netherlands eScience center

PRACE
GPU Programming Course

Alessio Sclocco and Ben van Werkhoven
December 16, 2020

Introduction: Alessio Sclocco

Alessio Sclocco
eScience Research Engineer
Netherlands eScience Center

Background:

- 2011-2012 junior researcher at VU Amsterdam
  - Working on GPUs for radio astronomy
- 2012-2017 PhD "*Accelerating Radio Astronomy with Auto-Tuning*" at VU Amsterdam, under the supervision or professors Henri Bal and Rob van Nieuwpoort
- 2015-2016 scientific programmer at ASTRON, the Netherlands Institute for Radio Astronomy
  - Designing and developing a real-time GPU pipeline for the Westerbork radio telescope
- 2019 visiting scholar at Nanyang Technological University in Singapore
- 2017-2020 eScience Research Engineer at the Netherlands eScience Center
  - Radio astronomy, climate modeling

Introduction: Ben van Werkhoven

Ben van Werkhoven
Senior Research Engineer
Netherlands eScience Center

Background:

- 2010-2014 PhD “Scientific Supercomputing with Graphics Processing Units” at the VU University Amsterdam in the group of prof. Henri Bal
- 2014-now working at the Netherlands eScience Center as the GPU expert in many different scientific research projects

GPU Programming since early 2009, worked on applications in computer vision, digital forensics, climate modeling, particle physics, geospatial databases, radio astronomy, and localization microscopy

Schedule

- 14:15 – 14:25 Course introduction
- 14:25 – 14:40 Introduction to GPU programming
- 14:40 – 14:50 GPU-Enabled libraries
- 14:50 – 15:05 Introduction to CUDA programming
- 15:05 – 15:20 Hands-on exercise

- 15:20 – 15:35 Break

- 15:35 – 15:50 CUDA memories part 1
- 15:50 – 16:05 Hands-on exercise
- 16:05 – 16:20 CUDA memories part 2
- 16:20 – 16:50 Hands-on exercise

- 16:50 - 17:05 Break

- 17:05 – 17:45 Program execution model
- 17:45 – 17:55 Closing

Slides

- Get your own copy of the slides so you can read along and click on links
  - See: https://github.com/benvanwerkhoven/gpu-course/
  - Clone the repository to have access to slides and hands-on exercises

- Our slides are sometimes very wordy, this is intentional, so they may serve as a reference that you can read again later

- In code samples on the slides we sometimes abbreviate the code a bit to save space

Online Course Organization

- Video chat for the event
  - Used for lectures and discussion
  - Keep your microphone muted while not talking

- Questions
  - Raise your hand
  - Write in the chat if the answer can wait until the end of the presentation
    - We will answer all written questions at the end of each module

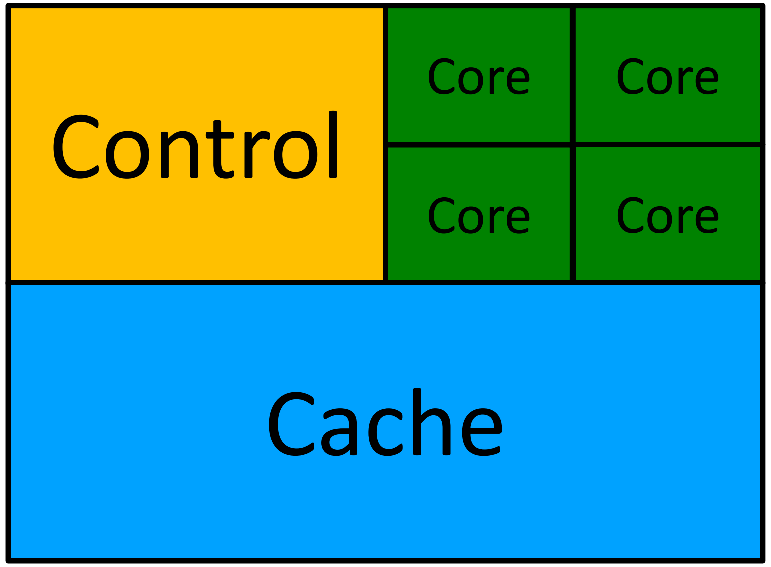
- Hands-on interaction
  - Write questions in the global chat if relevant to others
  - Contact Alessio or Ben for specific questions
    - Use private chat or have a video call

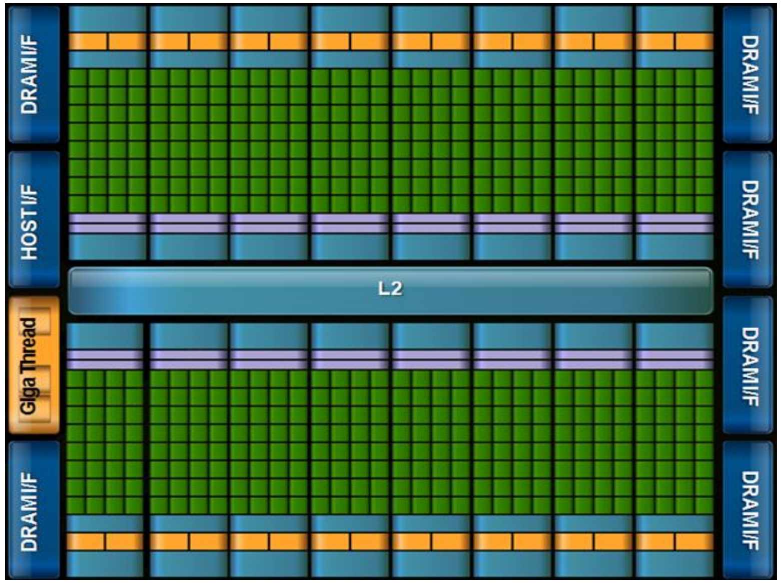
Introduction to GPU Programming

14:25 – 14:40

Design: GPUs vs CPUs

- Different goals produce different designs
  - GPU assumes that the workload 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



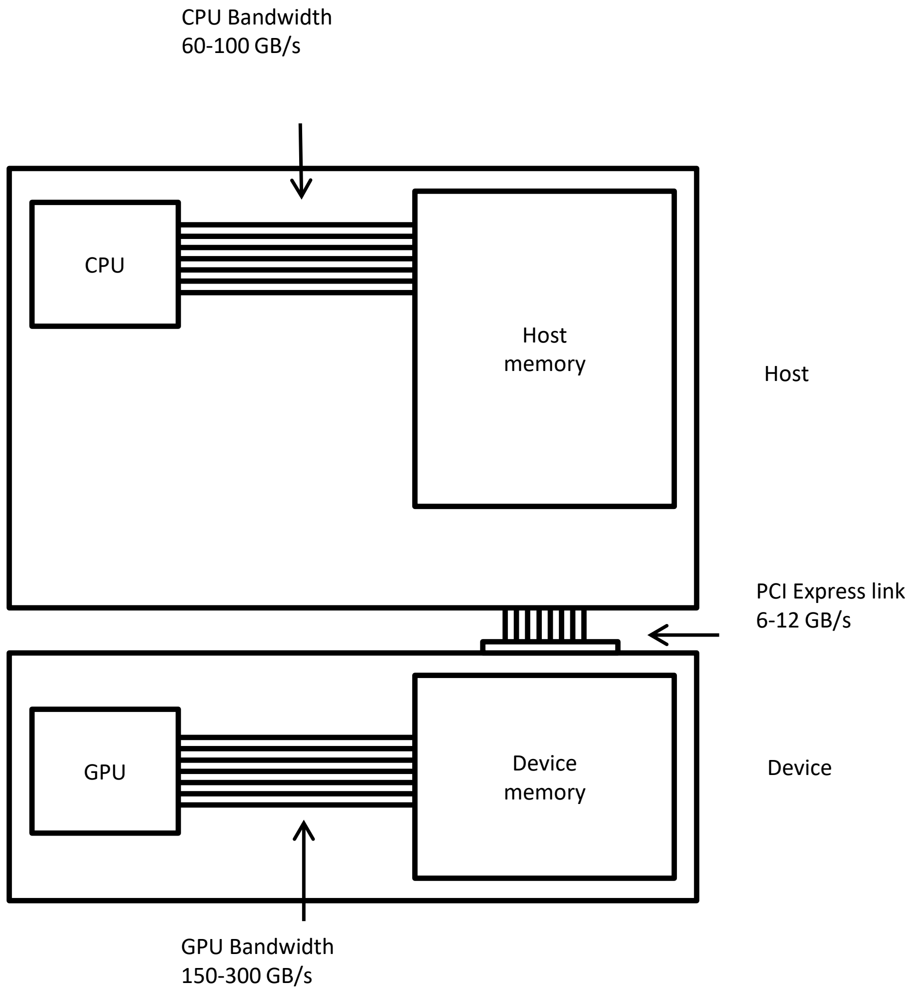


The computer architecture is very different:

- Algorithms need to be parallelized and mapped to the hardware
- Requires software to be rewritten in specialized programming language
- Optimizing for compute performance requires knowledge about hardware

GPUs are on separate devices:

- Must deal with separate memory space, limited bandwidth between host and device memory



GPU Programs consists of a host (CPU) and a device (GPU) part

The host part manages:

- Both host and device memory
- Data transfers between host and device memory
- Starting device *kernels* (functions on the device)

The device part consist of kernels, that:

- Are executed by huge amounts of parallel threads at the same time
- Divide the data-parallel workload among these threads
- Switches execution between groups of threads to hide memory latency

For the host code:

- Several language bindings for GPU Programming exist:
  - C/C++: CUDA and OpenCL
  - Python: PyCuda and PyOpenCL for CUDA and OpenCL programming
  - Java: JCuda and JOCL
  - Fortran: CudaFortran
  - Matlab: MexCuda (using mexfiles)

For the device code:

- Basically, three options:
  - Write your own kernels in CUDA or OpenCL
  - Use GPU-enabled libraries (kernels written by someone else)
  - GPU Code generators (kernels written by compilers)

Managing GPU Memory

- GPU memory is typically smaller than host memory (12GB vs 64GB)
- Multiple GPUs each have their own device memory space
- Data copied to the GPU may become stale on the host

- Transferring data to the GPU is expensive (because of the relatively low PCIe bandwidth, better with NVLink)

- In general, it is best to keep working on transferred data for as long as possible

- It is possible to overlap data transfers with GPU computations and data transfers in the opposite direction

Optimizing for Performance

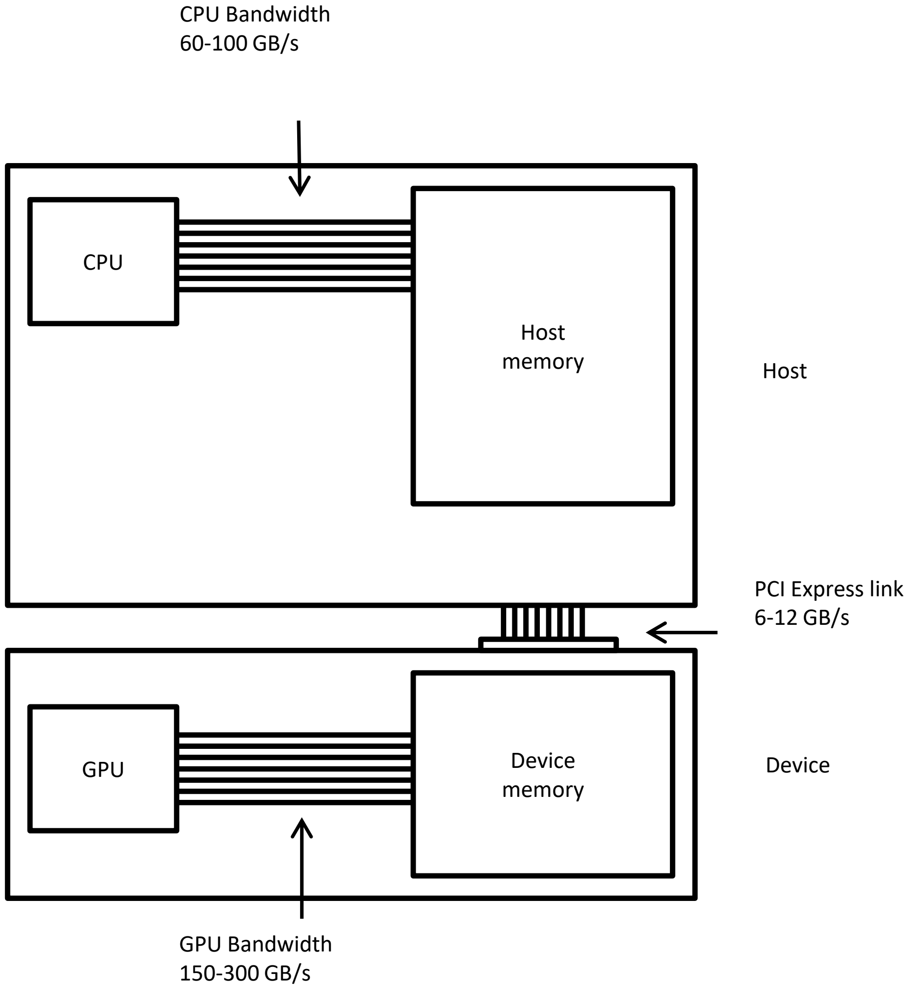
- There are many code optimizations that can be parameterized:
  - The number of threads per thread block in each dimension
  - Loop unrolling factors
  - The number of items processed per thread
  - The total work per thread block
  - Different schemes for using shared memory
  - Different parallelization schemes

- Optimizing GPU code is mostly finding the best performing combination for all parameters

- Auto-tuners can be used to automate the search process

Main differences CPU and GPU programming:

1. Algorithms need to be parallelized and mapped to the hardware
2. Requires software to be rewritten in specialized programming language
3. Optimizing for compute performance requires knowledge about hardware
4. Must deal with separate memory space, limited bandwidth between host and device memory



GPU-Enabled Libraries

14:40 – 14:50

GPU-Enabled Libraries

- Generally, the user is responsible for managing GPU memory
- Often use specialized objects that represents data in GPU memory
- Easy access to highly-optimized and tuned GPU routines

- Either focused on specific functionality or offering a ‘GPU Array’-like datatype

Function-Oriented Libraries

- Fast Fourier Transforms: cuFFT, clFFT, hipFFT, rocFFT, vkFFT
- BLAS (linear algebra): cuBLAS, clBlas, rocBlas, clBlast (auto-tuned), SYCL-BLAS
- Random number generation: cuRAND, rocRAND
- Sparse matrix operations: cuSparse, hipSparse
- Deep neural networks: cuDNN, OpenCV DNN, SYCL-DNN

- Practically all of these can be used directly from C++, many have Python bindings, bindings for other languages are not that commonly available or only supported by relatively small open source projects

Datatype-Oriented Libraries

- Matlab: gpuArray
  - Provides access to many operations using cuBLAS, cuFFT underneath
  - also JIT-compiles groups of pointwise array operations into CUDA kernels

- Python
  - CuPy: A NumPy-compatible array library accelerated by CUDA
    - Includes functionality for compiling ‘raw’ CUDA kernels
    - Includes bindings to cuBLAS, cuFFT, cuDNN, …
  - PyTorch: Open source machine learning framework
    - Includes Tensor data type with CUDA backend
    - Can be used to interface cuBLAS, cuRAND, cuFFT, cuDNN, cuSPARSE, …
  - Numba: Open source JIT compiler for Python/Numpy code
    - Compiles to CUDA or ROCm
    - Includes Python bindings to CUDA
  - cuDF: A Pandas-like GPU DataFrame library
    - Supports operations for loading, joining, aggregating, filtering, and otherwise manipulating data
    - Integrates with Dask for distributed and out-of-core computations

- ArrayFire, can be used from C, Rust, or Python
    - Includes functions for many image processing, linear algebra, and machine learning operations

CUDA SDK Samples include many simple example codes to illustrate how to use cuBLAS, cuFFT, and so on: https://developer.nvidia.com/cuda-code-samples

Examples on how to use libraries from AMD’s ROCm platform are included in the documentation: https://rocmdocs.amd.com

The datatype-oriented libraries often include their own memory managers, which can be great but sometimes complicates interoperability.

If you are not using C++ or Python, it is generally possible to write a small C++ code that calls the library function and can be called from another language, e.g. Fortran.

Introduction to CUDA Programming

14:50 – 15:05

Before we start:

- We are going to explain the CUDA Programming model
- We will try to avoid talking about the hardware for now
- For the moment, make no assumptions about the backend or how the program is executed by the hardware
- We 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
- Note that most CUDA code can be translated to HIP using Hipify to run on AMD hardware

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 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

General Parallel Programming Principles

- Parallelizing a computation sometimes requires to rethink your algorithms, for example:

```
//some sort of stencil, performs 1 read for every 3 writes
for (int i=1; i<N-1; i++) {
  double my_a = a[i];
  a_new[i-1] = 0.25*my_a;
  a_new[i] = 0.5*my_a;
  a_new[i+1] = 0.25*my_a;
}

//more or less the same, but with 3 reads for every 1 write
for (int i=1; i<N-1; i++) {
  a_new[i] = 0.25*a[i-1] + 0.5*a[i] + 0.25*a[i+1];
}
```

The latter is much easier to parallelize because it avoids concurrent writes to the same memory locations

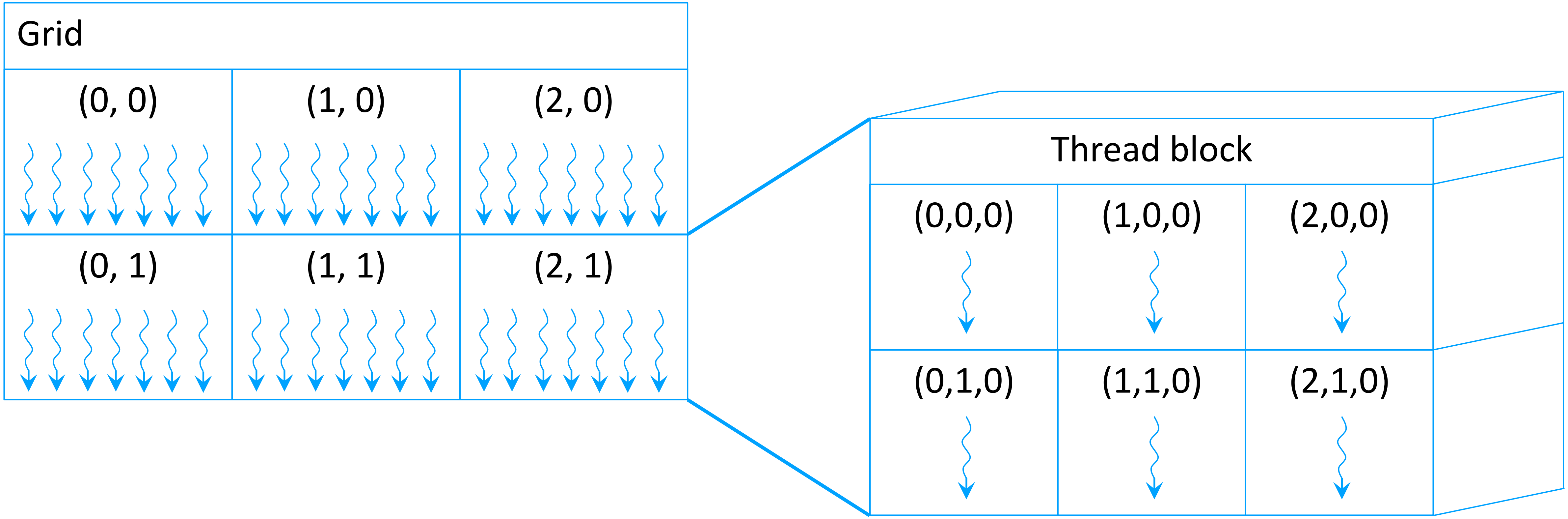
Kernel Programming Language

- The programming language for kernels is CUDA. It’s mostly C/C++, but with some additions and limitations

- Additions:
  - Function qualifiers `__global__`, `__device__`, and `__host__` can be used to declare a function as being a kernel, a device function, or a host function
  - Kernel and device functions have built-in variables, like threadIdx.xyz or blockIdx.xyz
  - Memory qualifiers `__constant__` and `__shared__` can be used to declare a variable to reside in either constant or shared memory space
- Limitations:
  - You cannot use any existing C functions, only functions with the `__device__` qualifier can be called from kernels.
  - A lot of standard C library functionality is not present, for example there is no `malloc()`, and for the first couple of years of CUDA there wasn’t even a `printf()` function

Thread Hierarchy

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



- In the CUDA programming model a thread is the most fine-grained entity that performs computations
- Threads within a kernel all execute the same program
- Threads direct themselves to different parts of memory using their built-in variables `threadIdx.xyz` (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

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 variable `blockIdx.xy` 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.xyz` and grid dimensions `gridDim.xy`

Starting a Kernel In C/C++

- 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, z);

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

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

Starting a Kernel In Python

- 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
threads = (x, y, z)
grid = (x, y, z)

# launch the kernel
vector_add([c, a, b], block=threads, grid=grid)

# wait for the kernel to complete
context.synchronize()
```

Hands-on Exercise

15:05 – 15:20

Setup Hands-on Session

- Login on Cartesius
- Execute (recommended to add to your .bashrc):
  - `module load 2020`
  - `module load NVHPC/20.7`
  - `alias gpurun="srun -N 1 –p gpu_short"`

- Load Python 3
  - `module load Python/3.8.2-GCCcore-9.3.0`
- Install python packages
  - `python –m pip install numpy`
  - `python –m pip install pycuda` (make sure you've typed `module load NVHPC/20.7` first)
  - `python –m pip install kernel_tuner`

- More information about running jobs on Cartesius
  - https://userinfo.surfsara.nl/systems/cartesius/usage/batch-usage

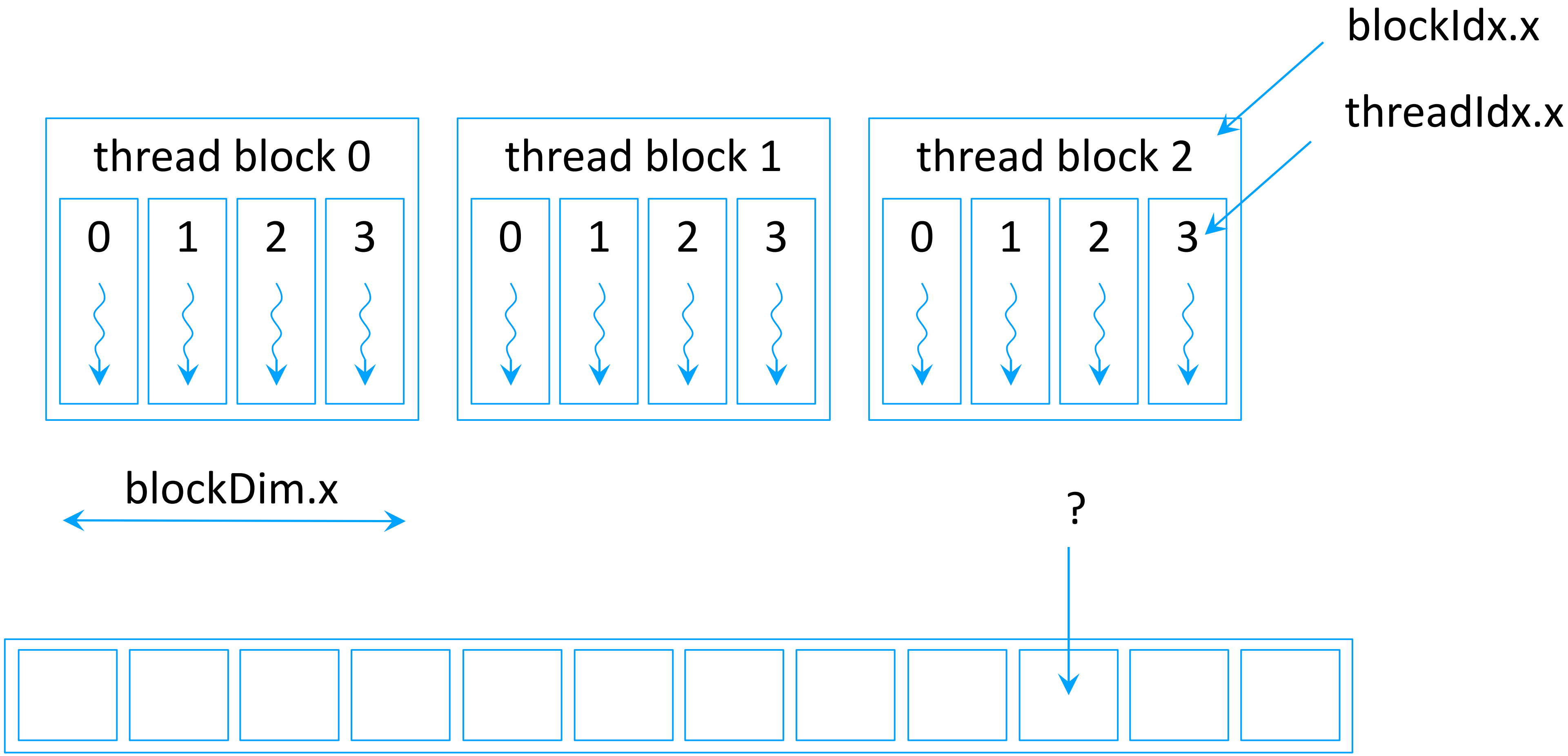
First Hands-on Session: Vector Add

- Select the first exercise
  - https://github.com/benvanwerkhoven/gpu-course/tree/master/vector_add

- Make sure you understand everything in the code, and complete the exercise!

- Hints:
  - Look at how the kernel is launched in the host program
    - https://documen.tician.de/pycuda/driver.html#pycuda.driver.Function
  - `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
thread block 1
thread block 2
0
1
2
3
0
1
2
3
0
1
2
3
blockDim.x
?

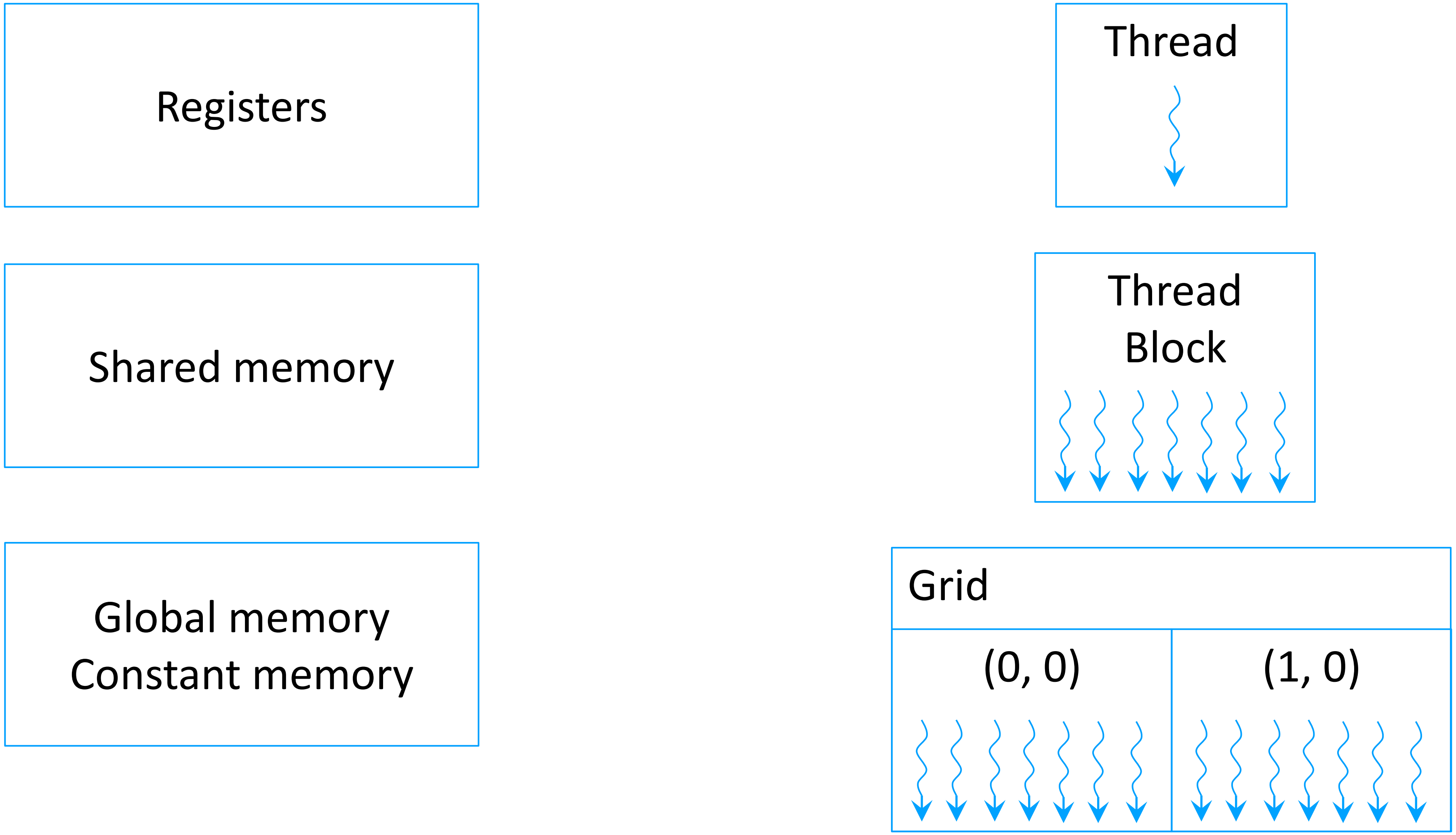
Break

15:20 – 15:35

CUDA Memories part 1

15:35 – 15:50

CUDA Memory Hierarchy



Registers

Shared memory

Global memory
Constant memory

Thread

Thread
Block

Grid

(0, 0)

(1, 0)

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, visible only within the same translation unit
  - Initialized by the host program using
    - C/C++ `cudaMemcpyToSymbol()`
  - Read-only for 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`

Hands-on Exercise

15:50– 16:05

Second Hands-on Session: Point in Polygon

- Select the second exercise
  - https://github.com/benvanwerkhoven/gpu-course/tree/master/pnpoly

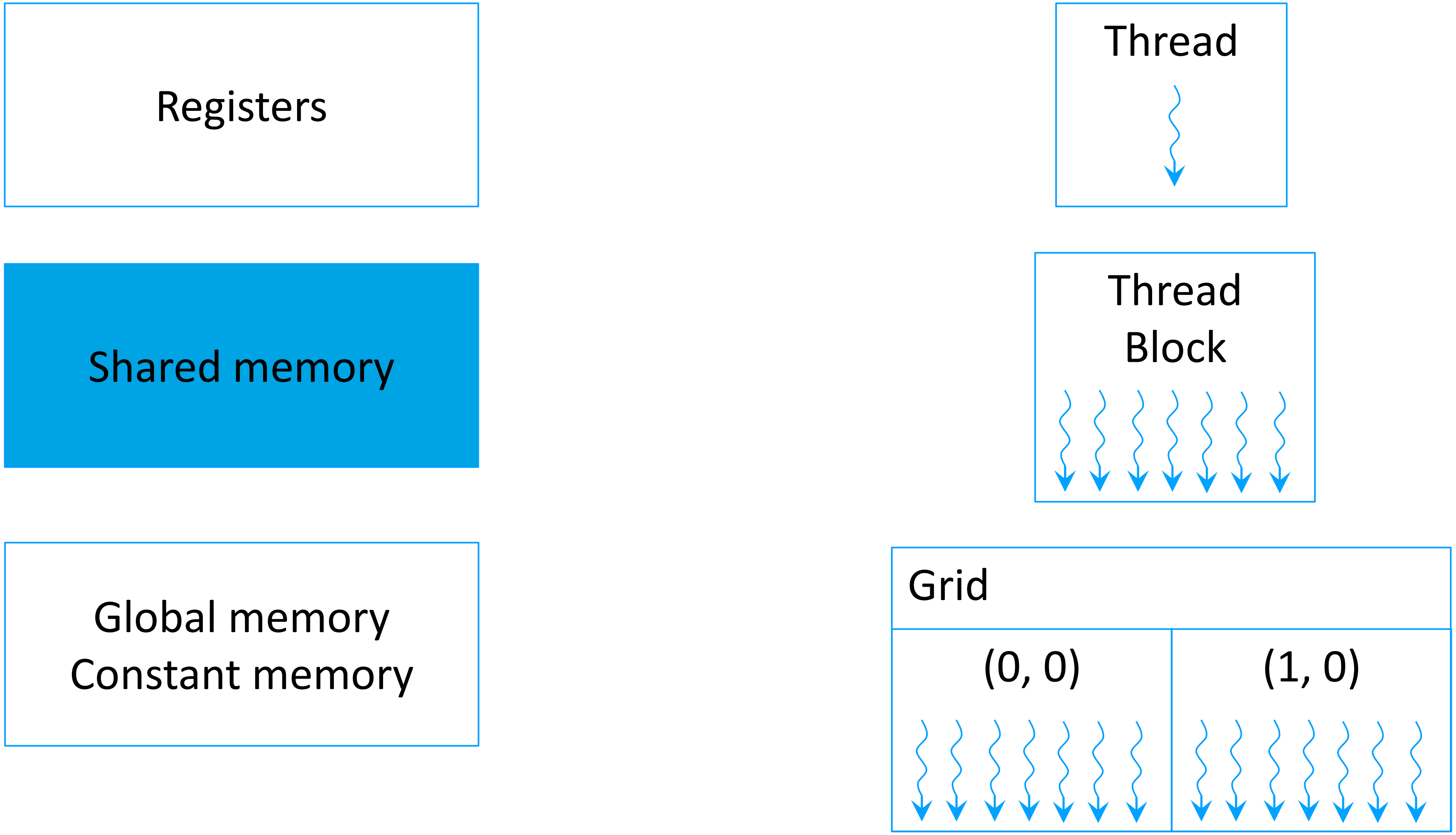
- Make sure you understand everything in the code, and complete the exercise!

- Hints:
  - Python users can use `memcpy_htod()`, but need to find the symbol to copy to
  - See PyCuda documentation on `get_global`

CUDA Memories part 2

16:05 – 16:20

CUDA Memory Hierarchy



Registers

Shared memory

Global memory
Constant memory

Thread

Thread Block

Grid

(0, 0)

(1, 0)

Memory Space: Shared

```
__global__ void histogram(int *output, int *values, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ int sh_output[NUM_BINS];                   //declare shared memory array
    if(i < n) {
        int bin = values[i];
        atomicAdd(&sh_output[bin], 1);                     //increment bin in shared memory
        __syncthreads();                                  //wait for all threads
    ...
```

- Shared memory
  - Variables have to be declared using `__shared__` qualifier, size known at compile time
  - In the scope of a thread block, all threads in a thread block see the same piece of memory
  - Not initialized, threads have to fill shared memory with meaningful values
  - Not persistent, after the kernel has finished, values in shared memory are lost
  - Not coherent, `__syncthreads()` is required to make writes visible to other threads within the thread block

Shared Memory: Example

```
__global__ void transpose(int h, int w, float* output, float* input) {
    int i = threadIdx.y + blockIdx.y * block_size_y;
    int j = threadIdx.x + blockIdx.x * block_size_x;

    __shared__ float sh_mem[block_size_y][block_size_x];     //declare shared memory array

    if (j < w && i < h) {
       sh_mem[threadIdx.y][threadIdx.x] = input[i*w+j];      //fill shared with values from global
    }
    __syncthreads();                                         //wait for all thread in block

    i = threadIdx.x + blockIdx.y * block_size_y;
    j = threadIdx.y + blockIdx.x * block_size_x;
    if (j < w && i < h) {
       output[j*h+i] = sh_mem[threadIdx.x][threadIdx.y];     //store to global using shared memory
    }
}
```

Hands-on Exercise

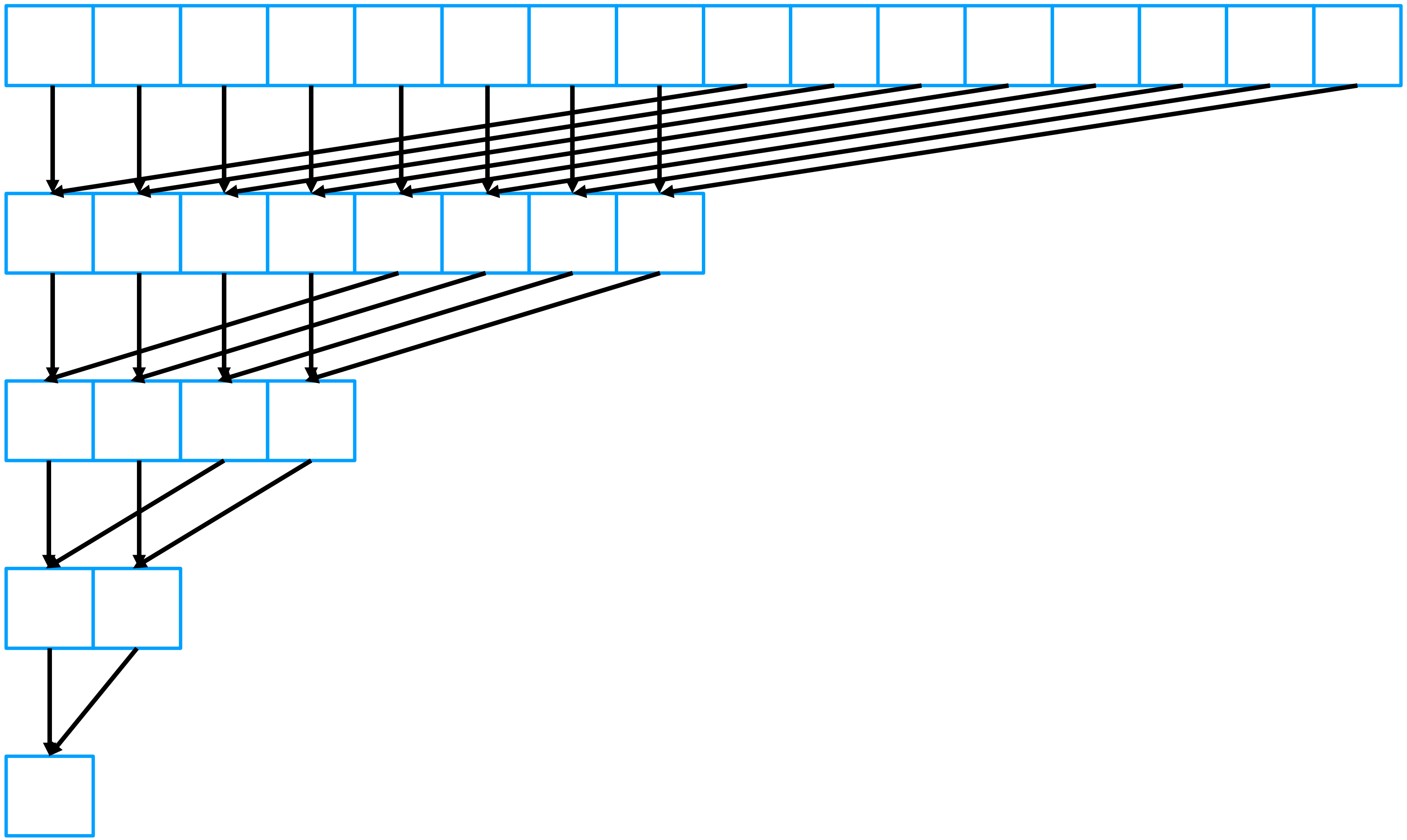
16:20 – 16:50

- Select the notebook for the third exercise
    - https://github.com/benvanwerkhoven/gpu-course/tree/master/reduction

- Implement the kernel such that shared memory is used to sum the per-thread partial sums into a single per-thread block partial sum
- Make sure you understand everything in the code, and complete the exercise!

- Hints:
    - The number of thread blocks does not depend on n. All threads from all blocks first iterate (collectively) over the problem size (n) to obtain a per-thread partial sum
    - Within the thread block the per-thread partial sums are to be combined to a per-thread block partial sum
    - Each thread block stores its partial sum to `out_array[blockIdx.x]`
    - The kernel is called twice, the second kernel is executed with only one thread block to combine all per-block partial sums to a single sum

Hint – Parallel Reduction



Break

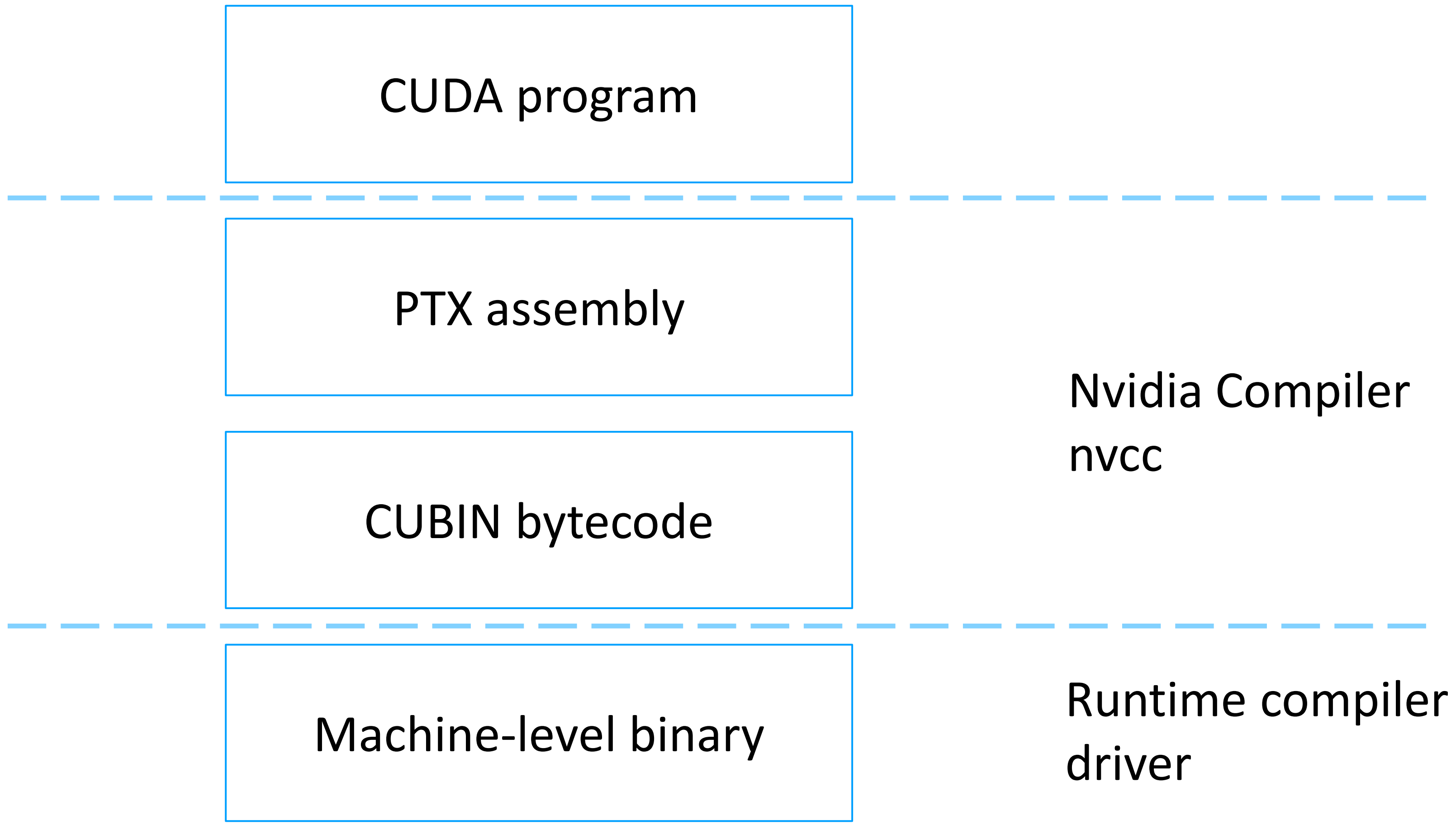
16:50 – 17:05

netherlands eScience center

Program Execution Model

17:05 – 17:45

Compilation



CUDA program

PTX assembly

CUBIN bytecode

Machine-level binary

Nvidia Compiler
nvcc

Runtime compiler
driver

Translation Table

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

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

- Remember: all threads in a CUDA kernel execute the exact same program
- Threads are actually executed in groups of (32) threads called *warps*
- 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

Predication

- All threads in a warp execute the exact same *instruction* at the same cycle

```
        mad.f32    %f1, %f2, %f3, %f1;         // c += a*b;
```

- The same instruction, but on different data

- What about control flow instructions? (if, else, for, while)
  - All threads in the warp execute all live paths, with some threads predicated

```
    if (a > 0.0f)
```

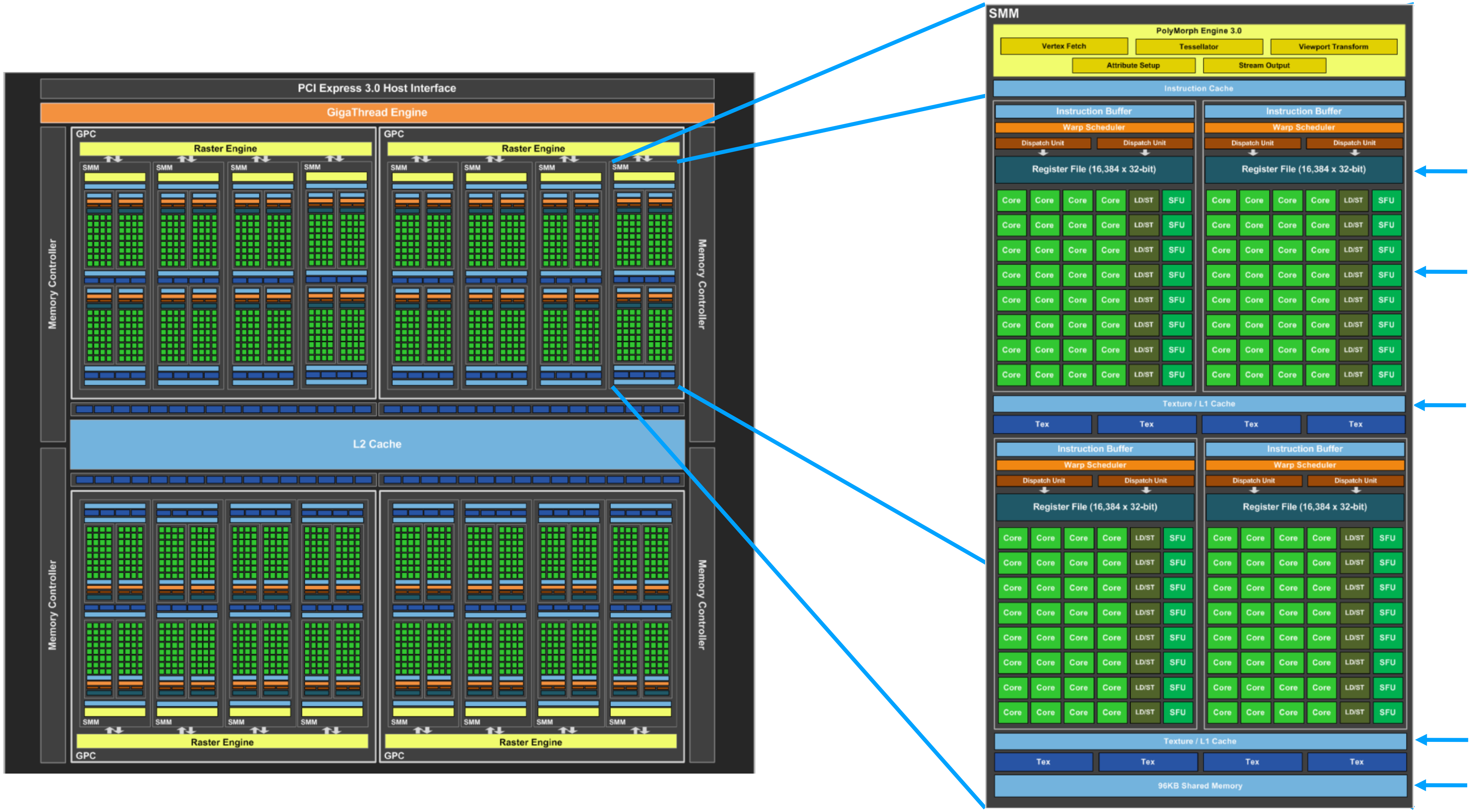
  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

- Thread index-dependent branching is usually harmless, in particular when you respect the warp size

```
        if (threadIdx.x < 32)
```

- The Volta architecture replaces predication with a per-thread program counter and call stack. The same performance recommendations apply however.

Maxwell Architecture



Register file

block of 32 cores

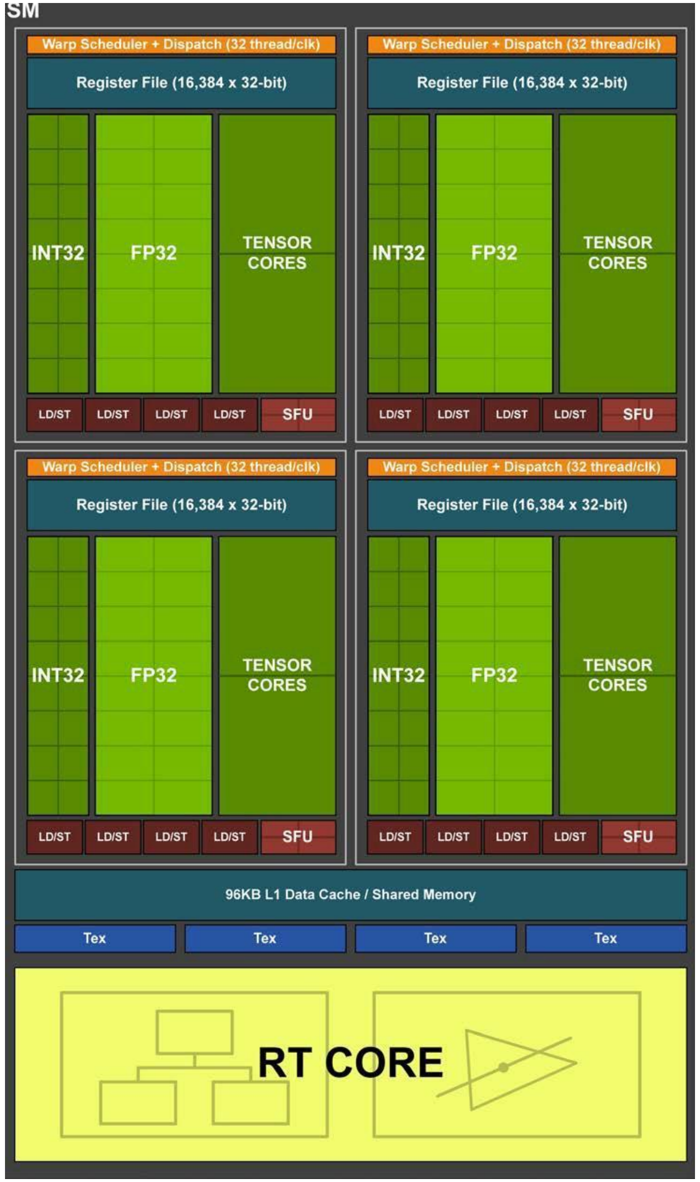
L1 Cache

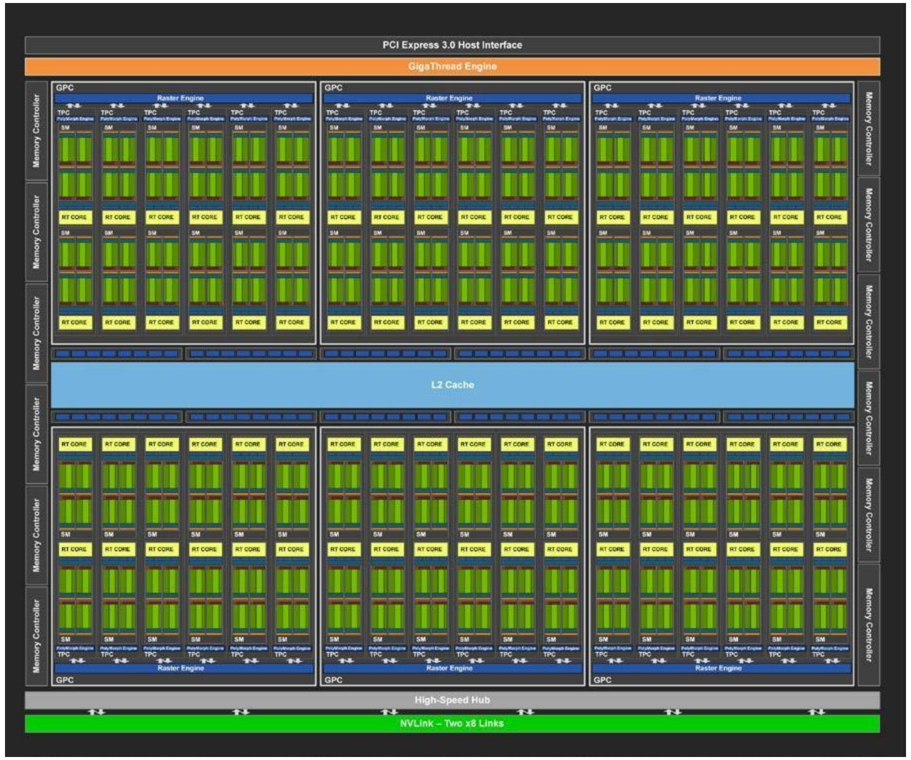
L1 Cache

Shared memory

Turing Architecture

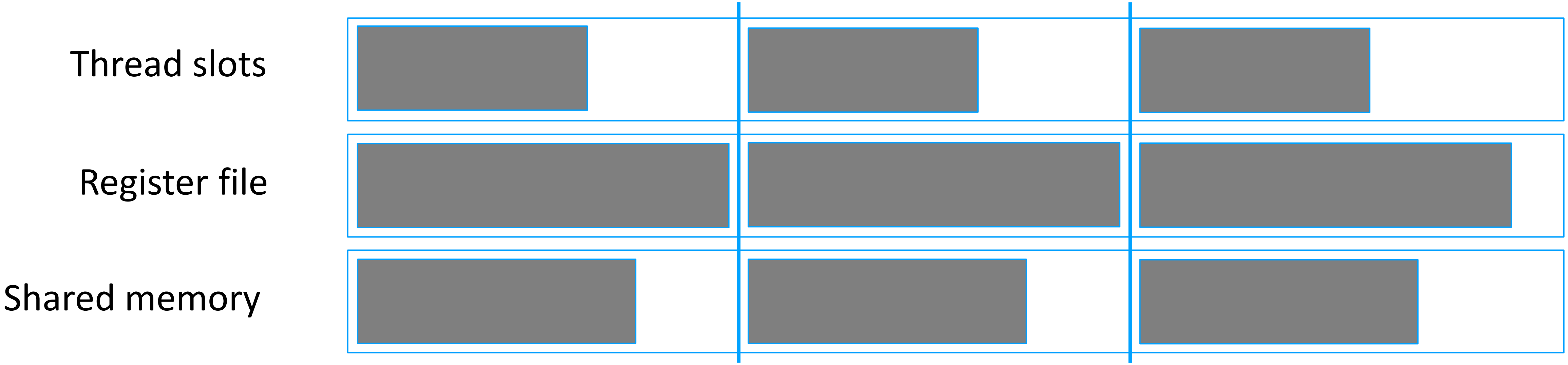
- Features specialized Tensor and RT cores
- Tensor cores can operate on 4/8/16 bit integers and 16-bit half-precision floating points
- RT cores used for Ray-Tracing in graphics workloads


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

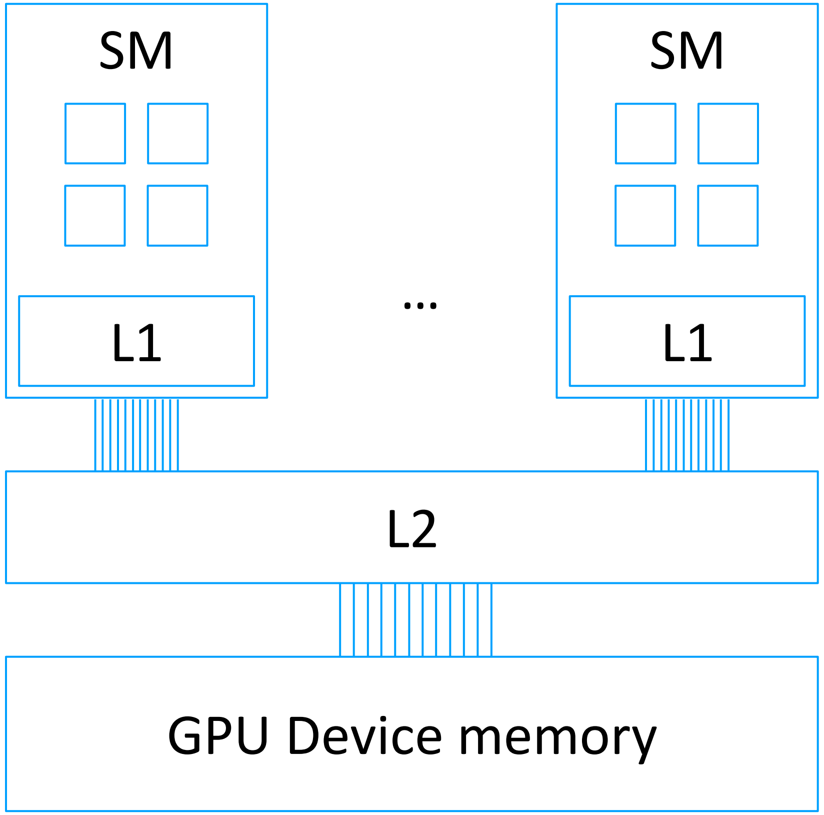


- The GPU consists of several (1 to 68) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Register file, Shared memory, Thread Slots, and Thread Block slots

- SM resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy*

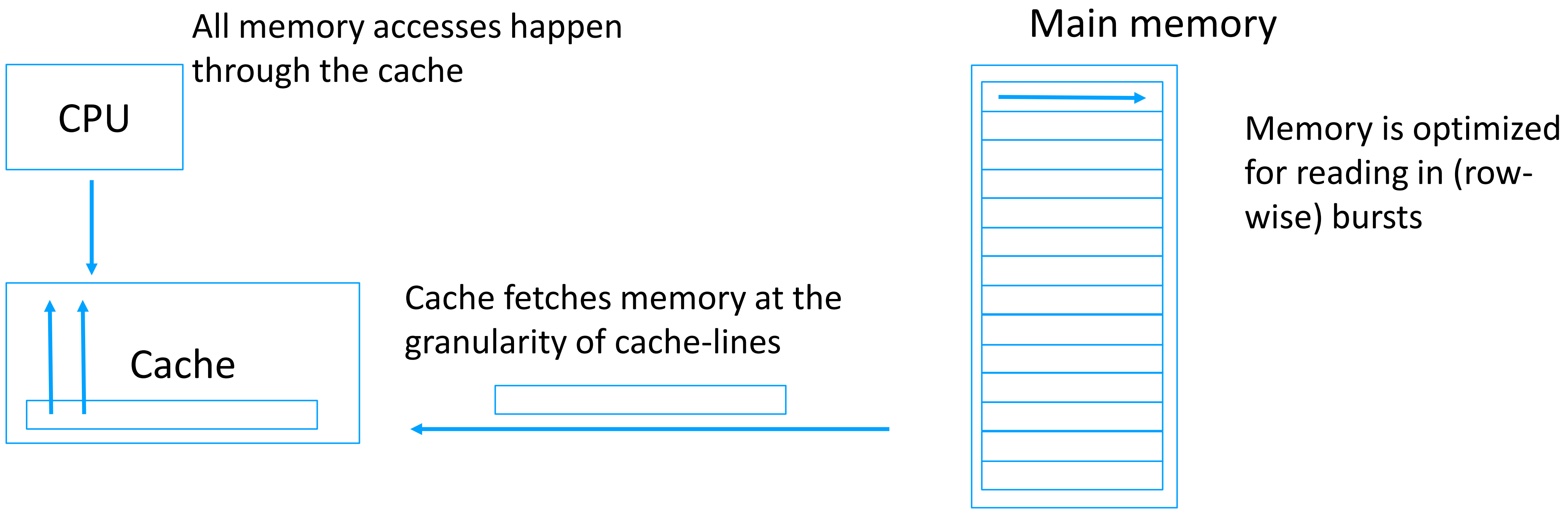


- Global memory is cached at L2, and for some GPUs also in L1
- When a thread reads a value from global memory, think about:
  - The total number of values that are accessed by the warp that the thread belongs to
  - The cache line length and the number of cache lines that those values will belong to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

Main memory

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

Cache fetches memory at the granularity of cache-lines

Subsequently accessing values that are adjacent on the same cache line is much faster than when each access requires a new cache line to be fetched

Optimizing Code

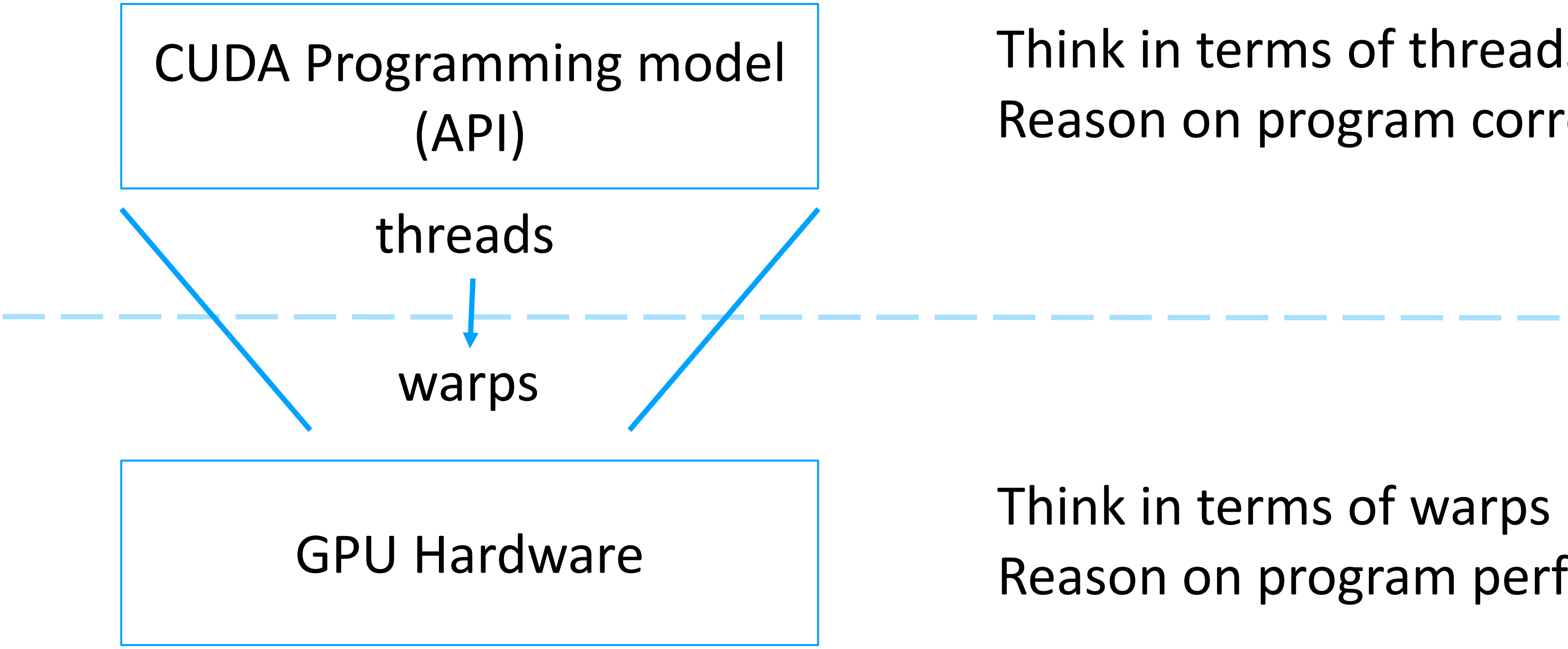
- Moving data around is more expensive than computing on it

- Start with a simple algorithm and keep it for readability and correctness checks

- Optimize only when needed
- Focus on the bottlenecks first

- Auto-tune (automatically explore the parameter space)
  - Different loop orderings
  - Different tile sizes, on multiple levels L3, L2, and L1
  - Different number of threads, thread blocks, vector lengths, etc
  - e.g. using the Kernel Tuner (https://github.com/benvanwerkhoven/kernel_tuner)

Overview



CUDA Programming model (API)

threads

warps

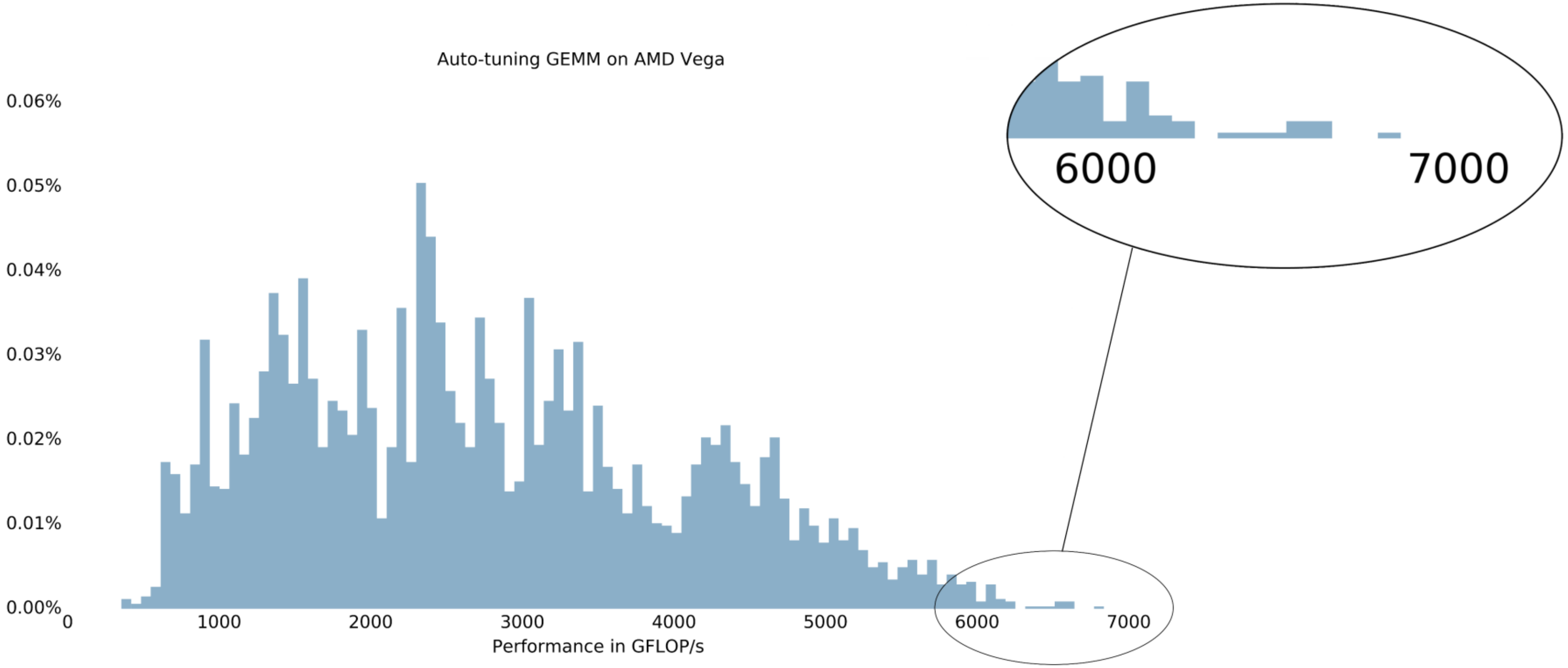
GPU Hardware

Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

- The number of combinations to try explodes rather quickly, even for single kernels, not to mention for tuning pipelines
- The best performing combination of tunable parameters will be different on different GPUs, and for different input sizes
- The best performing combination is often very hard to find!

- **Auto-tuning** is the process of automatically searching for the best performing kernel configuration



Kernel Tuner: a generic auto-tuner in Python

Easy to use:

- Can be used directly on existing kernels and code generators
- Inserts no dependencies in the kernels
- Kernels can still be compiled with regular compilers

Supports:

- Tuning functions in OpenCL, CUDA, C, and Fortran
- Large number of effective search optimizing algorithms
- Output verification for auto-tuned kernels and pipelines
- Tuning parameters in both host and device
- Python-based unit testing of GPU code
- …

https://github.com/benvanwerkhoven/kernel_tuner

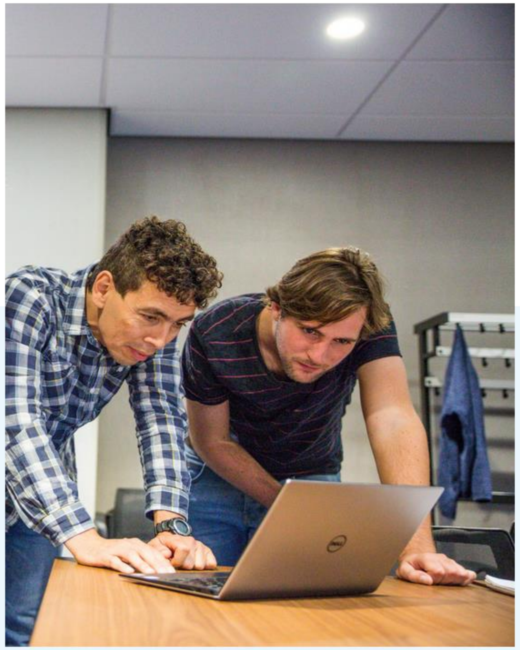
Closing

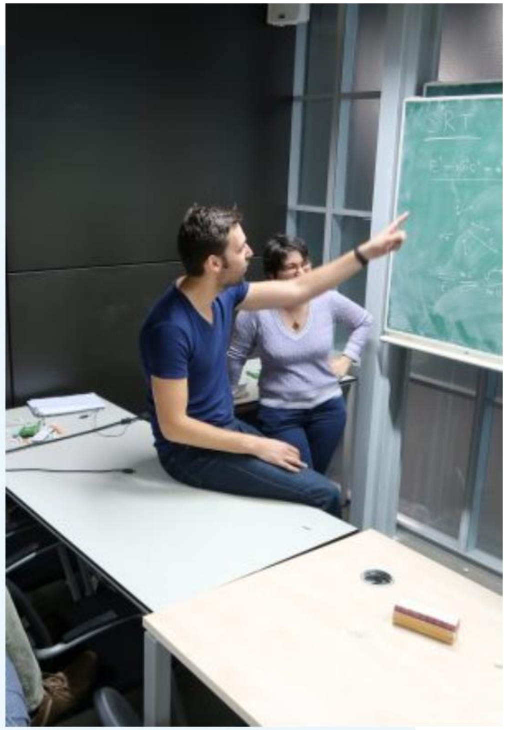
17:45 – 17:55

Collaborate with us!

We collaborate with researchers from universities and research institutes across the Netherlands on projects in every major discipline. We also participate in many EU-funded Horizon 2020 projects.

www.esciencecenter.nl



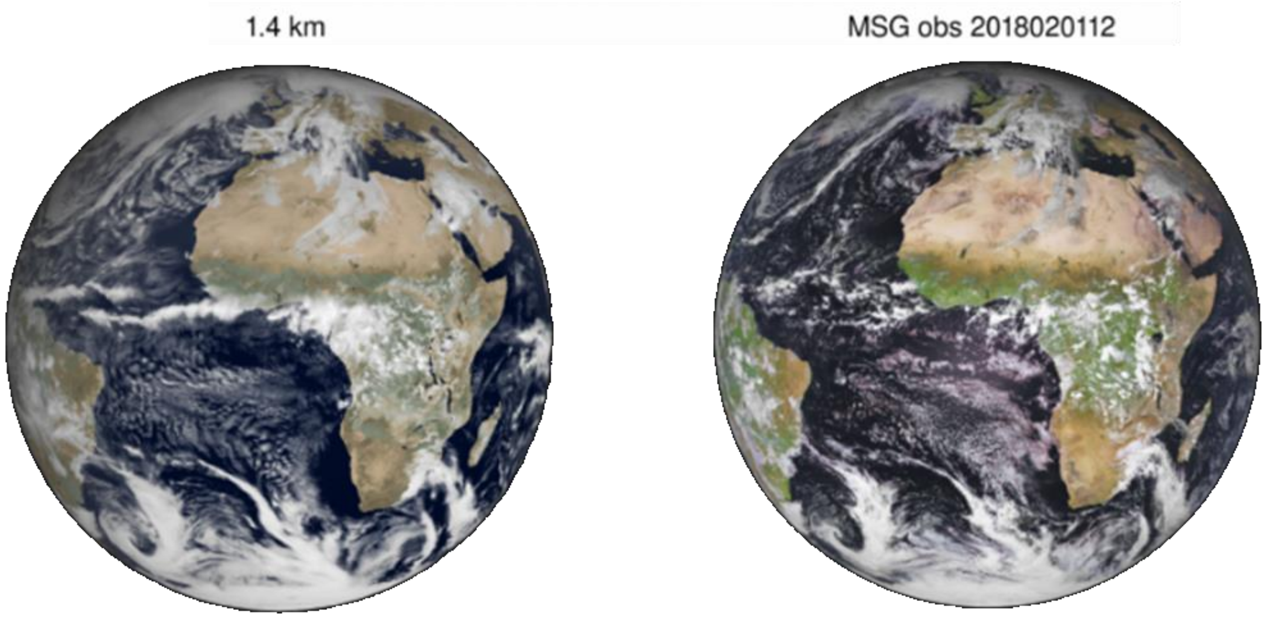


esiwace
CENTRE OF EXCELLENCE IN SIMULATION OF WEATHER
AND CLIMATE IN EUROPE

Funded by the
European Union

ESiWACE - for future exascale weather and climate simulations

Through the ESiWACE2 project we provide services on porting and performance optimization for weather and climate models



www.esiwace.eu

The projects ESiWACE and ESiWACE2 have received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreements No 675191 and 823988

In 2020 Ben, Alessio, and Willem Jan summarized the lessons they learned in ten years of developing GPU accelerated scientific applications in a paper presented at the International Conference on Computational Science (ICCS 2020):

"After years of using Graphics Processing Units (GPUs) to accelerate scientific applications in fields as varied as tomography, computer vision, climate modeling, digital forensics, geospatial databases, particle physics, radio astronomy, and localization microscopy, we noticed a number of technical, socio-technical, and non-technical challenges that Research Software Engineers (RSEs) may run into. While some of these challenges, such as managing different programming languages within a project, or having to deal with different memory spaces, are common to all software projects involving GPUs, others are more typical of scientific software projects. Among these challenges we include changing resolutions or scales, maintaining an application over time and making it sustainable, and evaluating both the obtained results and the achieved performance."

https://link.springer.com/chapter/10.1007/978-3-030-50436-6_29