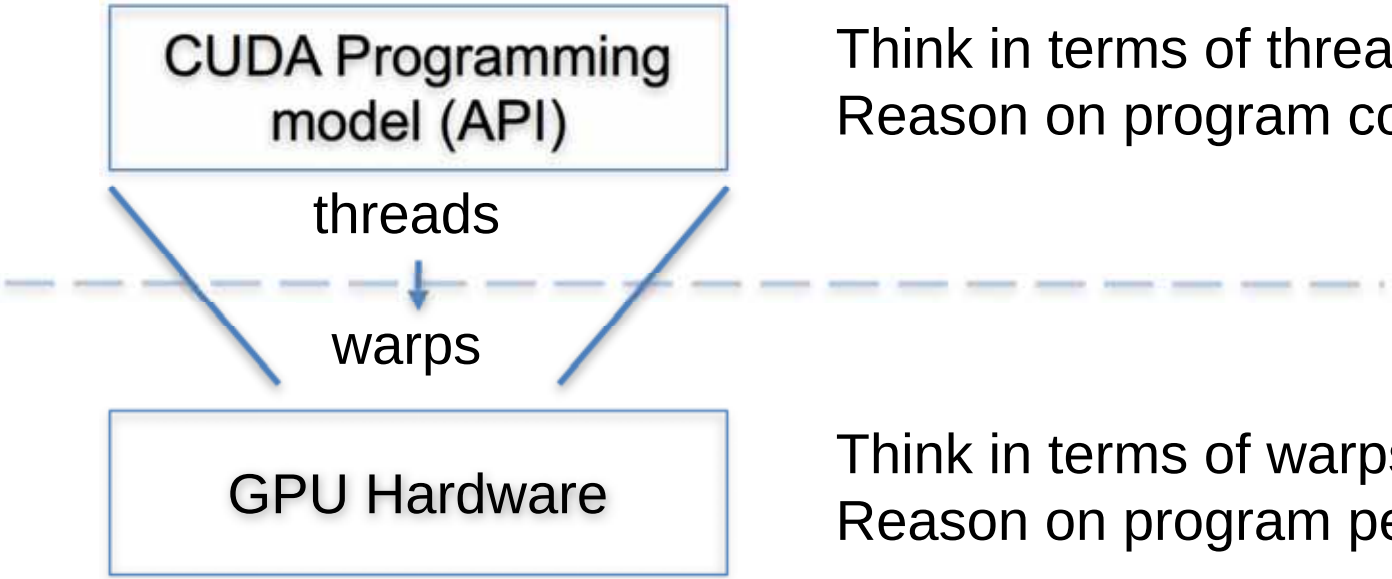
Cache fetches memory at the granularity of cache-lines

Main memory

Memory is optimized for reading in (row-wise) bursts

Overview

Proving correctness tomorrow morning / afternoon!



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

Tomorrow in Part 2 of GPU Development!

To do: setup the VerCors tool

- See https://github.com/utwente-fmt/vercors
- basic build:
  - Clone the VerCors repository:
    `git clone https://github.com/utwente-fmt/vercors.git`
  - Move into the cloned directory:
    `cd vercors`
  - Build VerCors with Ant:
    `ant clean`
    `ant`
  - Test build:
    `./unix/bin/vct --test=examples/manual --tool=silicon –lang=pvl,java`
- If this fails, there will be a VM with VerCors available tomorrow
- Do **NOT** delete your VM with Nsight, as we will use it again tomorrow afternoon!