Netherlands eScience Center GPU Programming course

Alessio Sclocco and Ben van Werkhoven
November 9, 2020

Morning Schedule

- 10:00 – 10:15 Course introduction
- 10:15 – 10:30 Introduction to GPU computing
- 10:30 – 10:45 Introduction to GPU programming models

- 10:45 – 10:55 Break

- 10:55 – 11:15 Introduction to GPU programming
- 11:15 – 11:35 Introduction to CUDA programming
- 11:35 – 12:00 Hands-on exercise

- 12:00 – 13:00 Lunch

Afternoon Schedule

- 13:00 – 13:15 CUDA memories 1
- 13:15 – 13:30 Hands-on exercise
- 13:30 – 13:45 CUDA memories 2
- 13:45 – 14:15 Hands-on exercise
- 14:15 – 14:45 Program execution model

- 14:45 – 15:00 Break

- 15:00 – 15:30 GPU optimizations
- 15:30 – 15:45 Hands-on exercise
- 15:45 – 16:00 Discussion

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

- 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 leave out ‘{‘ and ‘}’ to save space

Online Course Organization

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

- Questions
  - Raise your hand
  - Write in 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
  - Work on your own using the provided Jupyter notebooks
  - 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

netherlands eScience center

Introduction to GPU Computing

10:15 – 10:30

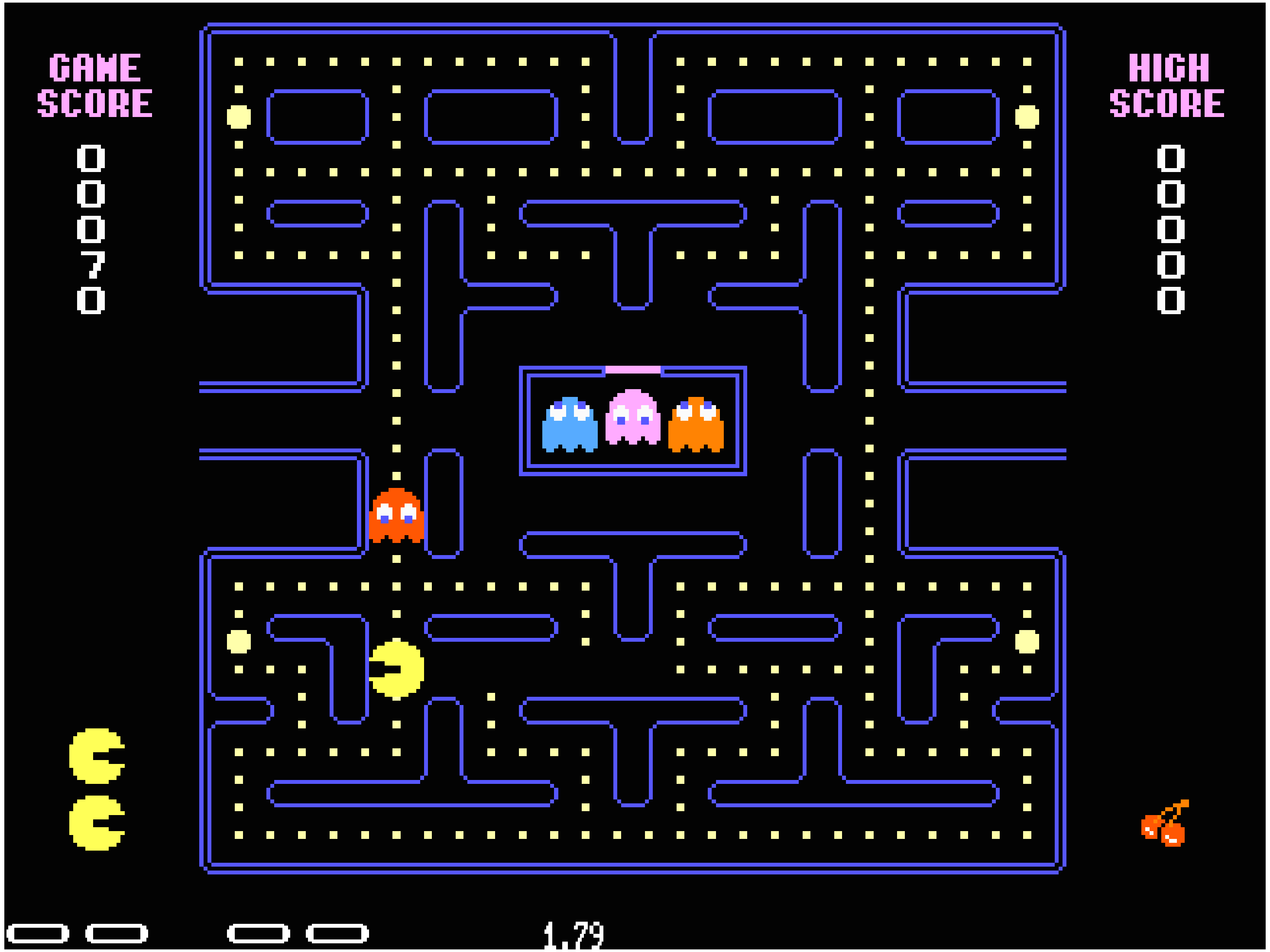
What is a GPU?

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

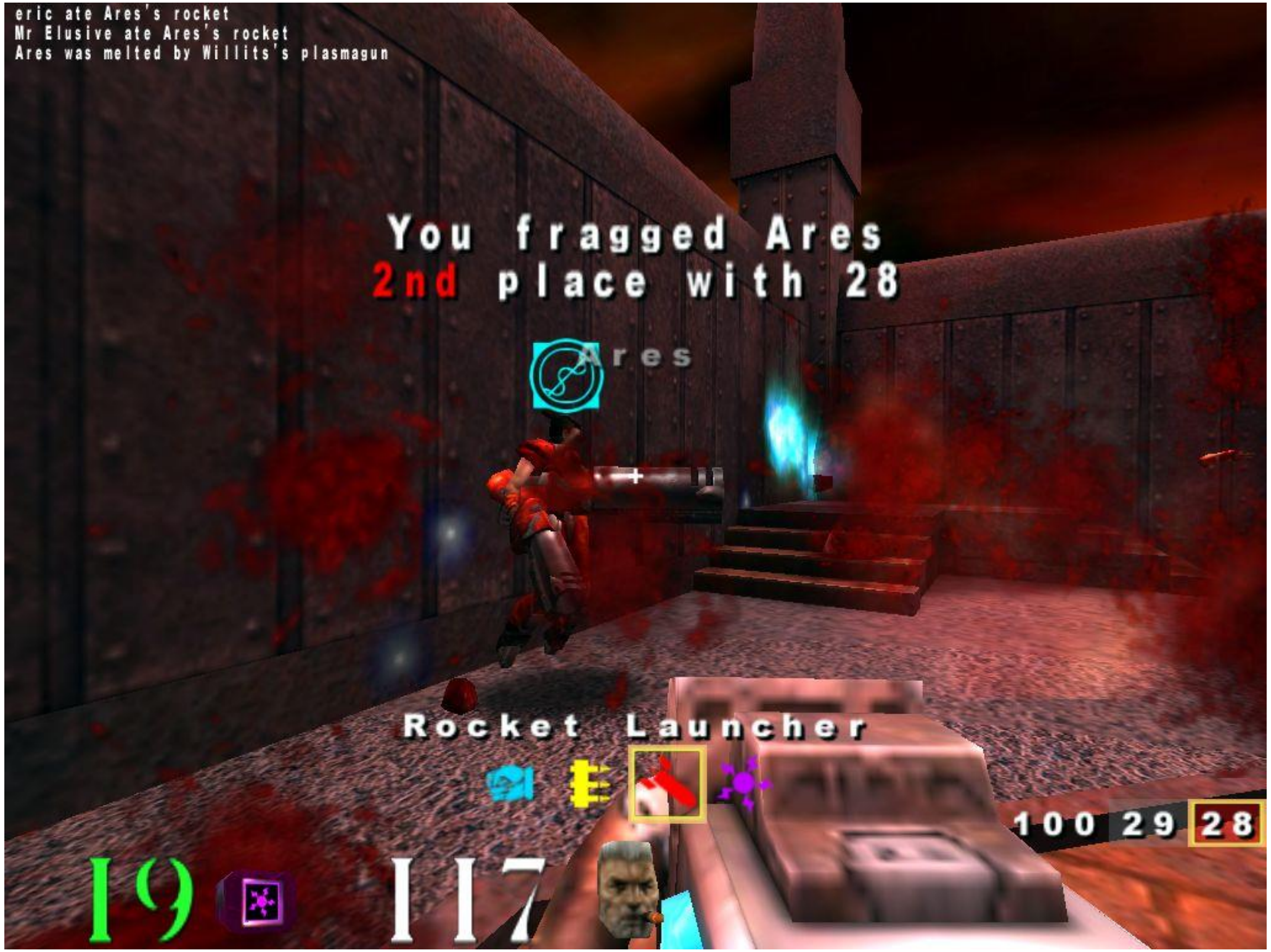
- GPGPU – General Purpose computing on GPUs

TITAN
NVIDIA

Graphics in 1980


GAME SCORE
00070
HIGH SCORE
00000
1.79

Graphics in 2000



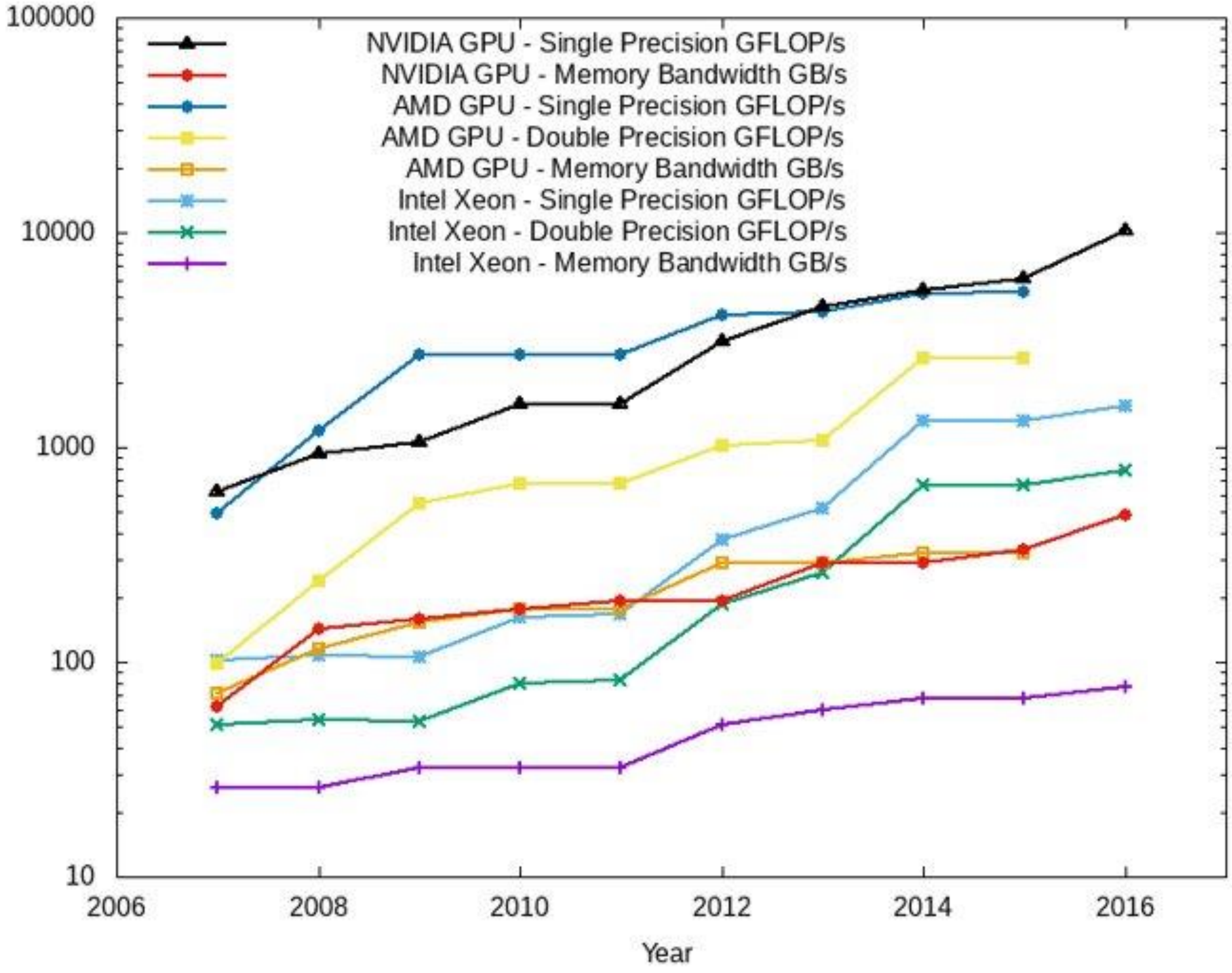
eric ate Ares's rocket
Mr Elusive ate Ares's rocket
Ares was melted by Willits's plasmagun

You fragged Ares
2nd place with 28

Ares
Rocket Launcher
100 29 28
19
117

Graphics Today

Performance Comparison: GPUs vs CPUs



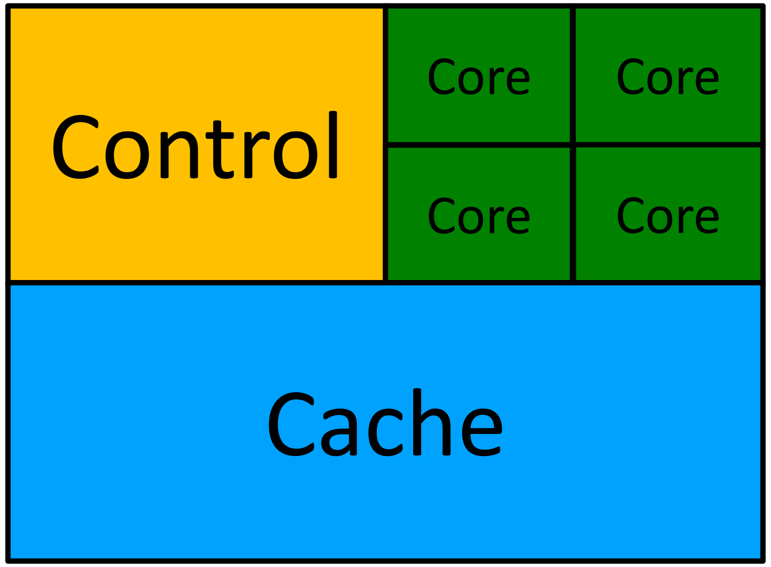
Oak Ridge's Summit: the World Fastest GPU System

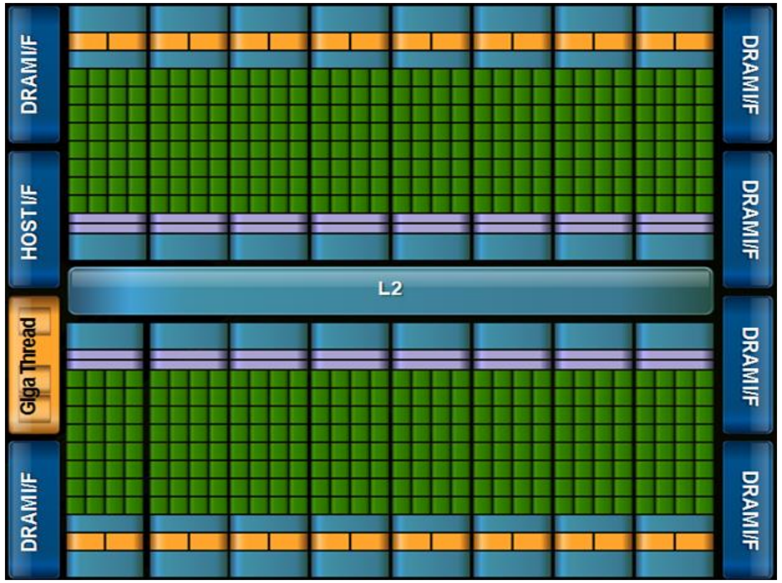
- Number 2 in TOP500 list (June 2020)
  - 200 PFLOP/s peak
  - 4,608 nodes
  - 13 MW power consumption
  - 9,216 CPUs
    - IBM Power 9
    - 2 CPUs per node
  - 27,648 GPUs
    - NVIDIA Volta V100
    - 6 GPUs per node
  - 2,397,824 cores

  - Six systems in top 10 using GPUs

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


Control
Core
Core
Core
Core
Cache

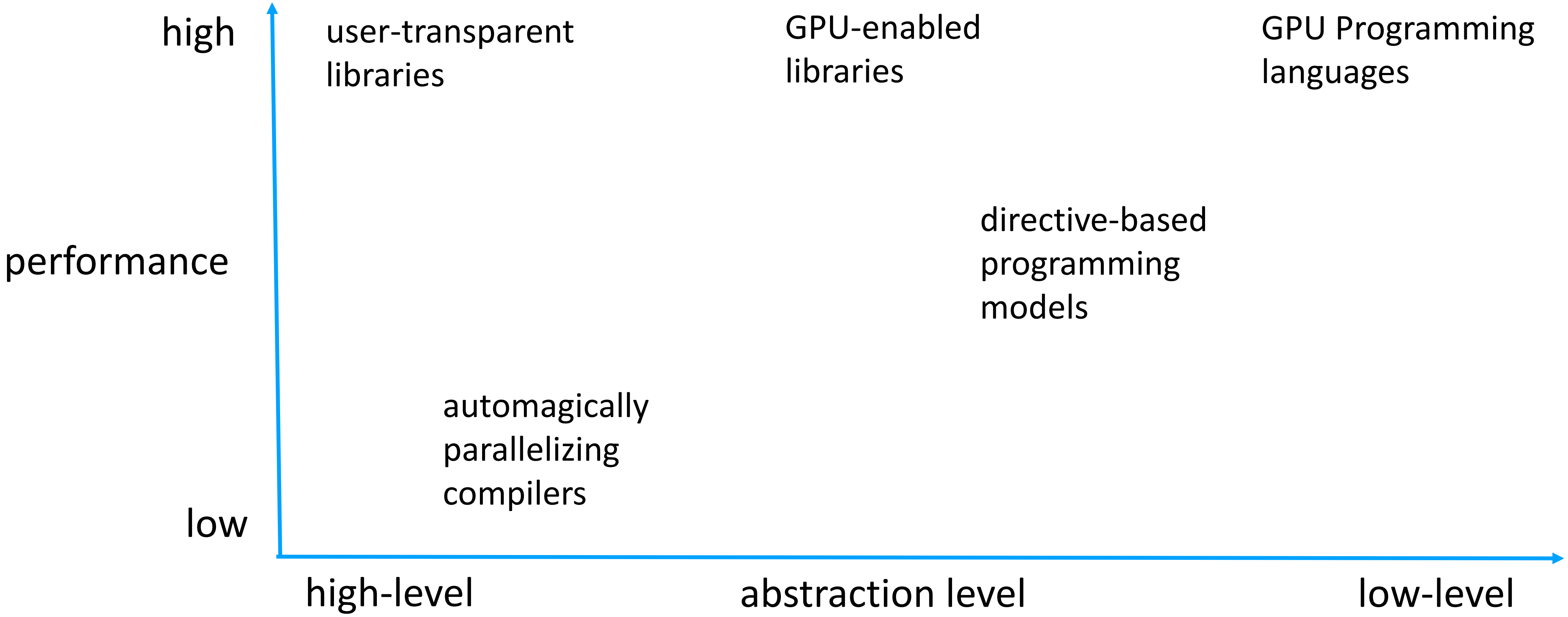

DRAM I/F
HOST I/F
Giga Thread
DRAM I/F
L2
DRAM I/F
DRAM I/F
DRAM I/F
DRAM I/F

netherlands eScience center

Introduction to GPU Programming Models

10:30 – 10:45

GPU Programming Models



high

performance

low

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

directive-based programming models

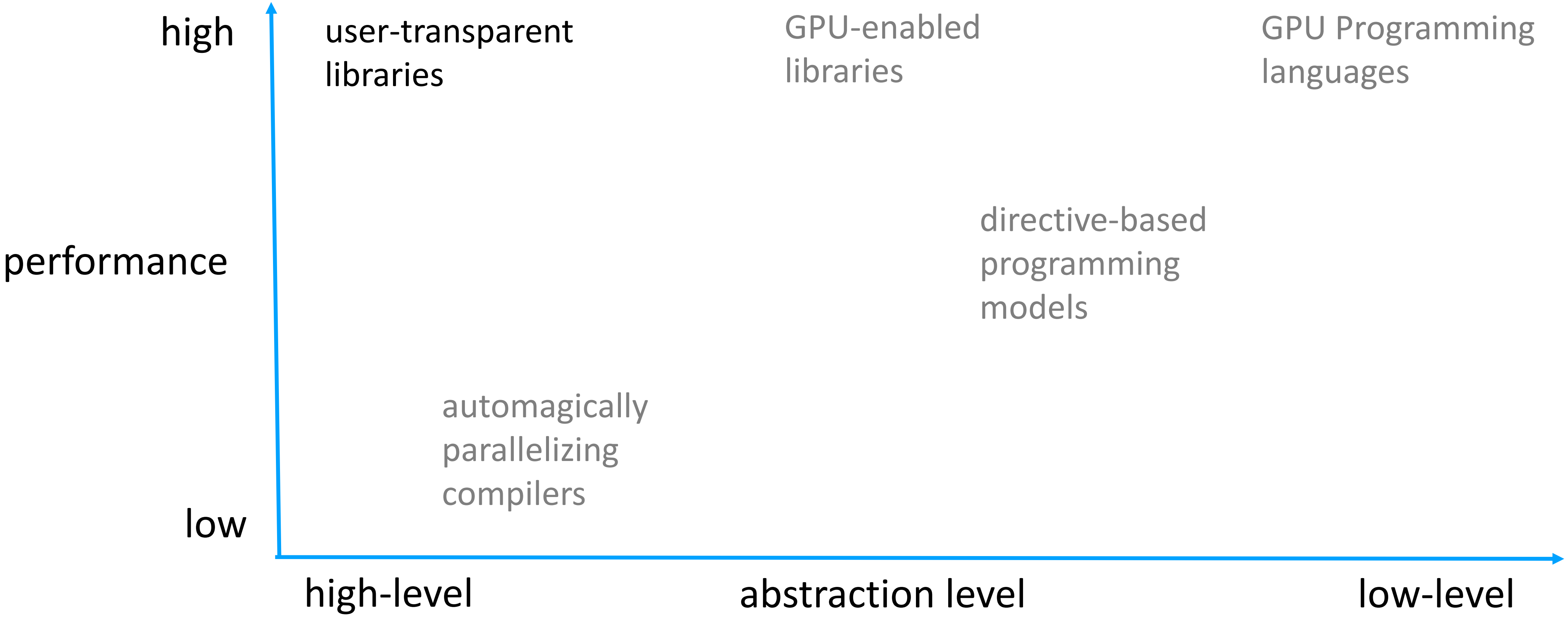
automagically parallelizing compilers

high-level

abstraction level

low-level

GPU Programming Models



high

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

performance

directive-based programming models

automagically parallelizing compilers

low

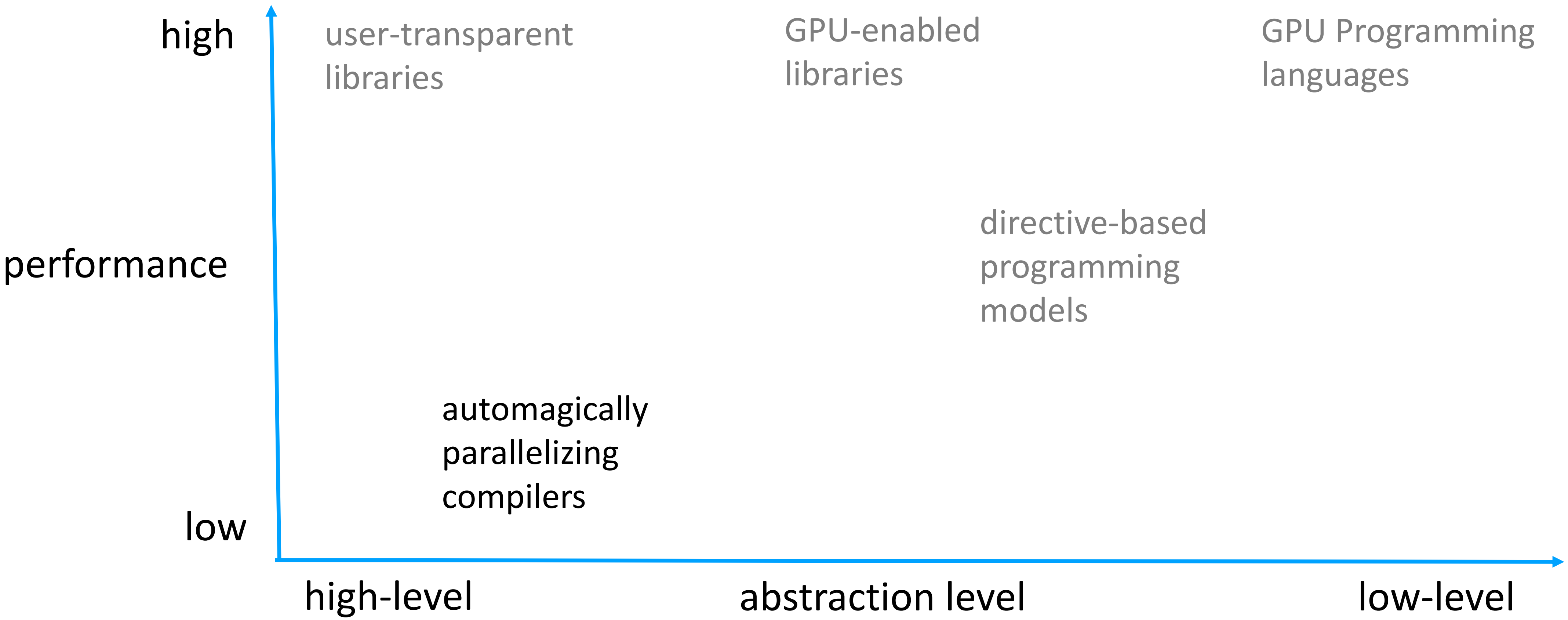
high-level

abstraction level

low-level

- Act as drop-in replacements for CPU libraries
- There aren’t that many, and their application is often limited

- Difficult to build:
  - Library must maintain state, any init() or destroy() methods already break transparency
  - Library designer has to decide how to manage GPU memory
- Difficult to use:
  - Optimizing application performance is hard when you don’t know what happens inside the library



high

performance

low

high-level

abstraction level

low-level

user-transparent
libraries

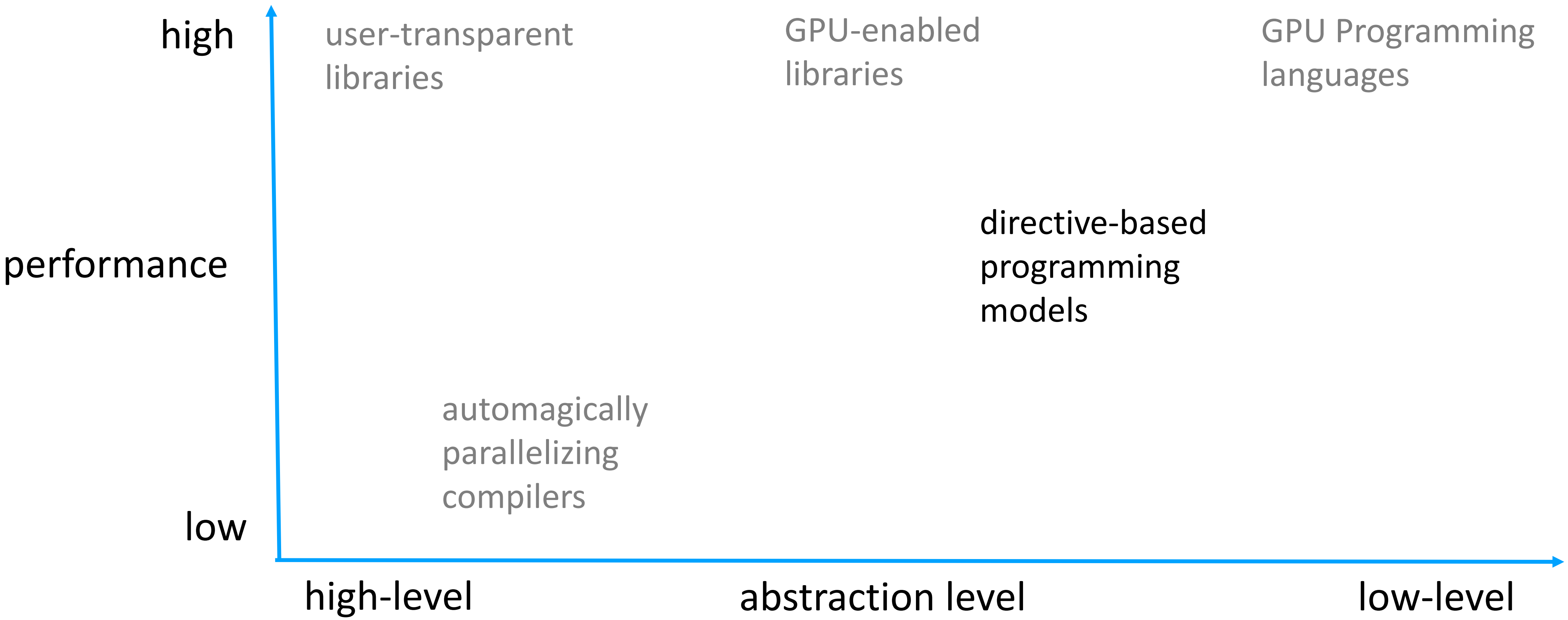
GPU-enabled
libraries

GPU Programming
languages

directive-based
programming
models

automagically
parallelizing
compilers

GPU Programming Models



high

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

performance

directive-based programming models

automagically parallelizing compilers

low

high-level

abstraction level

low-level

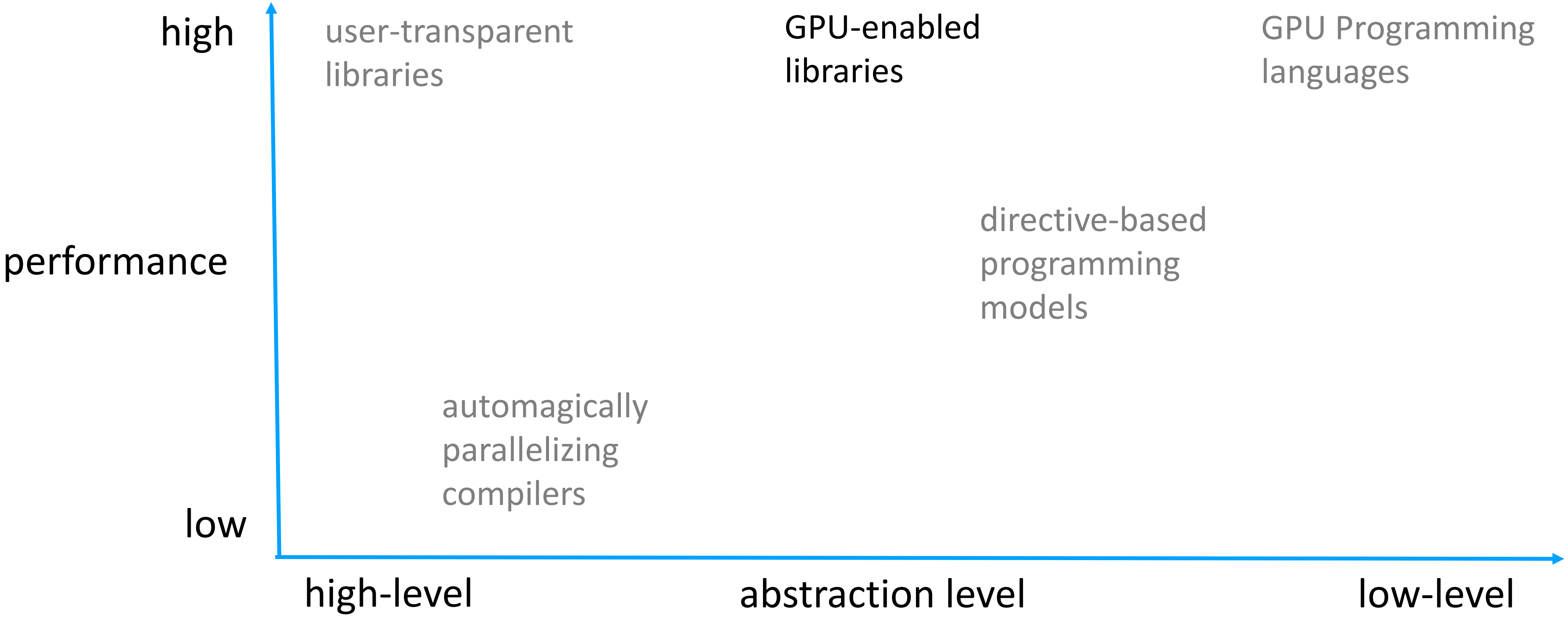
OpenACC and OpenMP:

- Open standards for *directives* that can be implemented by compilers

- Directives are language constructs that specify how compilers should process their input

- What does a directive look like?
  - In C: `#pragma acc` *directive-name [clauses]*
  - In Fortran: `!$acc` *directive-name [clauses]*
- Example:
  - `#pragma acc parallel`
    Tells the compiler that the following structured block should be executed in parallel on the current accelerator device

GPU Programming Models



high

performance

low

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

directive-based programming models

automagically parallelizing compilers

high-level

abstraction level

low-level

- User is responsible for managing GPU memory
- Often use specialized objects that represents data in GPU memory
- Easy access to highly-optimized and auto-tuned GPU routines
- Either focused on specific functionality or offering a ‘GPU Array’-like datatype

- Examples of function oriented libraries:

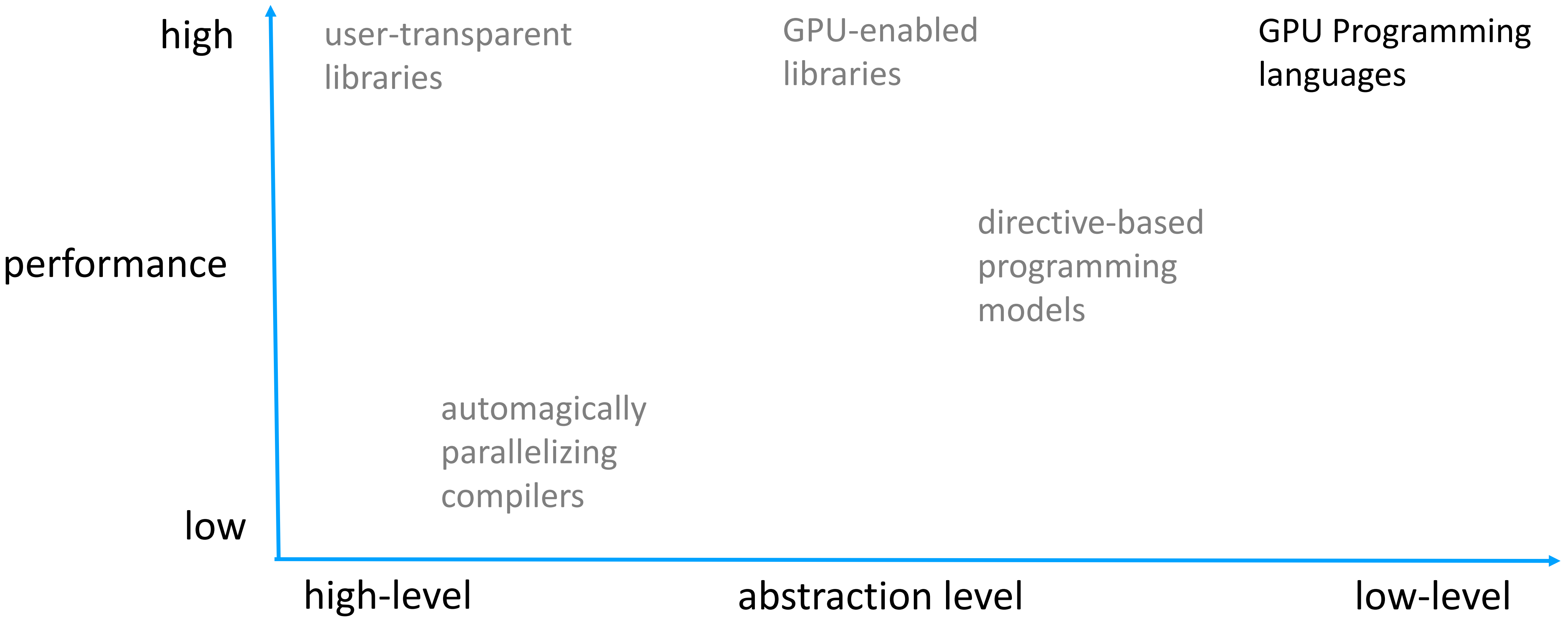
Nvidia	OpenCL
cuFFT	clFFT
cuBLAS	clBlast
cuRAND	
cuSparse	
cuDNN	

Examples of array-like libraries:

Name	languages
gpuArray	Matlab
GPUArray	python
arrayFire	C, Python, Rust

Language bindings for languages other than C/C++ can be a bit more difficult to find

GPU Programming Models



high

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

performance

directive-based programming models

automagically parallelizing compilers

low

high-level

abstraction level

low-level

Break

10:45 – 10:55

Introduction to GPU Programming

10:55 – 11:15

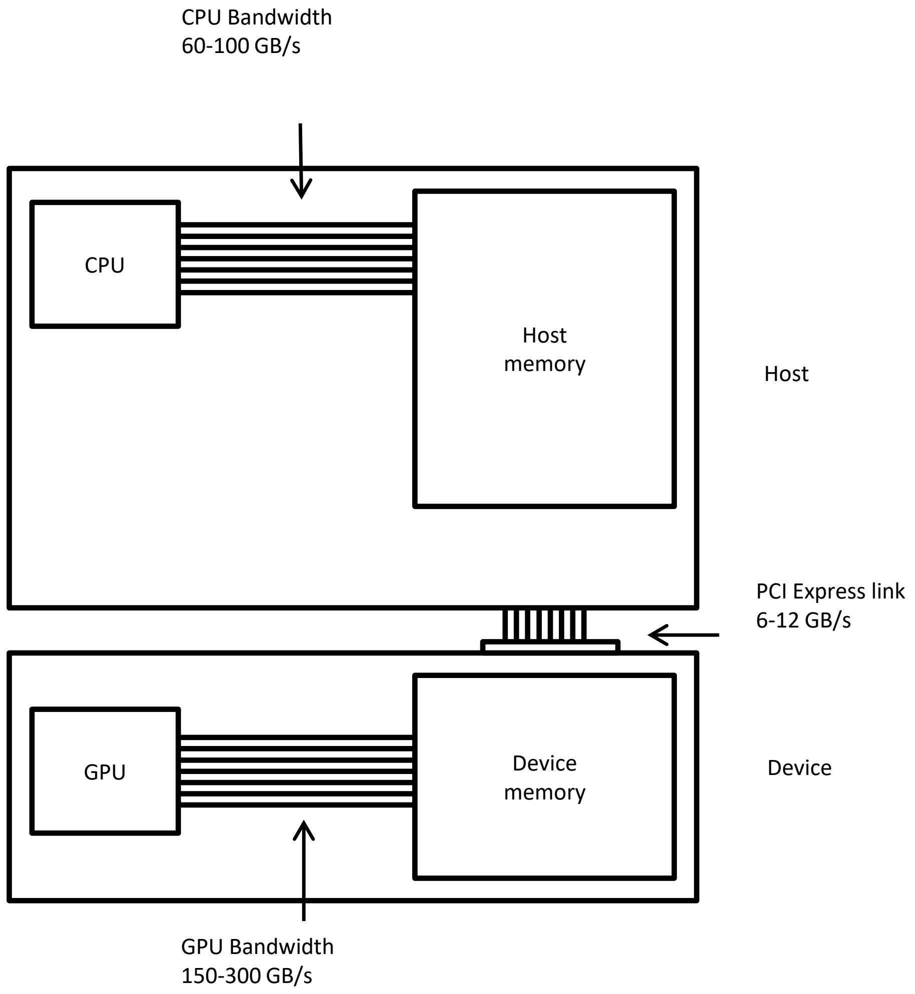
Why is GPU Programming Different?

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:

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


CPU Bandwidth
60-100 GB/s
CPU
Host memory
Host
PCI Express link
6-12 GB/s
GPU
Device memory
Device
GPU Bandwidth
150-300 GB/s

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)

- 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 really just finding the best performing combination for all of the parameters

- Auto-tuners are used to automate the search process

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

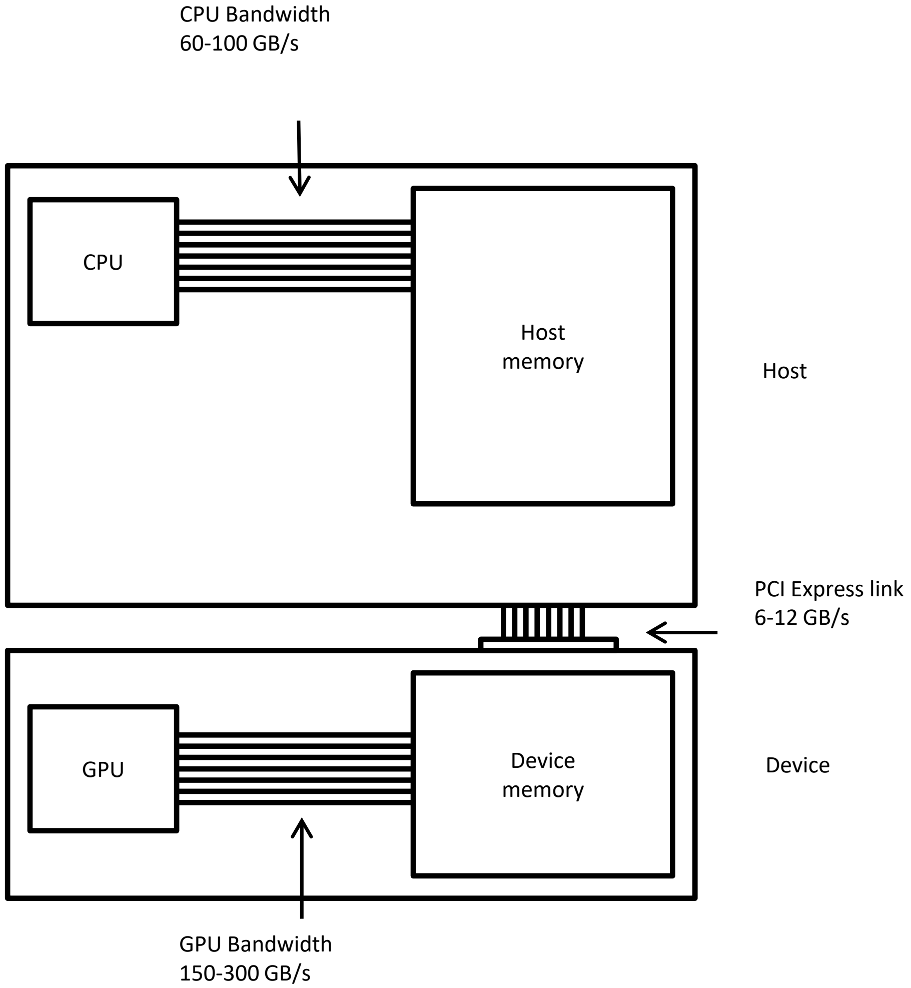
- Transferring data to the GPU is expensive (because of the relatively low PCIe bandwidth, better with NVLink)

- In general it’s best to keep working on transferred data for as long as possible

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

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. Have to deal with separate memory space, limited bandwidth between host and device memory


CPU Bandwidth
60-100 GB/s
CPU
Host
memory
Host
PCI Express link
6-12 GB/s
GPU
Device
memory
Device
GPU Bandwidth
150-300 GB/s

Introduction to CUDA Programming

11:15 – 11:35

Before we start:

- I’m going to explain the CUDA Programming model
- I’ll 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
- 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.

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

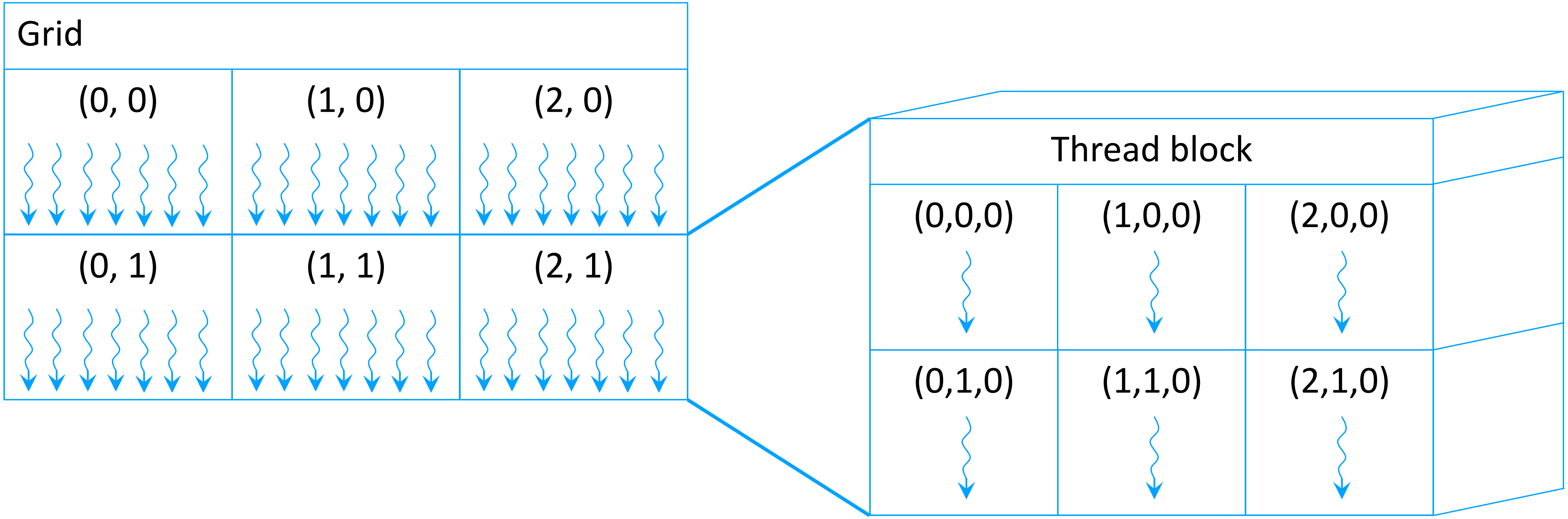
The device part:

- 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

- 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



Grid

(0, 0) (1, 0) (2, 0)

(0, 1) (1, 1) (2, 1)

Thread block

(0,0,0) (1,0,0) (2,0,0)

(0,1,0) (1,1,0) (2,1,0)

- 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`

- 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();
```

Hands-on Exercise

11:35 – 12:00

Setup hands-on session

- Login on DAS-5 (fs0.das5.cs.vu.nl)
- Execute (recommended to add both to your .bashrc):
  - `module load cuda10.1`
  - `alias gpurun="srun -N 1 -C TitanX --gres=gpu:1"`

- Installing Python 3
  - Follow these steps:
    - `wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh`
    - `bash Miniconda3-latest-Linux-x86_64.sh` (allow to add to .bashrc)
    - `export PATH="$HOME/miniconda3/bin:$PATH"`
  - `pip install numpy jupyter`
  - `pip install pycuda` (make sure you've typed `module load cuda10.1` first)

- Follow the guide to use Jupyter notebooks on DAS-5
  - https://guide.esciencecenter.nl/#/nlesc_specific/e-infrastructure/das5

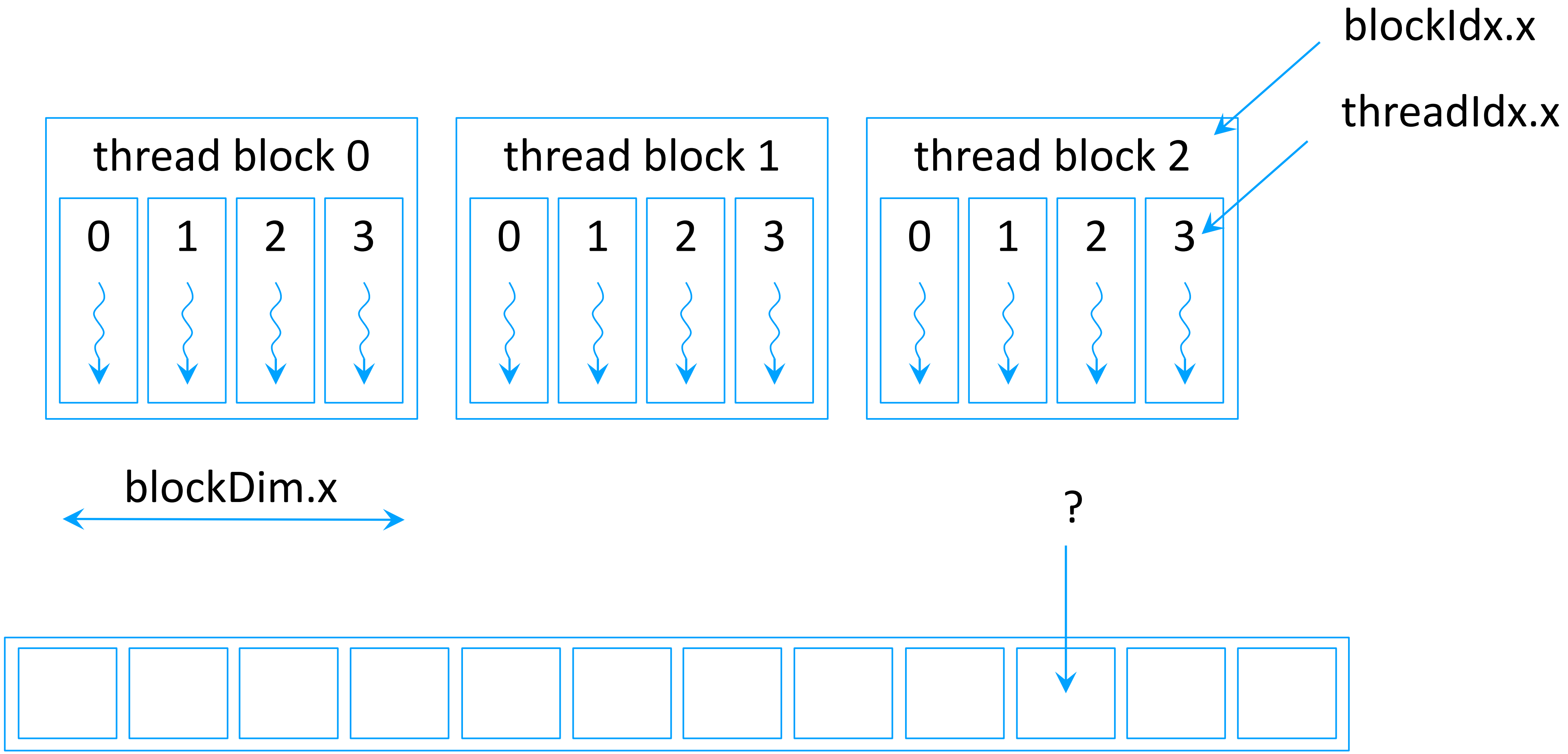
First Hands-on Session: Vector Add

- Select the notebook for the first exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/vector_add.ipynb

- 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
blockDim.x
?

Lunch

12:00 – 13:00

netherlands eScience center

CUDA Memories

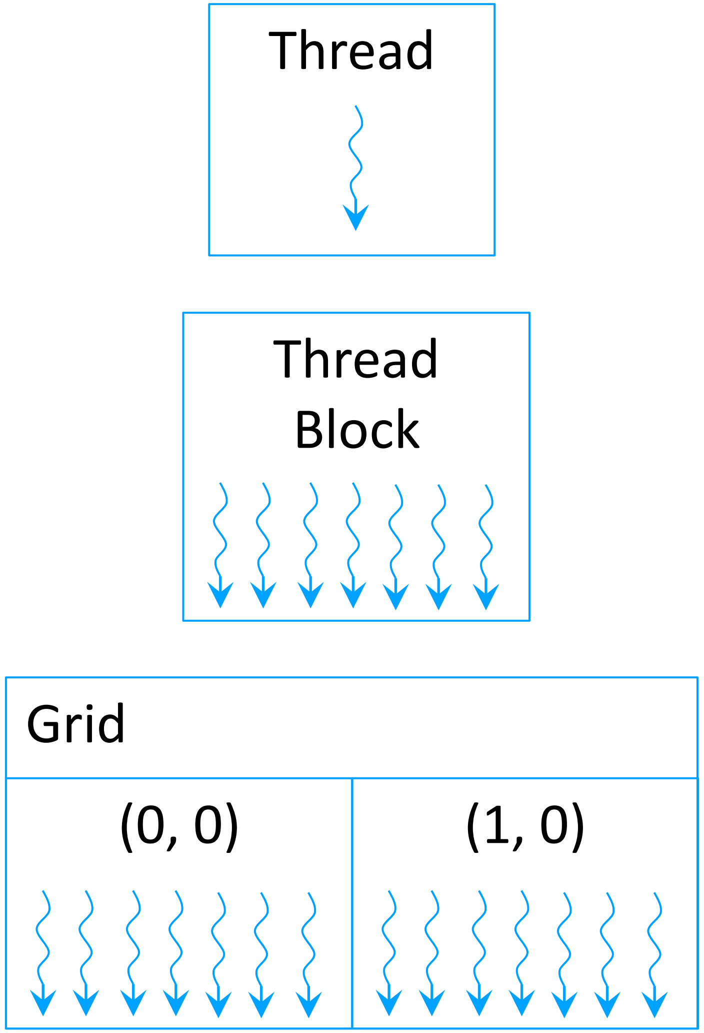
13:00 – 13:15

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 `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`

Hands-on Exercise

13:15 – 13:30

- Select the notebook for the second exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/pnpoly.ipynb

- 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

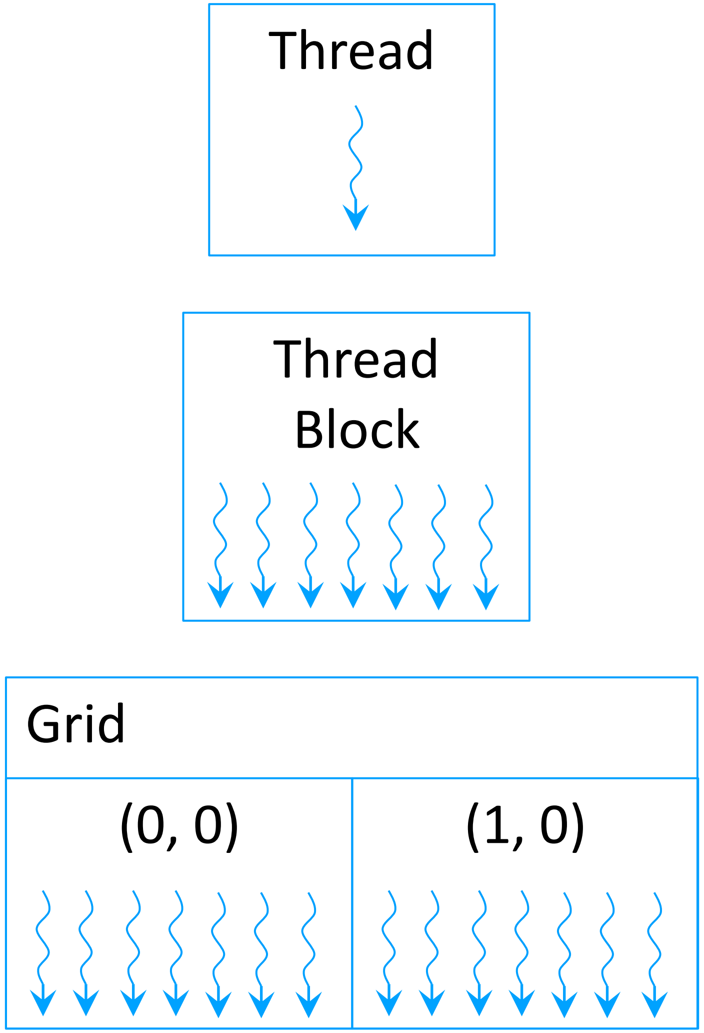
13:30 – 13:45

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

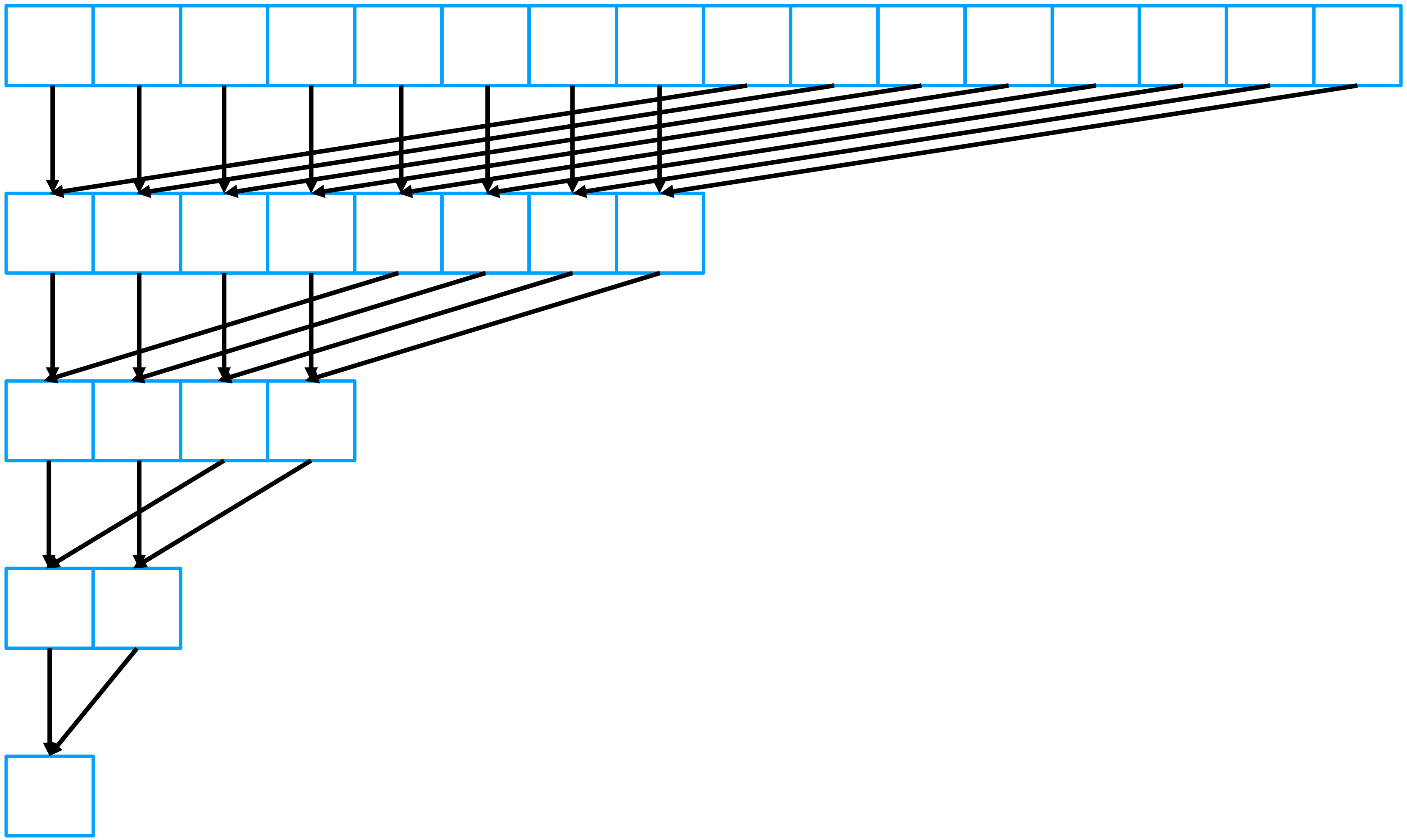
13:45 – 14:15

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

- 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

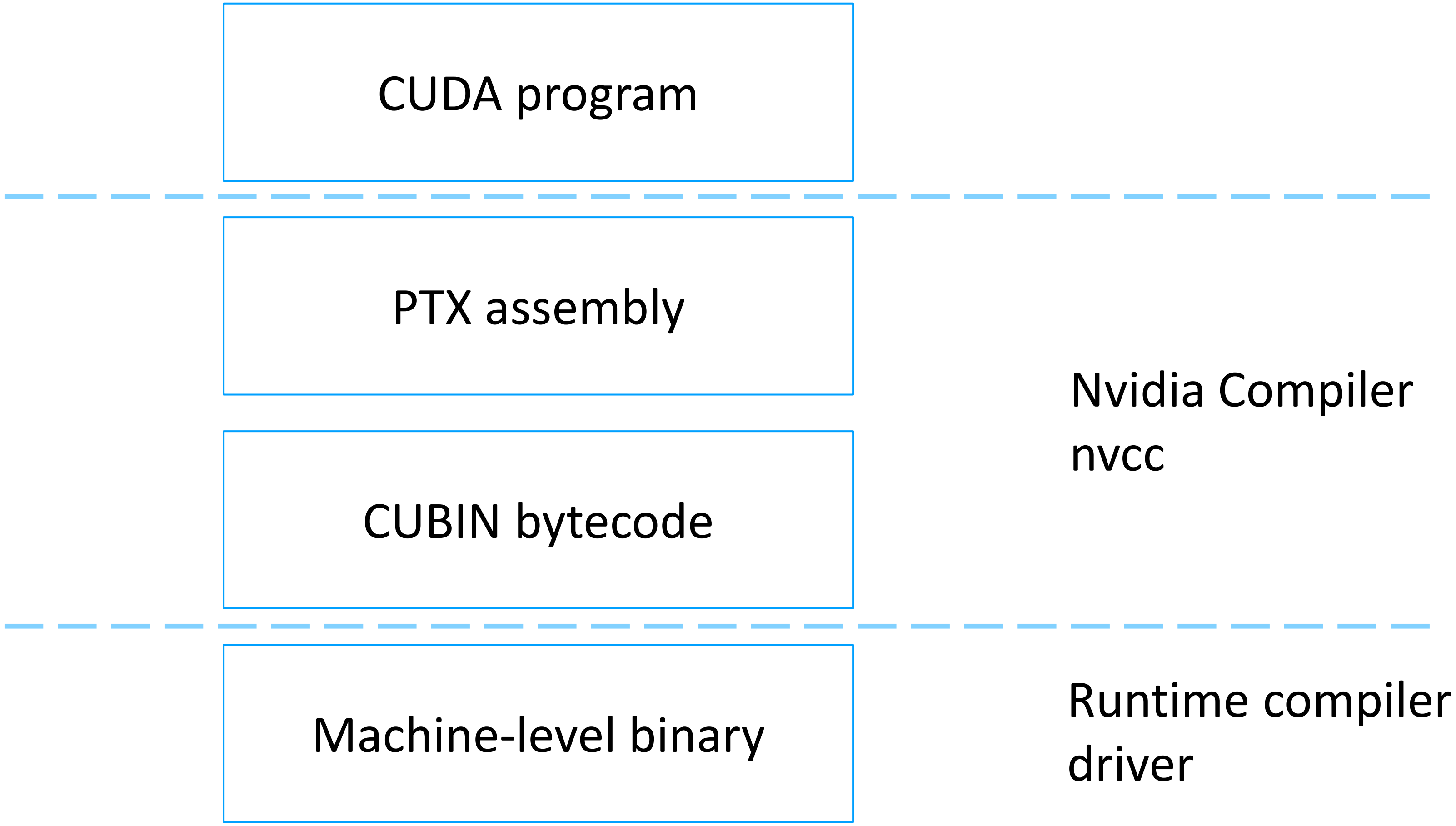
Hint – Parallel Reduction



Program Execution Model

14:15 – 14:45

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**

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*
- 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)
```

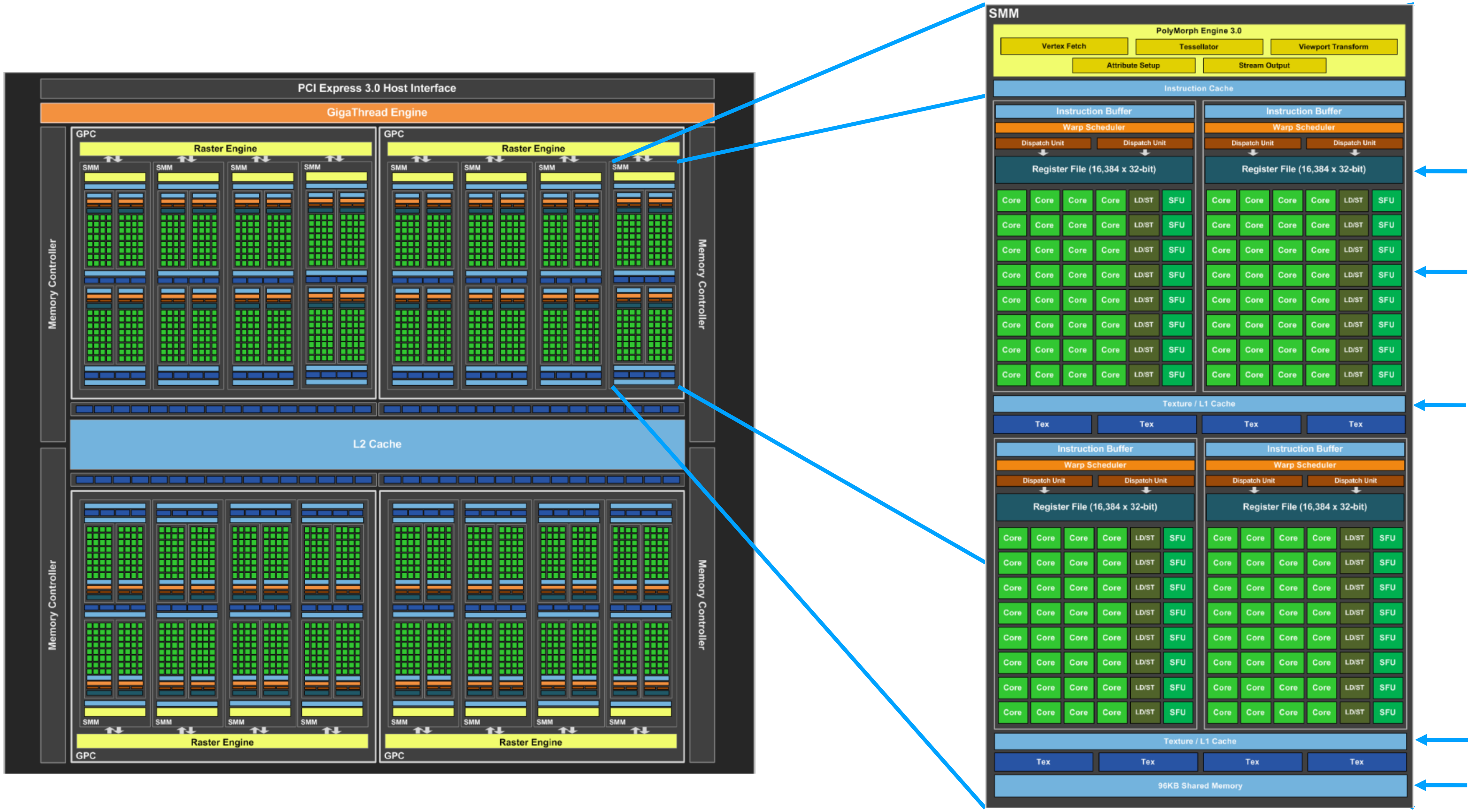
  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

- 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


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
Raster Engine
SMM
Memory Controller
L2 Cache
SMM
PolyMorph Engine 3.0
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory

Register file

block of 32 cores

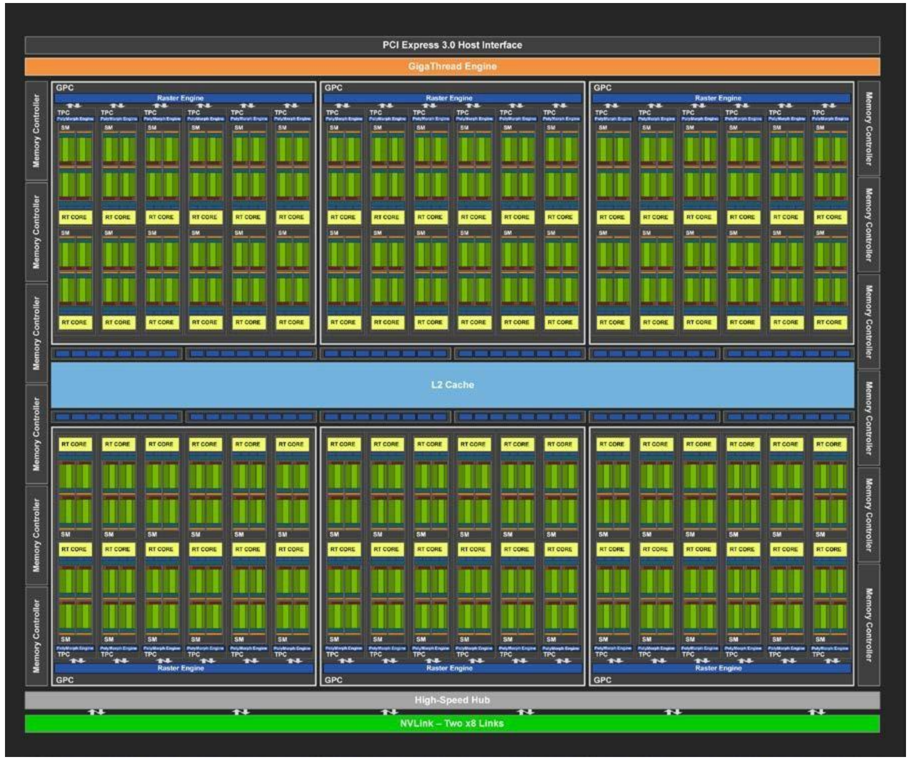
L1 Cache

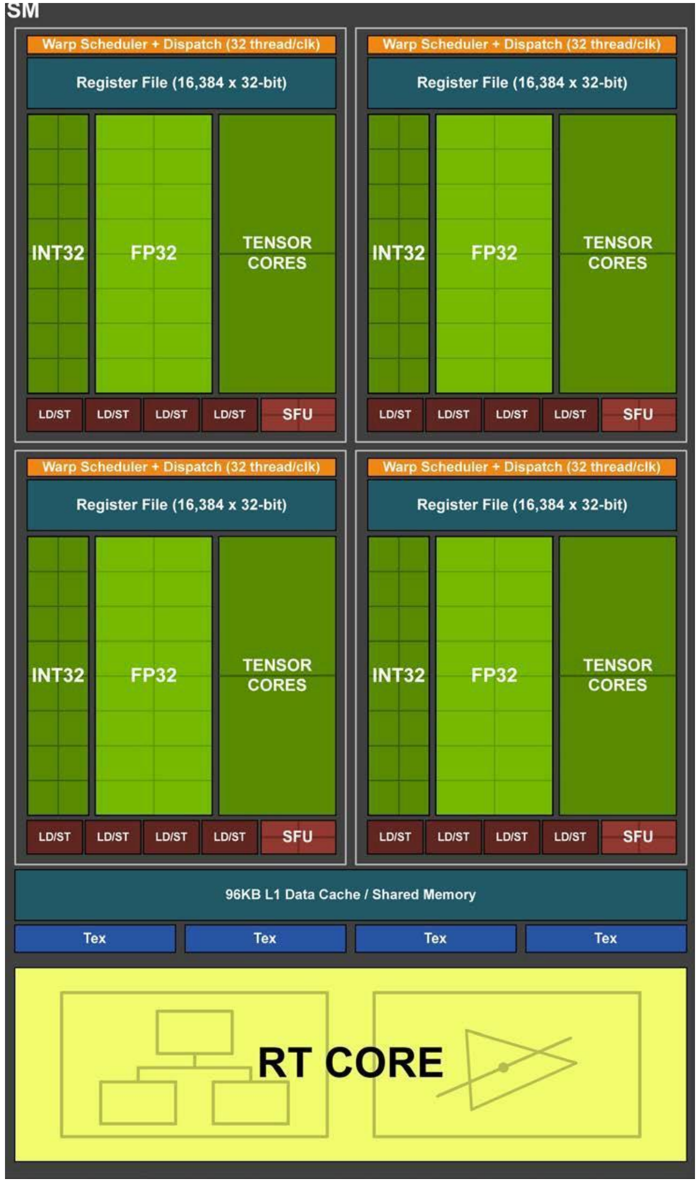
L1 Cache

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

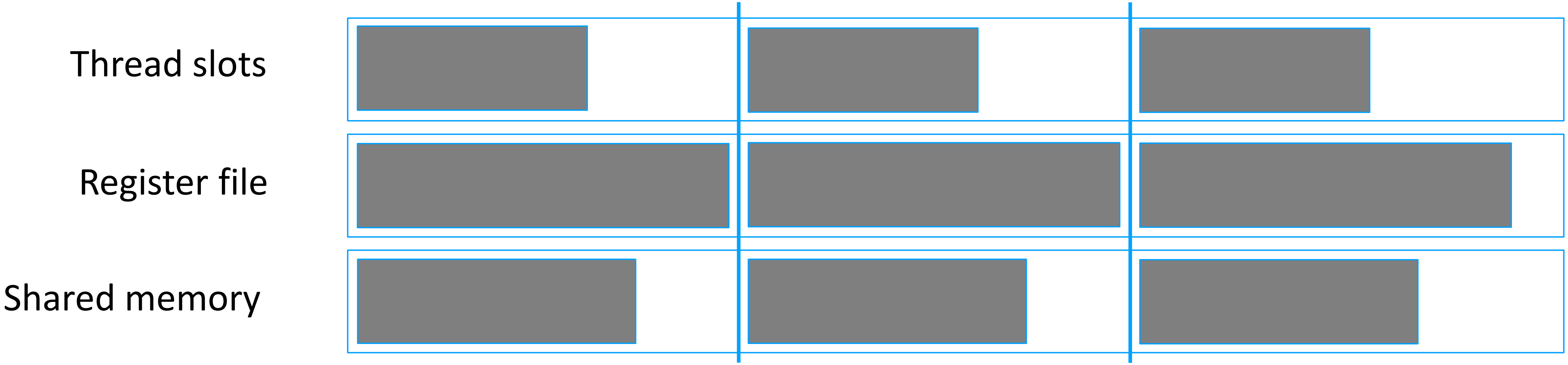

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
Raster Engine
TPC
SM
RT CORE
L2 Cache
Memory Controller
High-Speed Hub
NVLink – Two x8 Links


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

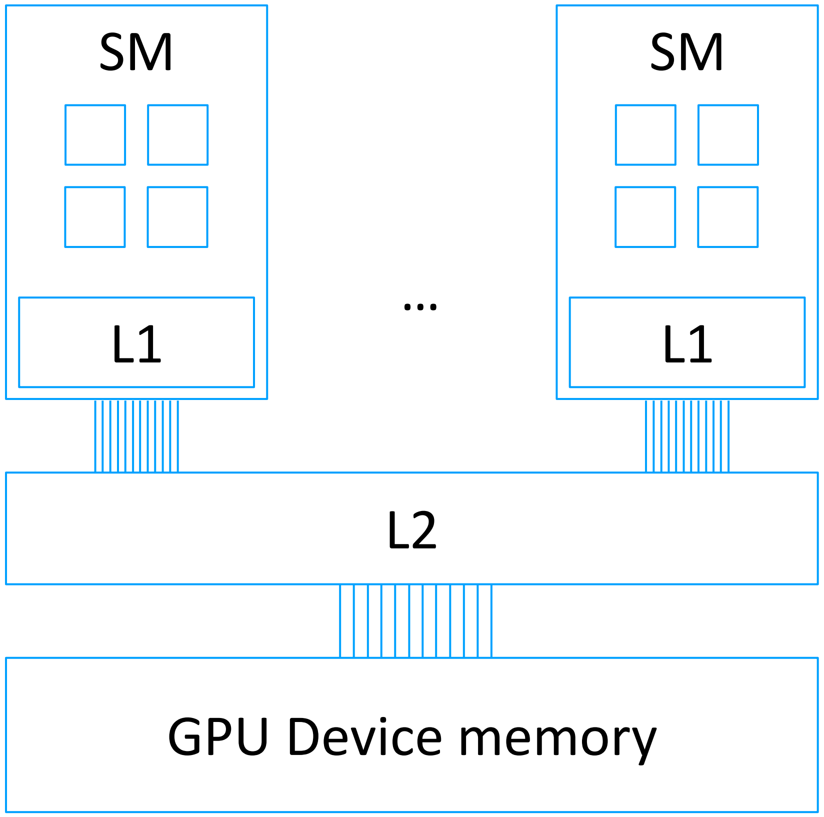
Resource Partitioning

- 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*

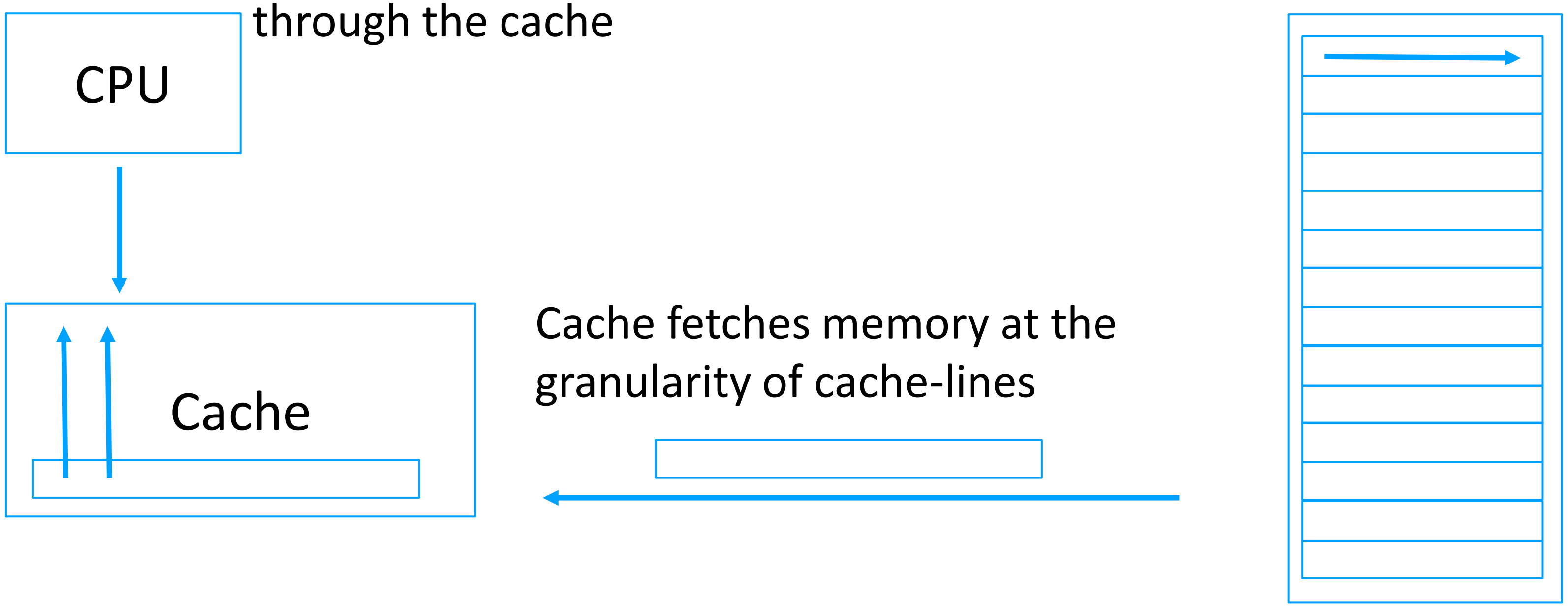

Thread slots
Register file
Shared memory

- 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



CPU

All memory accesses happen through the cache

Cache

Cache fetches memory at the granularity of cache-lines

Main memory

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

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

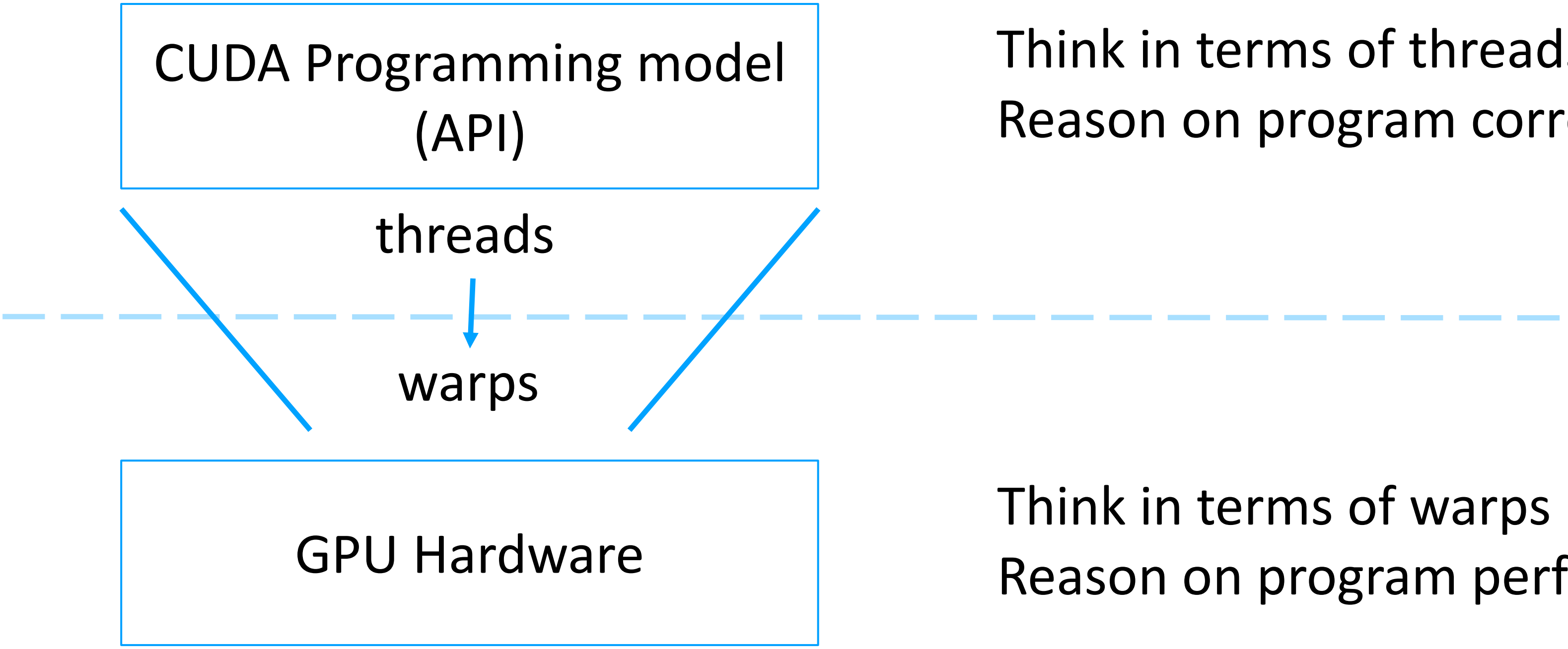
- Moving data around is more expensive than computing on it

- 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

Break

14:45 – 15:00

netherlands eScience center

GPU Optimizations

15:00 – 15:30

Optimizing Code (intentionally repeated slide)

- 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

Optimization

- Algorithmic Optimization
  - Cleverly rewriting your algorithm to reduce the total amount of computations
  - Reducing the complexity in terms of ‘Big-O notation’

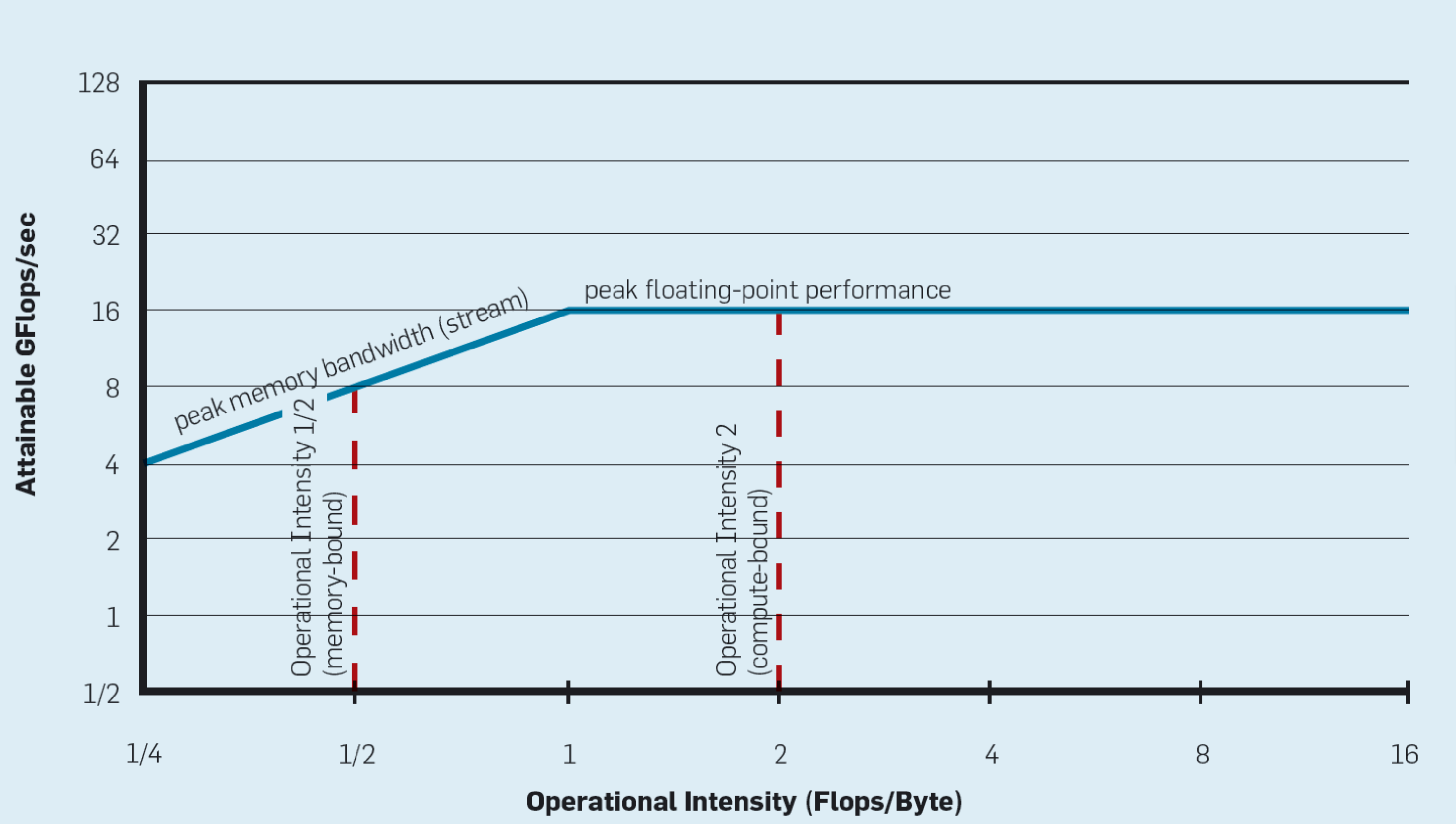
- Execution Optimization
  - Increasing hardware utilization without doing double work
  - Optimizing the code such that it maps efficiently onto the hardware
  - Rewriting the code such that it will run faster on the hardware

Arithmetic Intensity and Operational Intensity

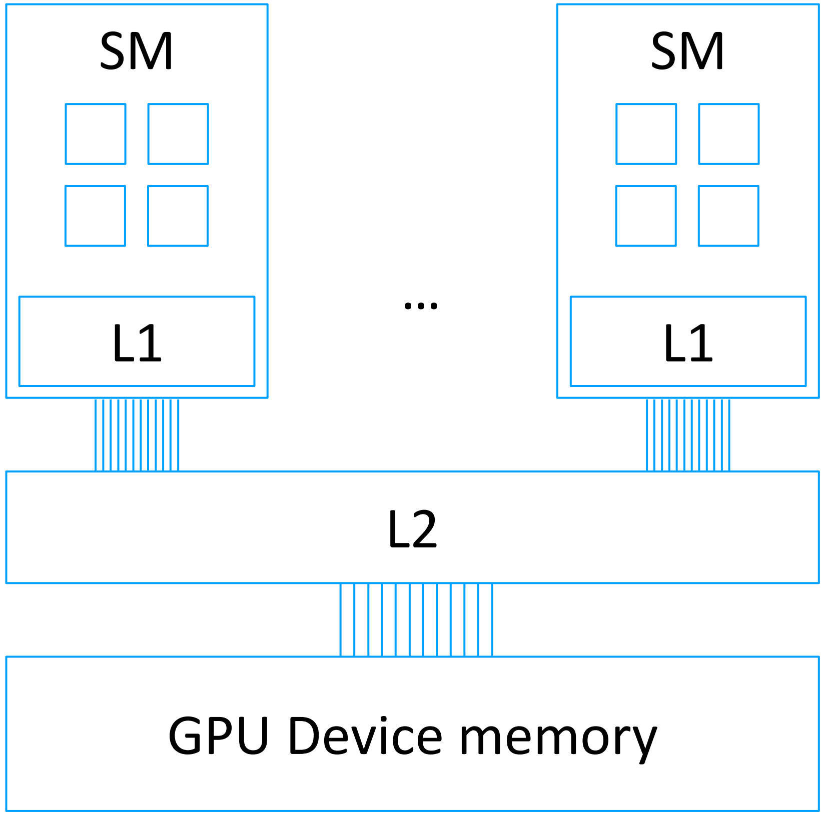
- Arithmetic Intensity is the ratio between compute operations and memory operations
- $$\text{Arithmetic Intensity} = \frac{\text{total operations}}{\text{total bytes required}}$$
- $$\text{Operational Intensity} = \frac{\text{total operations}}{\text{total bytes actually loaded from memory}}$$
- Actual bytes loaded from memory may be different, because memory loads are performed through the cache-hierarchy

- The roofline model tells us if our kernel is compute-bound or memory-bandwidth bound

- Attainable performance (e.g. in GFLOP/s) =
  min(theoretical peak compute performance,
  theoretical peak memory bandwidth x operational intensity)

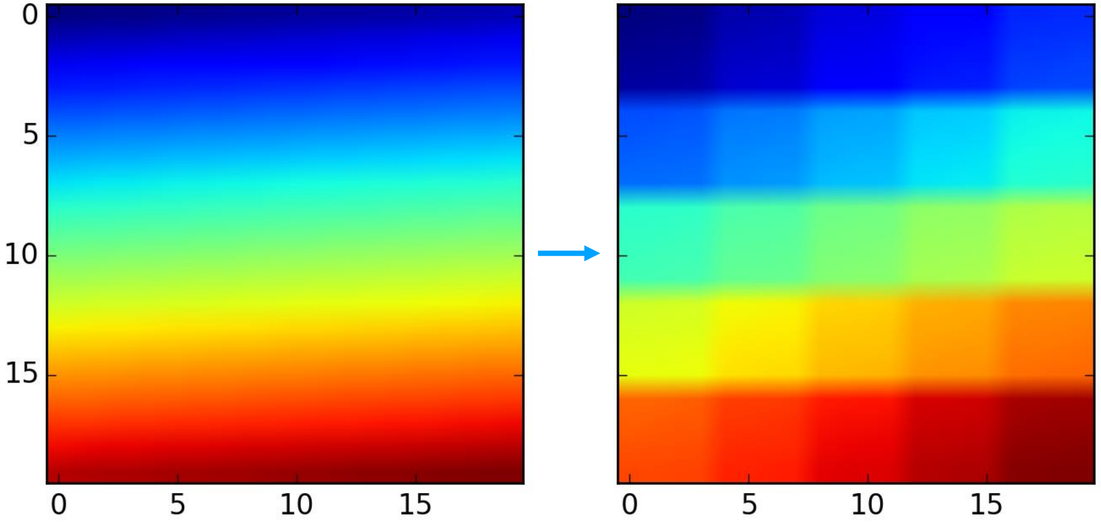


- Roofline model tells us to optimize the operational intensity
- Remember:
  Operational Intensity = $\frac{\text{total operations}}{\text{total bytes loaded}}$
- The most important thing is to mimize the bytes loaded from memory. This means reusing data in memory locations closest to the functional units for as long as possible.



Loop Blocking

- Loop blocking a technique to change the iteration order to improve data locality
- Also known as tiling, or loop nest optimization



```
for (int j=0; j<m; j++)
    for (int i=0; i<n; i++)
        some_array[j*n+i]
```

- After applying loop blocking:

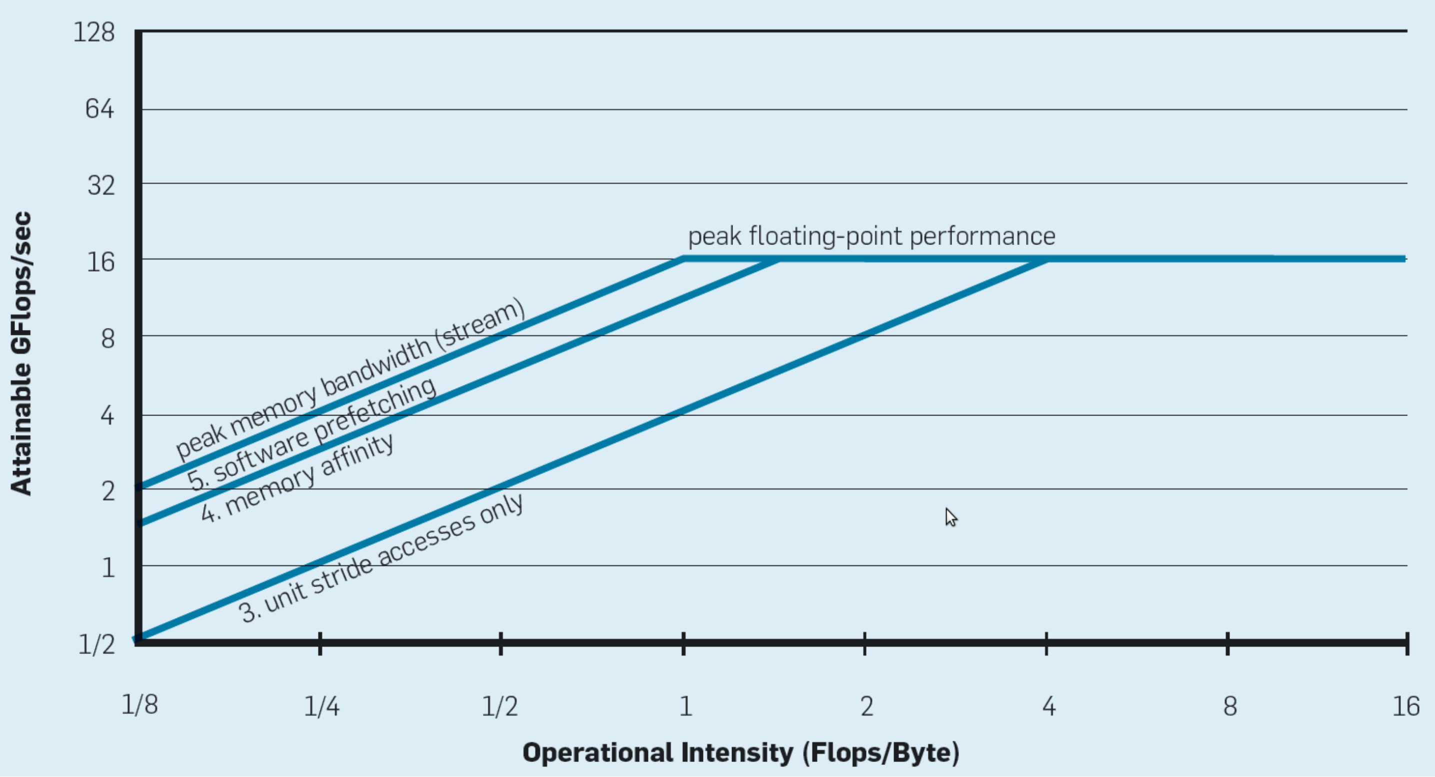
```
for (int j=0; j<m; j+=block_size)              //iterate over blocks
    for (int i=0; i<n; i+=block_size)

        for (int bj=0; bj<block_size; bj++)  //iterate within each block
            for (int bi=0; bi<block_size; bi++)

                some_array[(j+bj)*n+i]
```

Optimizing Achieved Memory Bandwidth

- However, our GPU code is not likely to achieve the theoretical peak memory bandwidth, unless we pay close attention



If your kernel’s performance is bandwidth limited, think of this as the following:

> “the performance is limited by the rate at which operands
> can be supplied to the functional units”

This means your kernel is either latency or throughput bound

Hiding memory latency is about keeping a sufficient number of data operations in flight

This requires sufficient amounts of parallelism:

- Thread-level parallelism: ensure that the number of executing threads remains high
- Instruction-level parallelism: give threads enough independent work to avoid stalls on earlier instructions, e.g. using vector data types
- Ensure that memory operations can be served in parallel, avoid bank-conflicts in shared memory and registers

GPU Memory is optimized for accesses in a ‘coalesced’ manner:

- This means that threads within a warp (adjacent in the x-dimension, and after that y-dimension), should access consecutive memory locations.

Remember threads are executed in warps:

- Memory accesses by threads are grouped into warps and mapped onto cachelines
- The cachelines are then loaded from or stored to memory
- If threads load data with a stride, for example `array[threadIdx.x * 2]`, the amount of data actually loaded from memory is twice as large
- If you only use each other element, you essentially half your effective bandwidth!

Ideally, threads should only read and write consecutive and aligned memory locations

- Think of cacheline lengths at various cache levels
- Be aware of cacheline alignment in memory

- Ensuring sufficient thread-level parallelism is about keeping the GPU's SMs sufficiently occupied, one way to control this is to vary the number of threads per block

- Remember occupancy depends on resource usage: shared memory, registers, thread slots, and thread block slots.

- Thread block dimensions are often fixed at compile, because they are used:
  - to declare the size of shared memory arrays
  - as bounds in for-loops, which may be unrolled if known at compile time
  - as offsets within index calculations that can be optimized away by the compiler

- Registers are per-thread and are determined by the compiler, so the amount of registers needed by the thread block can be controlled by adjusting the size of the thread block

- Figuring out the optimal thread block dimensions for your kernel is usually not possible without experimentation

Further Reading on GPU Memory Optimization

Excellent presentations on GPU memory optimization:

- GPU Memory bootcamp talks by Tony Scudiero [git repo]
  - Best Practices [Slides] [Video]
  - Beyond the Best Practices [Slides] [Video]
  - Collaborative Access Patterns [Slides] [Video]

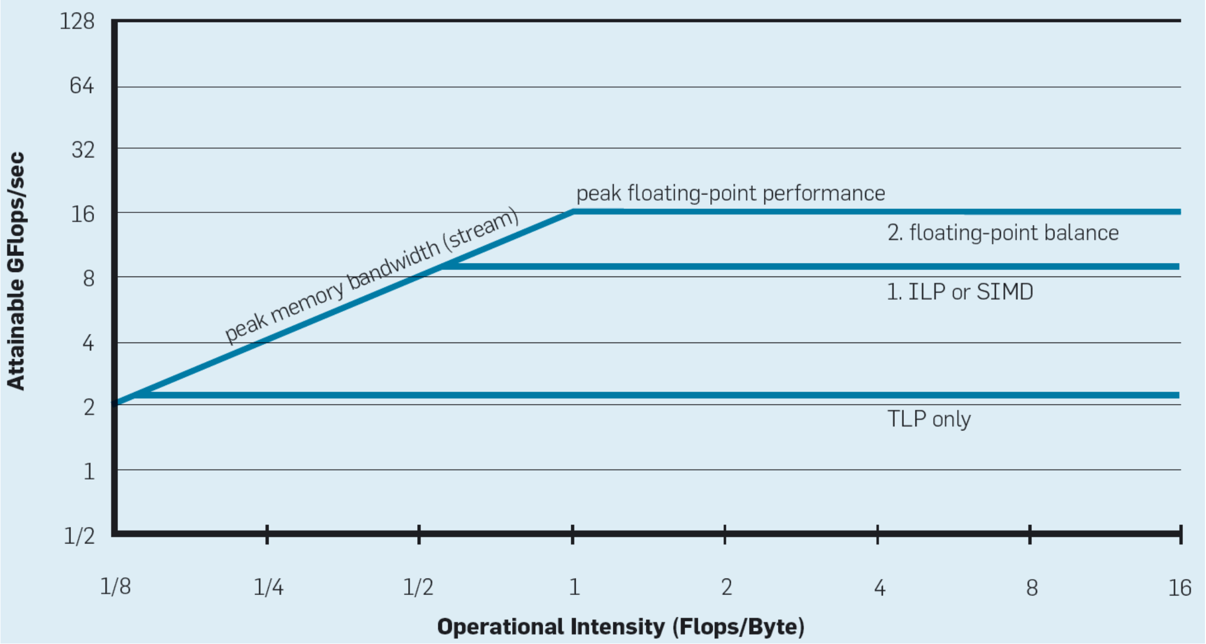
Instruction Stream Optimization

If your kernel is compute-bound, it might still not achieve the peak theoretical compute performance of your GPU

Most important reason for this is that your kernel is not issuing compute instructions to all functional units at every cycle while it's running

Your kernel is doing more than just floating-point arithmetic, for example: memory operations, barrier synchronizations, index calculations, or checking loop conditions.

Some of this is inherent to the algorithm your kernel implements, but some instruction overhead can be avoided



A very common code transformation for GPU code is increasing work per thread
(a.k.a thread coarsening / 1xN tiling / thread block merge)

- Before:

```
sum = a[i] + b[i];
```

- After:

```
sum0 = a[i] + b[i];
sum1 = a[i + 1] + b[i + 1];
```

- Or the right way for the GPU, without introducing stride-2 memory accesses:

```
sum0 = a[i] + b[i];
sum1 = a[i + block_size] + b[i + block_size];
```

The above assumes `block_size` is a multiple of the warp size

What we just did is doubling the amount of work per thread, so for the same input size, so our kernel now only needs half the number of threads to execute

Advantages:

- We reduce a number of redundant operations that were previously distributed across multiple threads, e.g. array index calculations, loop accounting.

- Additionally, if there are multiple threads that need to load the same data, we are now reusing that data in registers.

Disadvantages:

- We trade thread-level parallelism for instruction-level parallelism, increased resource usage further reduces parallelism when it limits occupancy

- Why only double the amount of work? Why not increase work by an arbitrary factor, for example named `tile_size`?

```
#pragma unroll
for (int ti=0; ti<tile_size; ti++) {
   sum = a[i+ti*block_size] + b[i+ti*block_size];
}
```

Note that we've just increased work in one dimension. If we are working on an N-dimensional problem, we might just increase work in any of up to N of dimensions.

```
#pragma unroll
for (int ti=0; ti<tile_size_x; ti++) {
   #pragma unroll
   for (int tj=0; tj<tile_size_y; tj++) {
      sum = a[(j+tj)*n+i+ti*block_size] + b[(j+tj)*n+i+ti*block_size];
   }
}
```

Another way to reduce redundant instructions and improve data locality is to fuse loops

- Before:

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

- After:

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

If N is small and known at compile time the compiler will do this for you

Another way to reduce instructions and increase instruction-level parallelism is to unroll loops

- Before:

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

- After:

```
for (int i=0; i<N; i+=2) {
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
}
```

This of course only works if 2 is a divisor of N

- Even better:

```
#pragma unroll <factor>
for (int i=0; i<N; i++) {
   c[i] = a[i] + b[i];
}
```

If N is known at compile-time the compiler will try to apply (partial) loop unrolling

Overview of GPU Optimization Trade-offs

Improve memory throughput
- Find the best way to map your algorithm’s data-parallel iteration space to threads and thread blocks
- Coalesce memory accesses, possibly by using shared memory as intermediate storage
- Increase operational intensity as much as possible by utilizing specialized memories and apply various levels of loop blocking to improve cache utilization

Minimize memory latency
- Reduce/hide memory latency by using vector data types to increase size of memory loads and stores
- Fuse and/or partially unroll loops to condense instruction stream and increase instruction-level parallelism

Condense the instruction stream
- Select the best performing thread block dimensions, trade-off between instruction stream efficiency, data reuse, and latency hiding
- Increase or decrease work per thread in N dimensions to condense instruction stream, at the cost of increased resource usage

Optimizing GPU Code for Performance

Requires that you get all the details exactly right:

- Mapping of the problem to threads and thread blocks
- Thread block dimensions
- Data layouts in the different memories
- Tiling factors
- Loop unrolling factors
- How to overlap computation and communication
- …

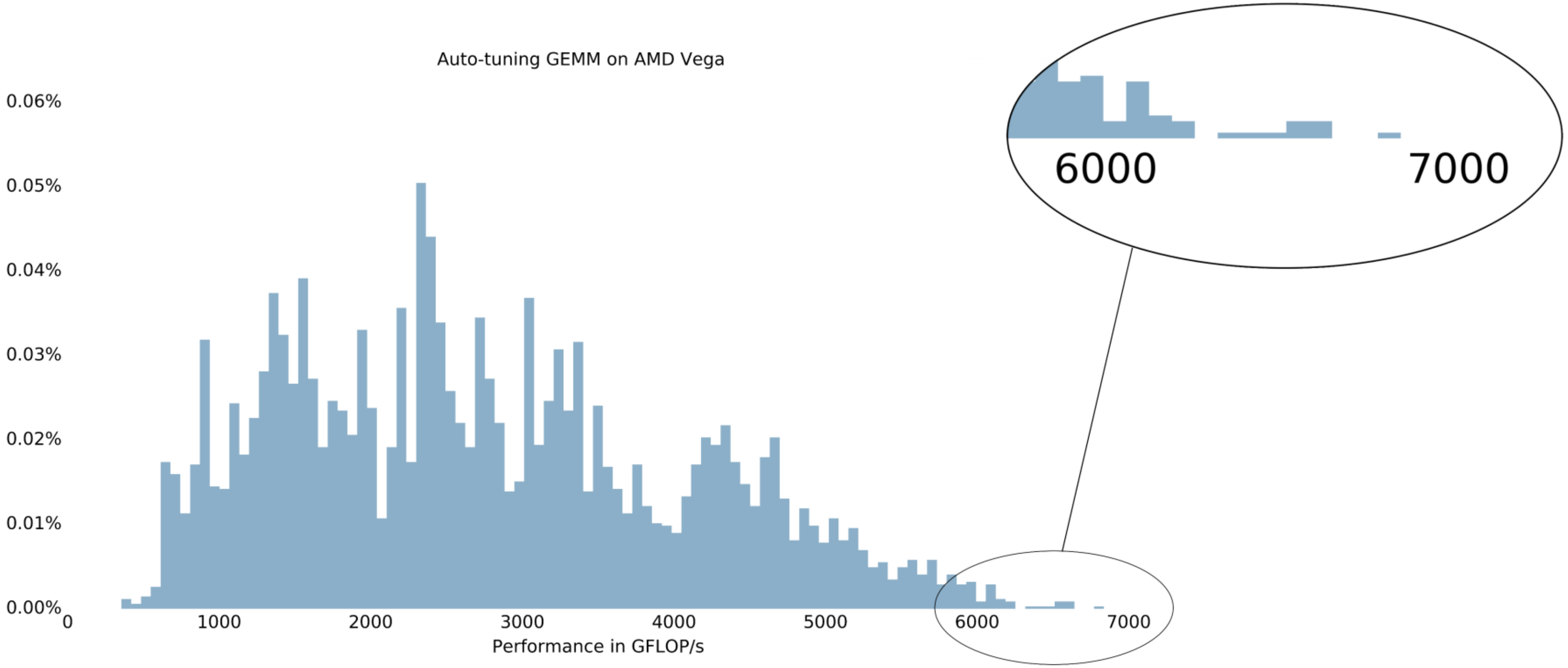
Problem:

Creates a very large and discontinuous search space

Auto-Tuning for 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

Hands-on Exercise

15:30 – 15:45

Fourth Hands-on Session: Local Averages

- Select the notebook for the fourth exercise
  - https://github.com/benvanwerkhoven/gpu-course/blob/master/notebooks/local_averages.ipynb

- The kernel is correct, your task is to optimize the code and improve performance
- You can keep working on this exercise after the course ends

- Hints:
  - Is the memory access pattern optimal?
  - Are threads mapped the right way for the problem?
  - Could vector operations be useful?
  - Is there any data reuse available?

Discussion

15:45 – 16:00