netherlands eScience center

B-Q Minded Summer School GPU Programming course

By Alessio Sclocco and Ben van Werkhoven
August 26, 2019

Schedule

- 08:45 – 10:45 Introduction to GPU Computing
- 11:00 – 12:30 High-level intro to CUDA Programming Model
- 13:30 – 15:40 CUDA memories and execution
- 16:00 – 18:00 Hands-on / assignment

Download the slides!

- Get your own copy of the slides so you can read along and click on links
  See: https://github.com/benvanwerkhoven/gpu-course/

- 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

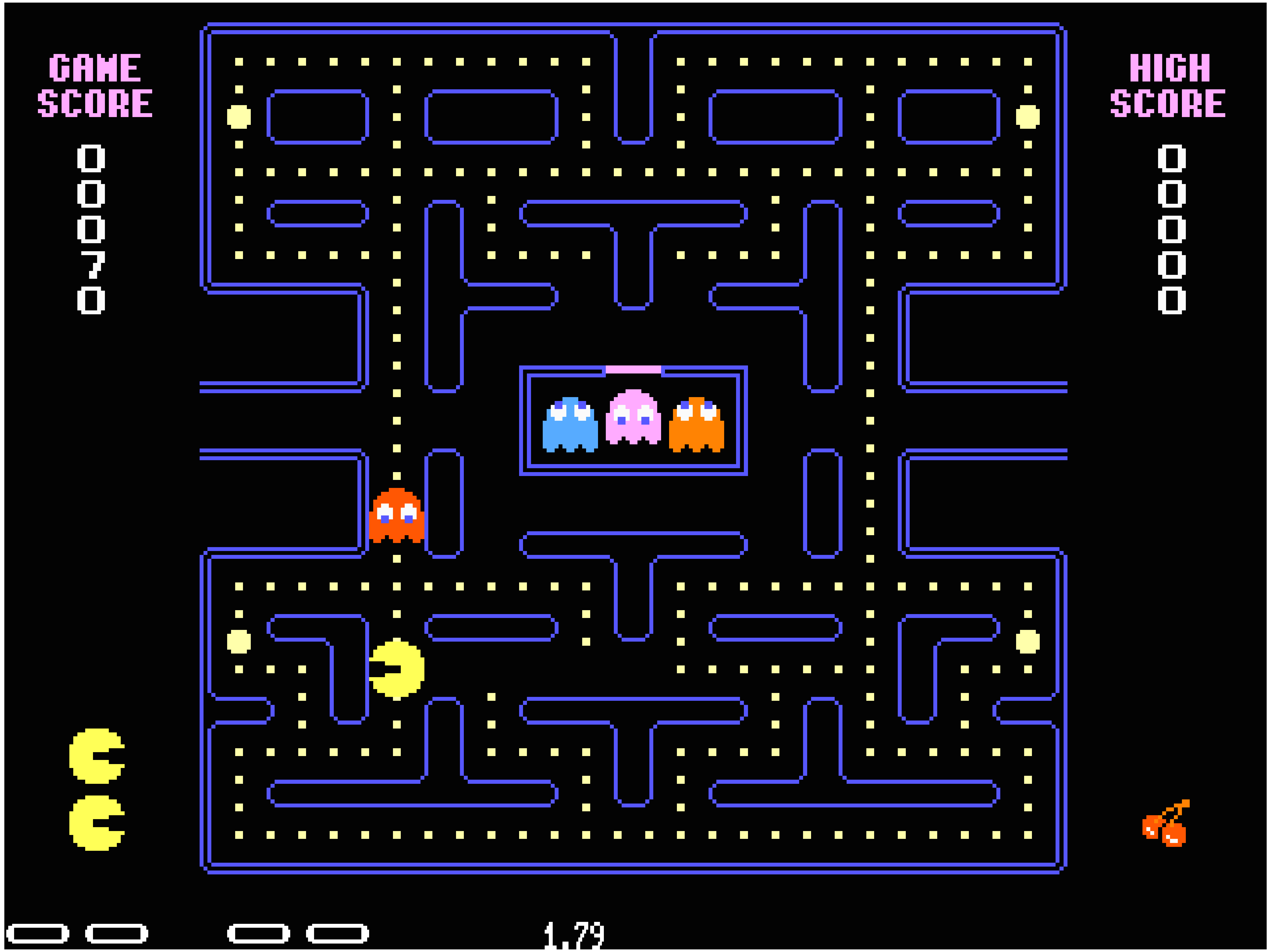
Introduction to GPU Computing

What is a GPU?

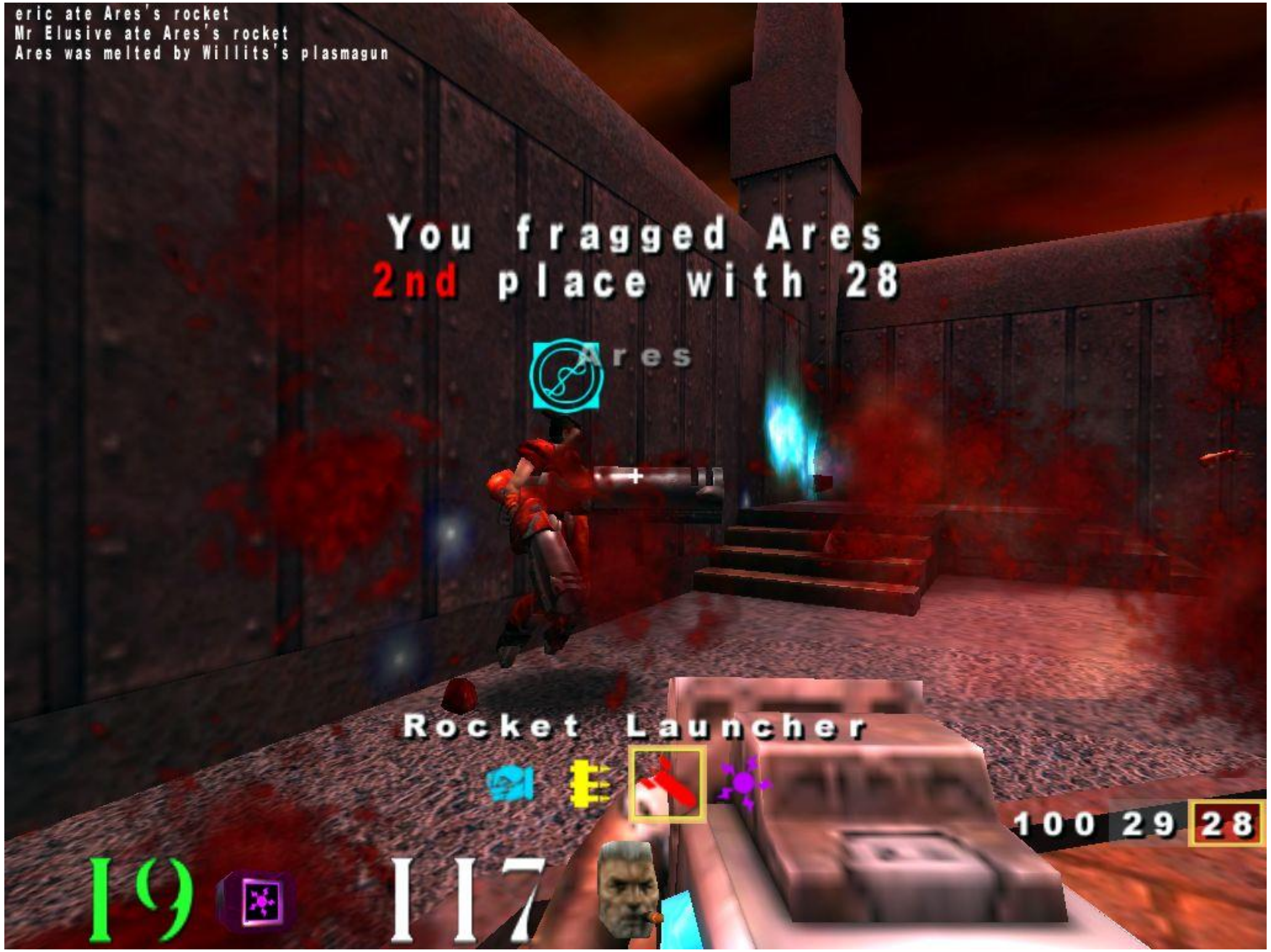
- Graphics Processing Unit –
  The computing chip on a graphics card

- GPGPU

Graphics in 1980



Graphics in 2000

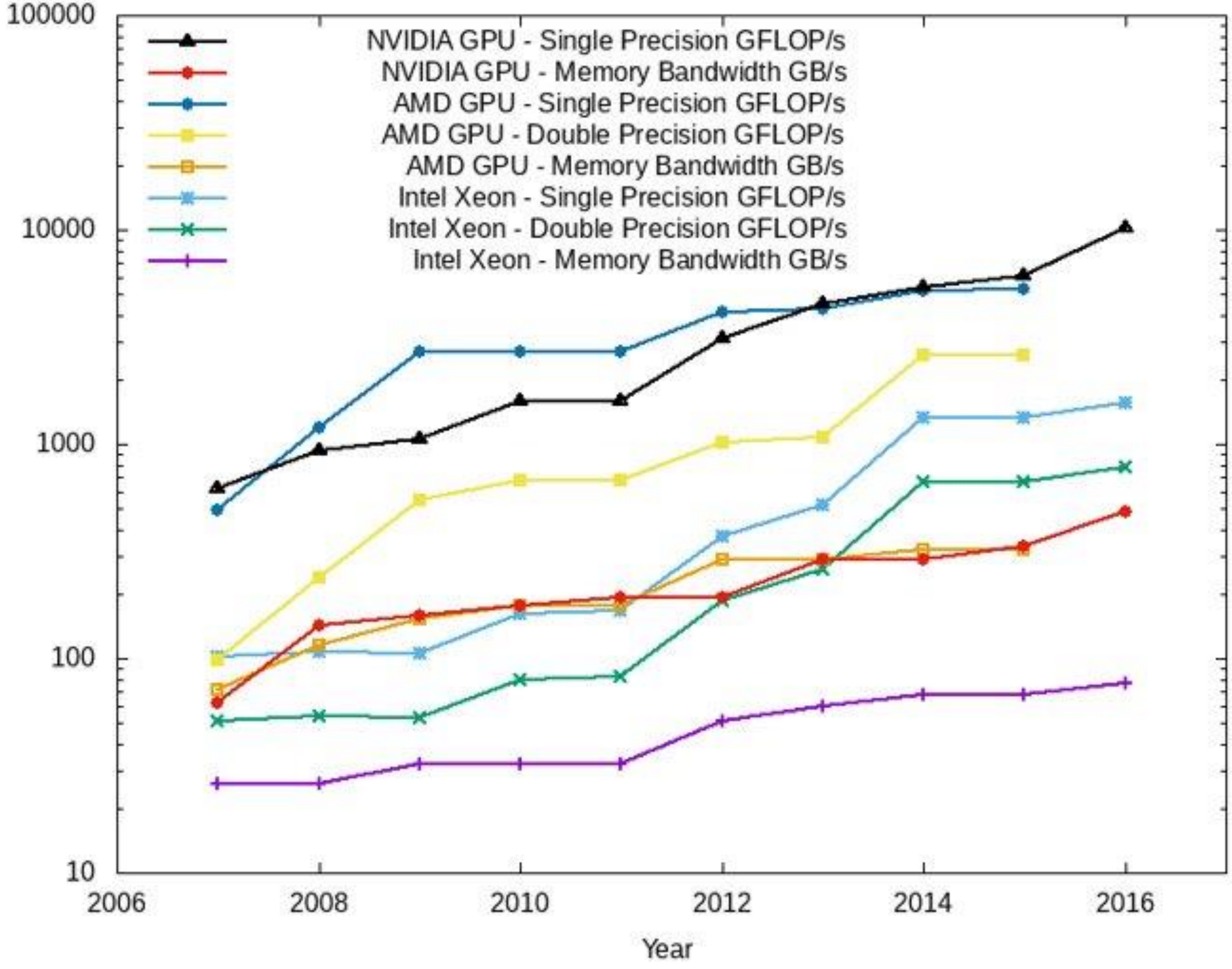


eric ate Ares's rocket
Mr Elusive ate Ares's rocket
Ares was melted by Willits's plasmagun

You fragged Ares
2nd place with 28

Graphics Now

Performance Comparison: CPUs vs GPUs

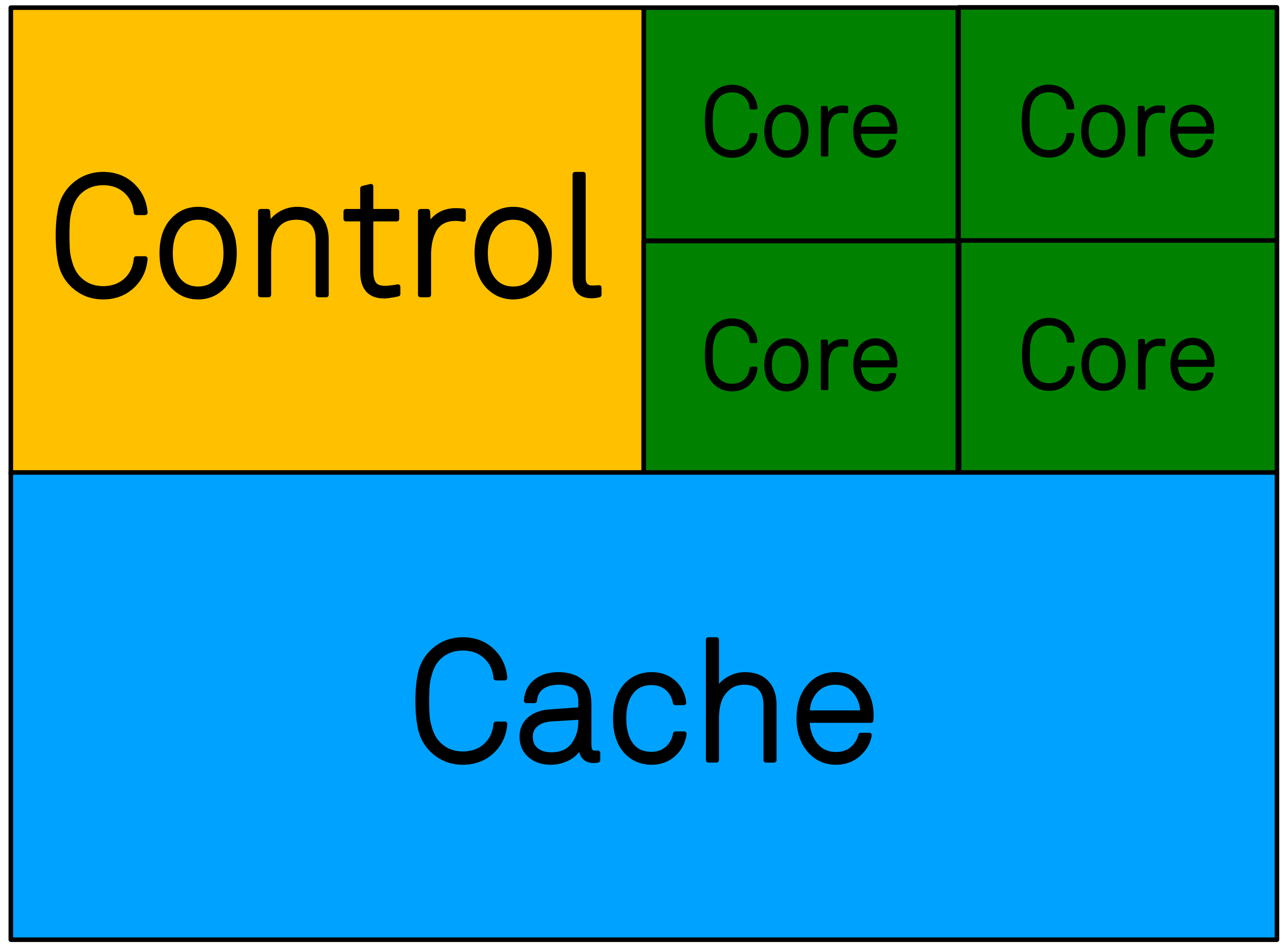


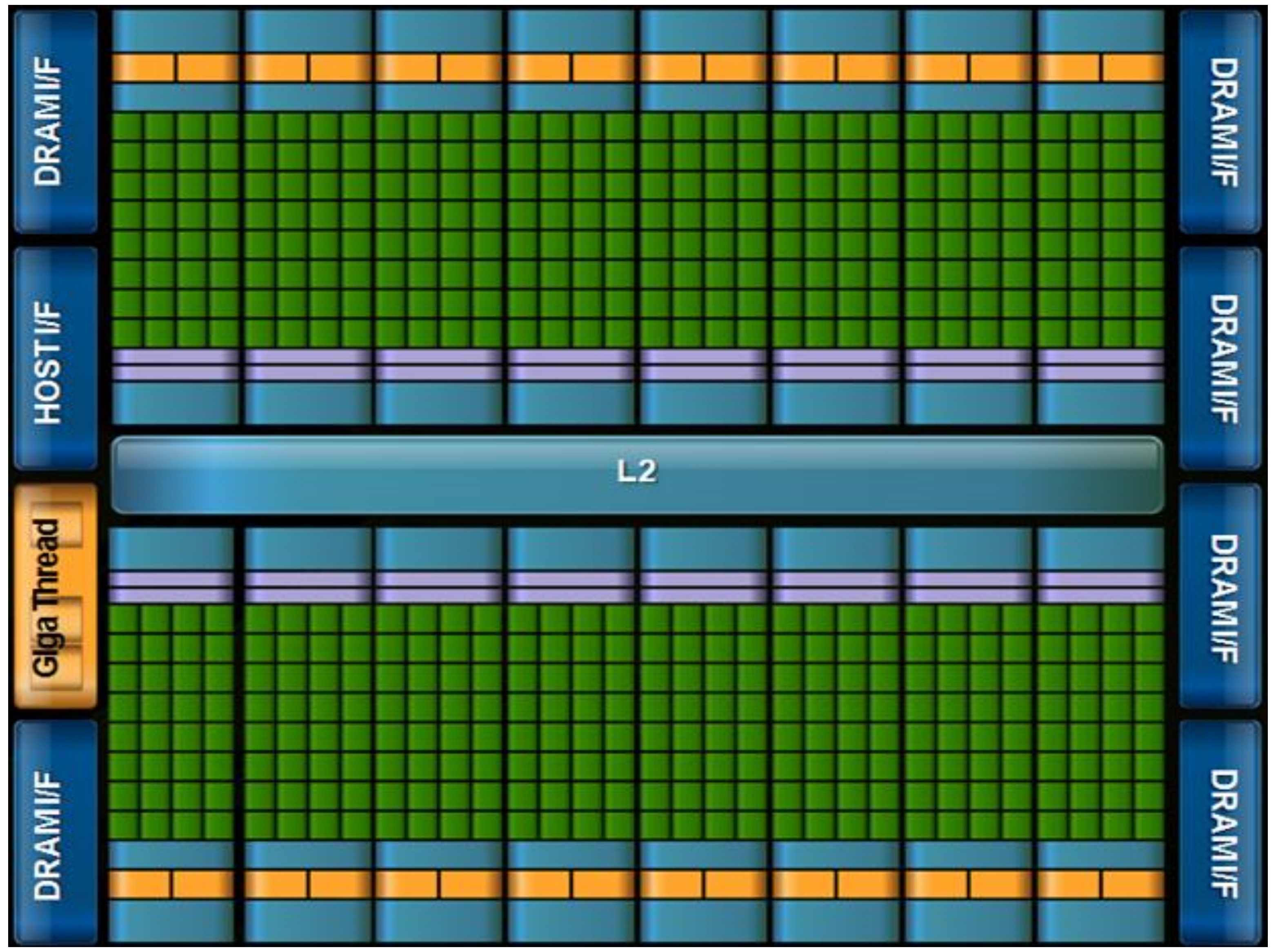
The World's Fastest Supercomputer: Oak Ridge’s Summit

- Number 1 in TOP500 list (June 2019)
  - 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
    - 6 GPUs per node
  - 2,397,824 cores

Design: CPUs vs GPUs

- 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





A high-level introduction to GPU Programming

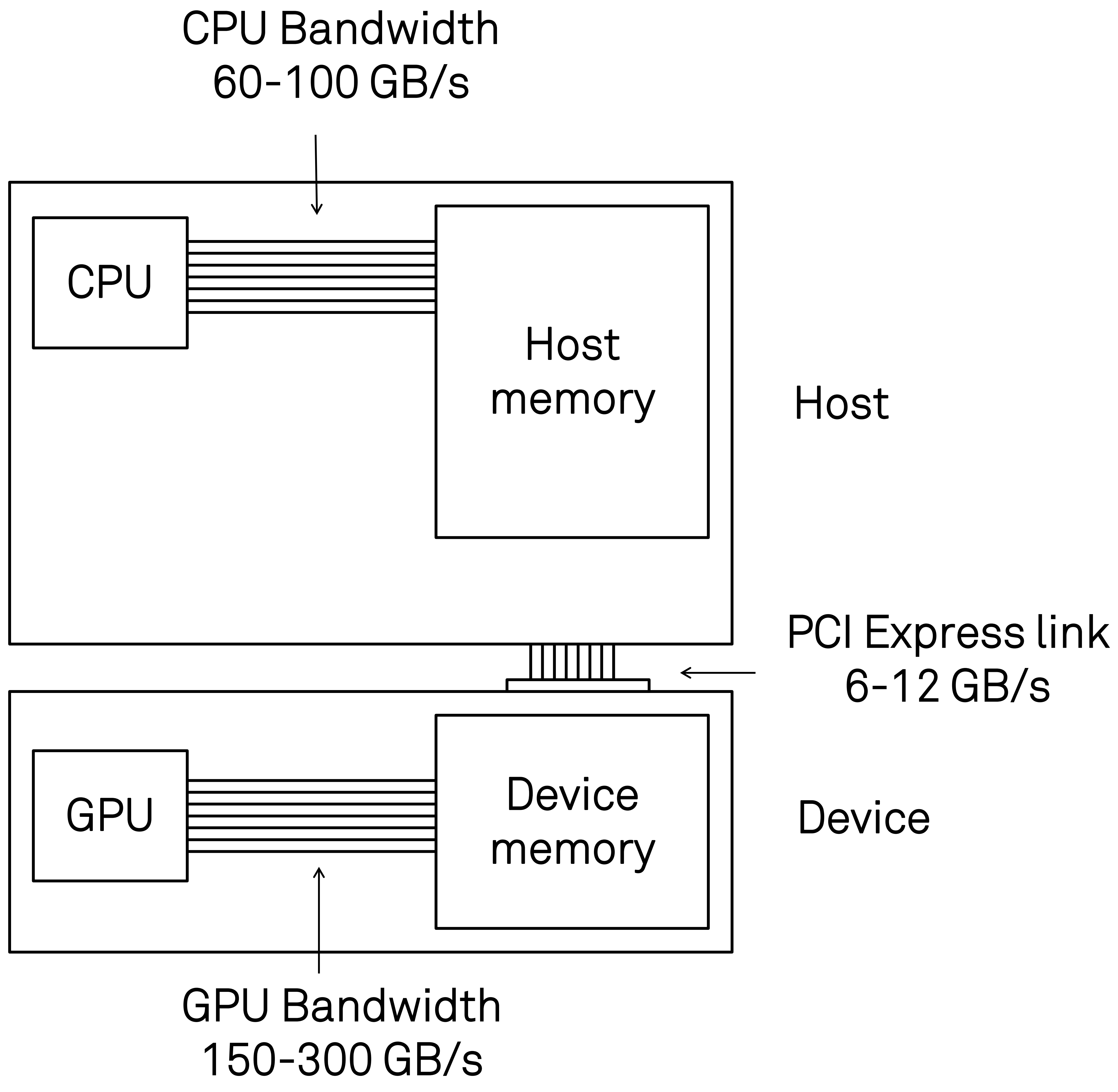
Why is GPU Programming different?

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



Parallelization and mapping

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

Rewriting software

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)

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 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)

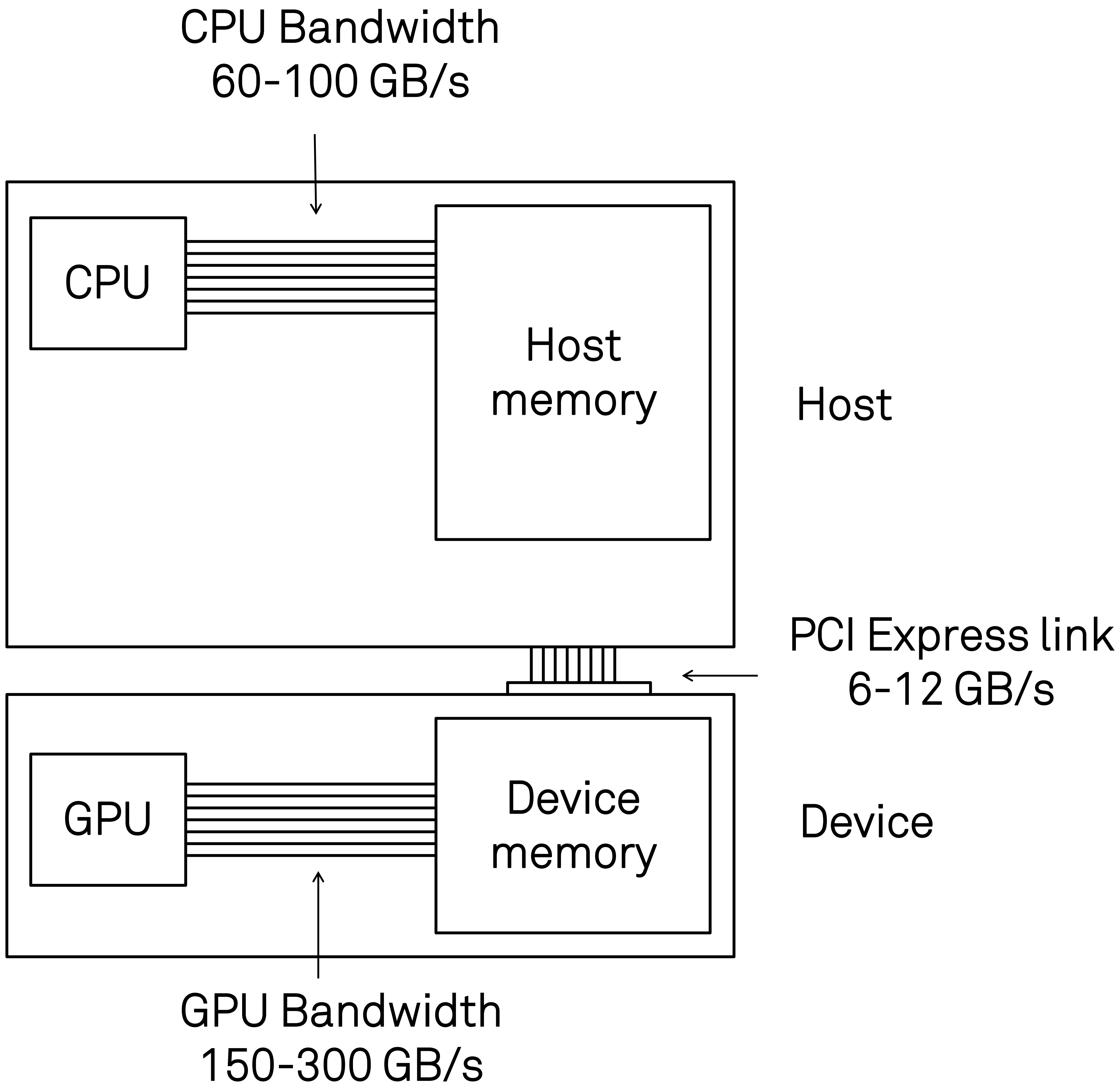
- In general it’s best to keep working on transferred data for as long as possible

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

Summary

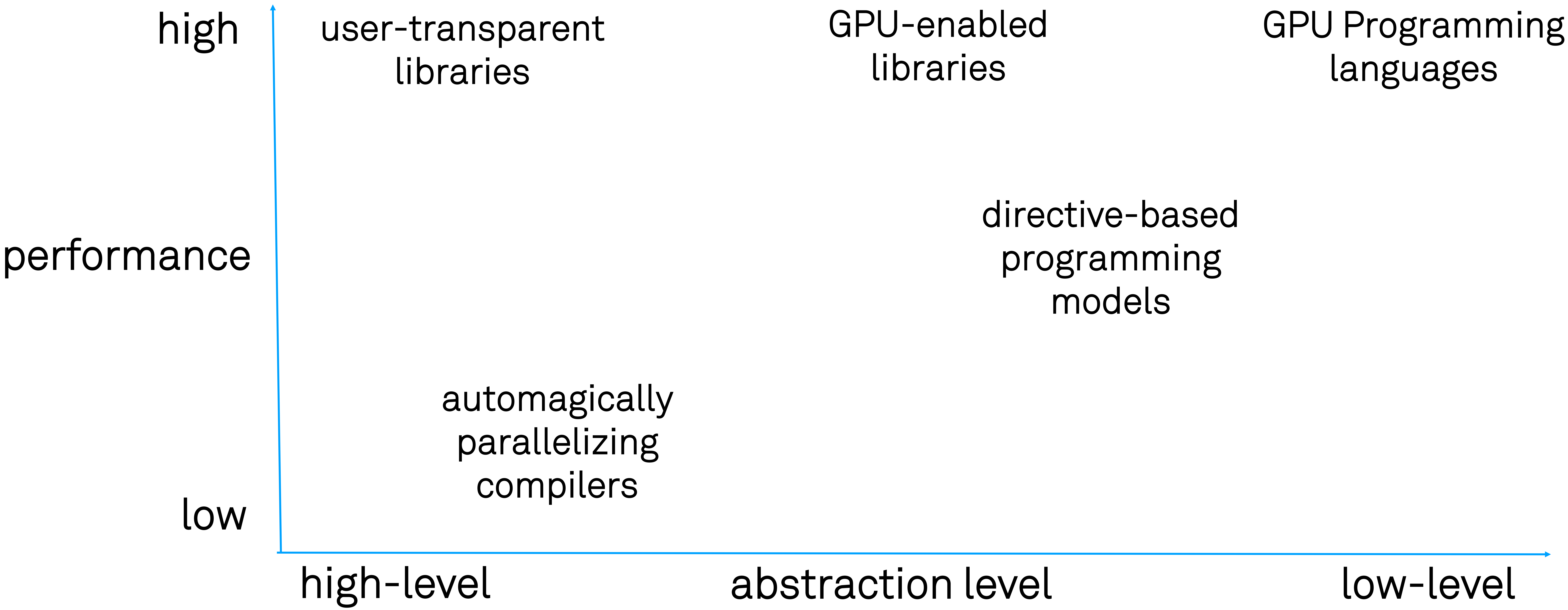
Main differences normal and GPU programming:

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

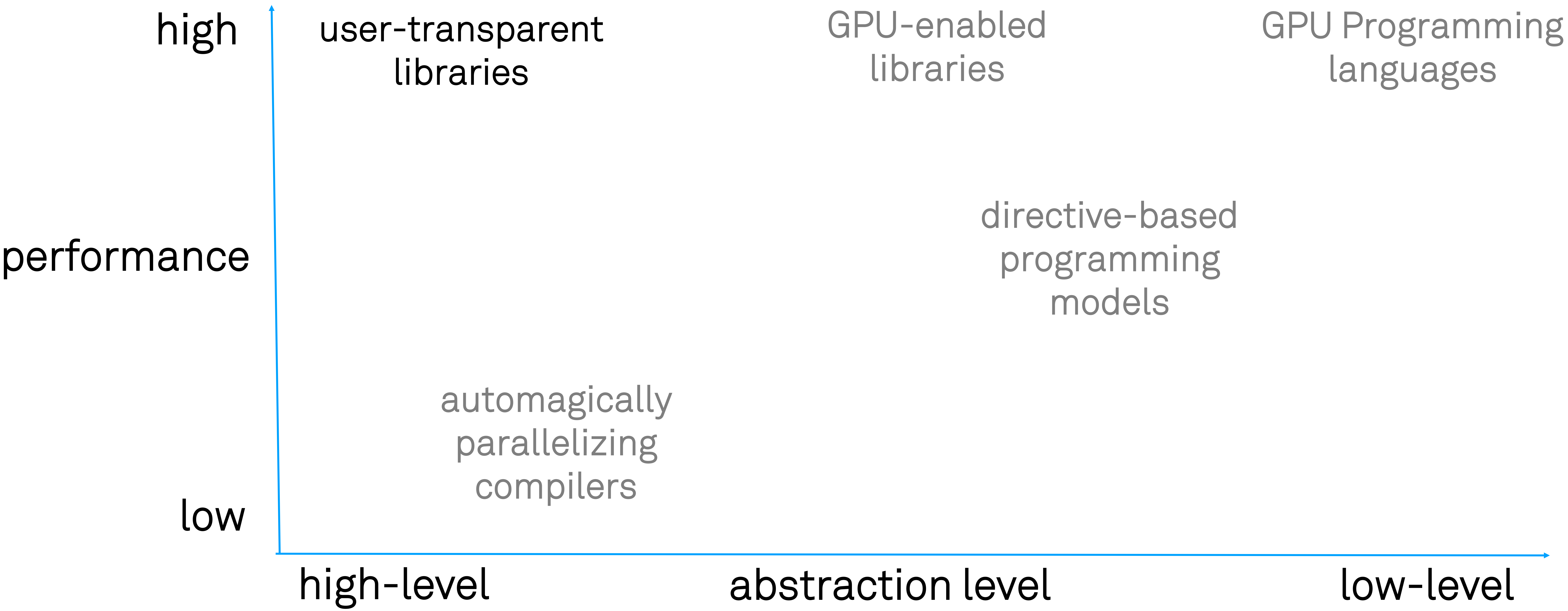


Overview of GPU programming technologies

GPU Programming techniques


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 techniques



high

performance

low

user-transparent libraries

GPU-enabled libraries

GPU Programming languages

directive-based programming models

automagically parallelizing compilers

high-level

abstraction level

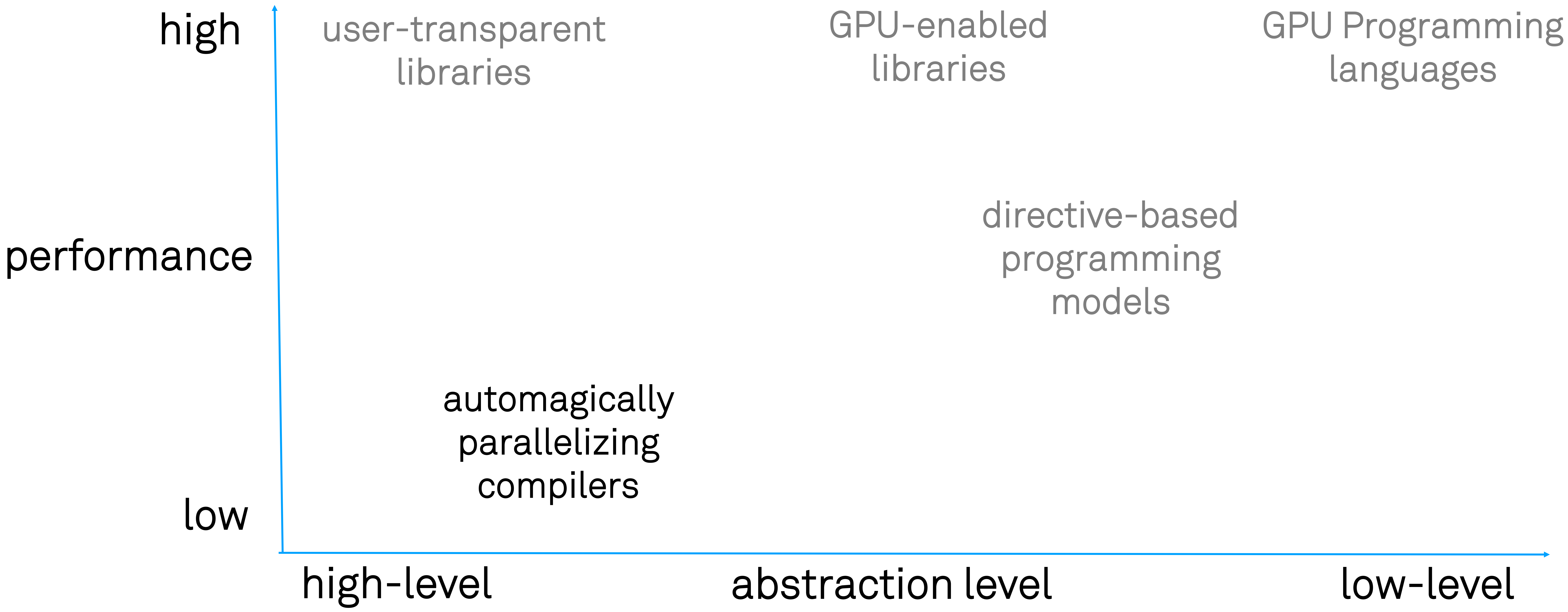
low-level

User-Transparent libraries

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

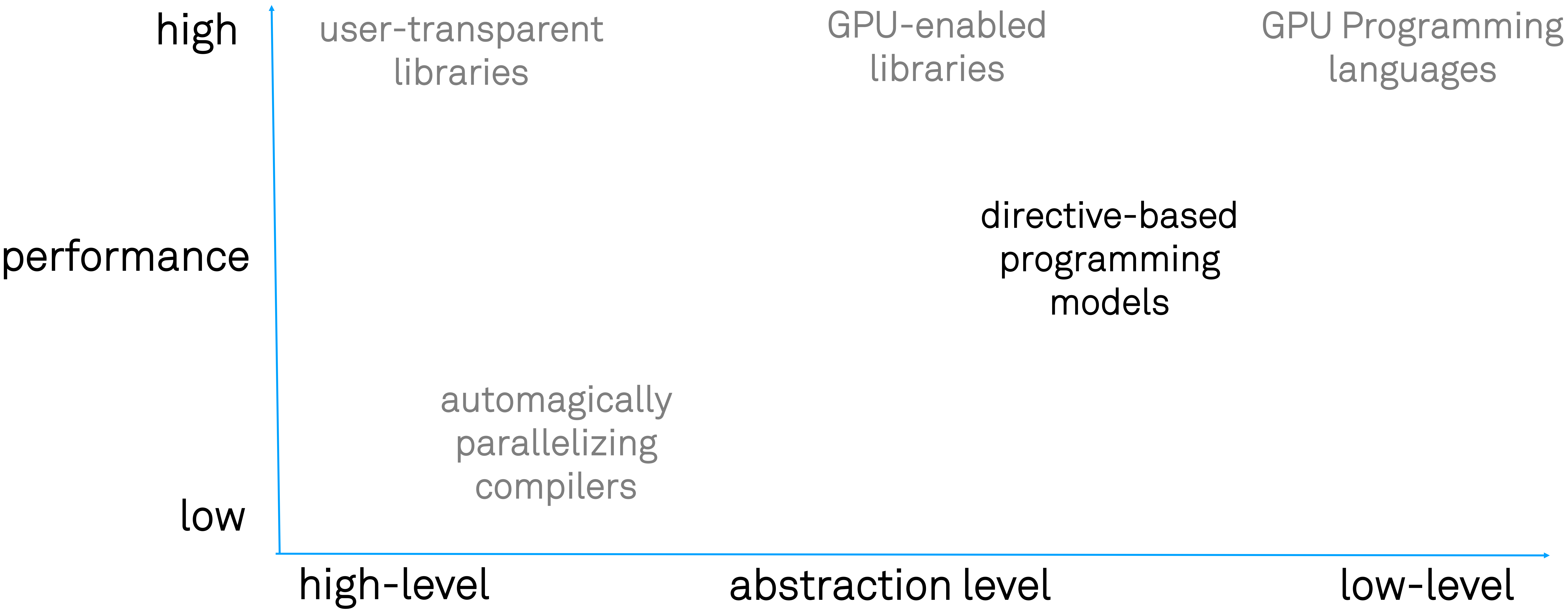
- Difficult to build:
  - Library must maintain state, any init() or destroy() methods already break transparency
  - Library designer has to decide how to manage GPU memory
- Difficult to use:
  - Optimizing application performance is hard when you don’t know what happens inside the library

GPU Programming techniques



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

GPU Programming techniques



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

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

Pros and Cons

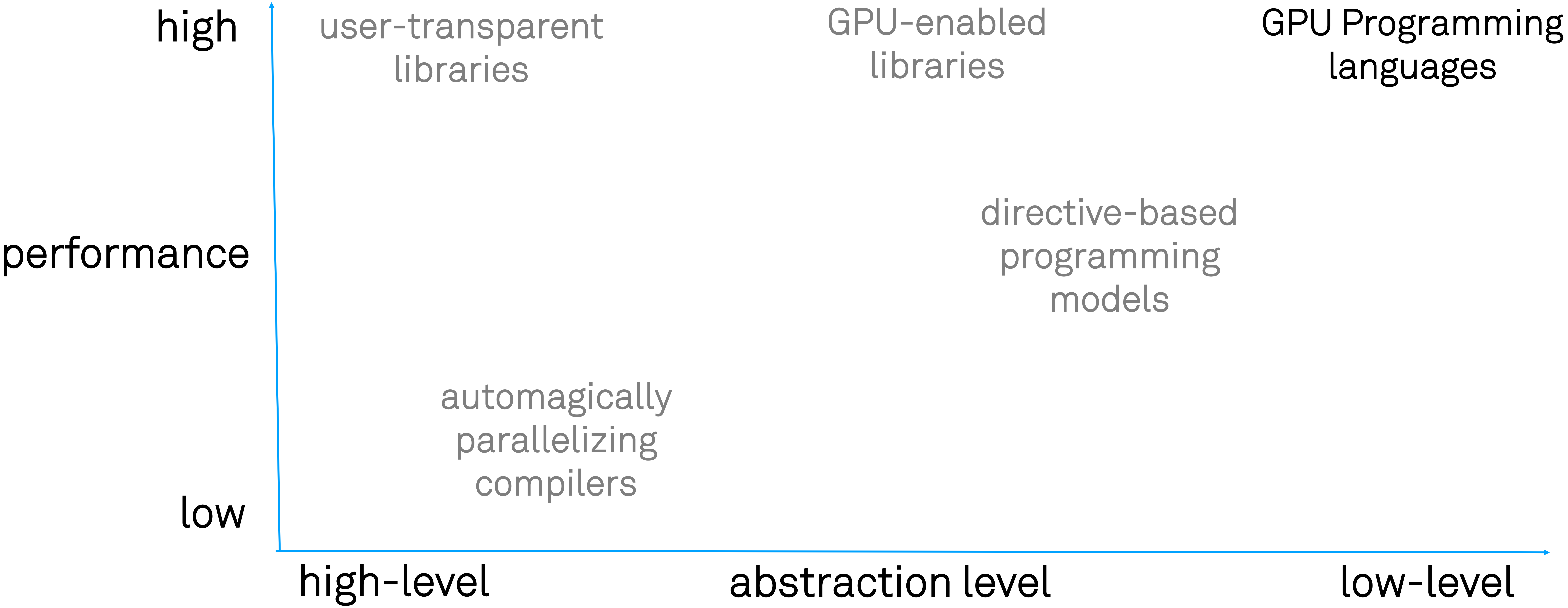
- Advantages:
  - Program is kept in the original language, with directives
  - Easy to get some performance improvement
  - Can serve as a gentle introduction to GPU Programming
- Drawbacks:
  - False sense of security: Directives move the responsibility for program correctness from the compiler to the user, if you say something is parallel the compiler will parallelize it regardless of whether it actually is
  - False sense of simplicity: If you want high performance you still need to know a great deal about (and provide device-specific parameters for) the device your code targets
  - Directives can become really numerous and can obfuscate the original program, having a separate source may be cleaner
  - Accelerating a program with directives for high performance can still require changes to the original code, such as changing data layouts, reordering and merging loops, and so on

Memory management

- From the OpenACC specification:
  “In the OpenACC model, data movement between the memories can be implicit and managed by the compiler, based on directives from the programmer. However, the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:”
  - Memory bandwidth between host memory and device memory
  - Device memory can be smaller than host memory
  - Pointers to host memory can not be dereferenced on the device and vice versa

- So while it’s *“implicit and managed by the compiler”* you have specify all information in a similar way as you would with CUDA or OpenCL, only using directives instead of function calls.

GPU Programming techniques



high

performance

low

high-level

abstraction level

low-level

user-transparent libraries

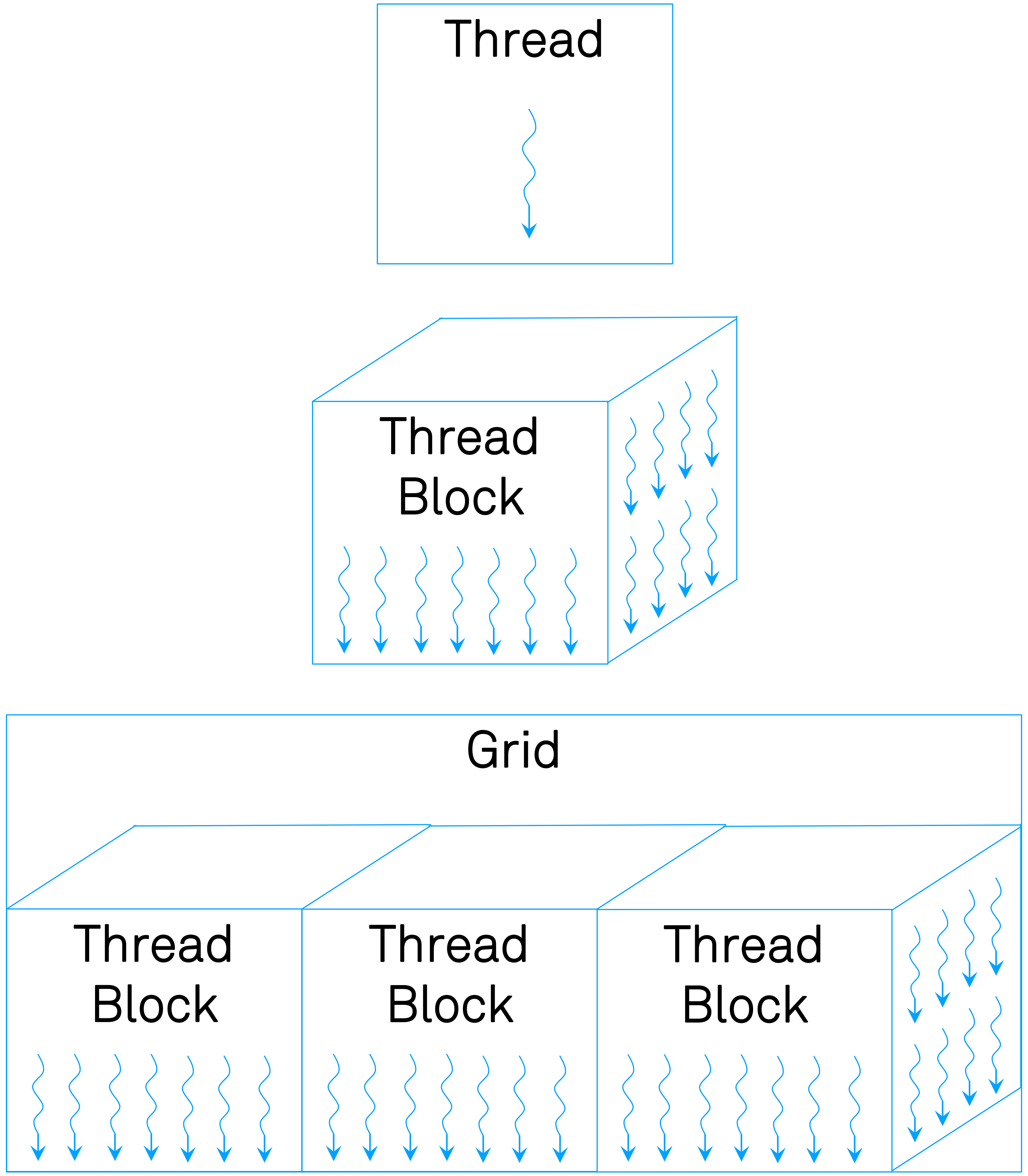
GPU-enabled libraries

GPU Programming languages

directive-based programming models

automagically parallelizing compilers

Writing GPU kernels

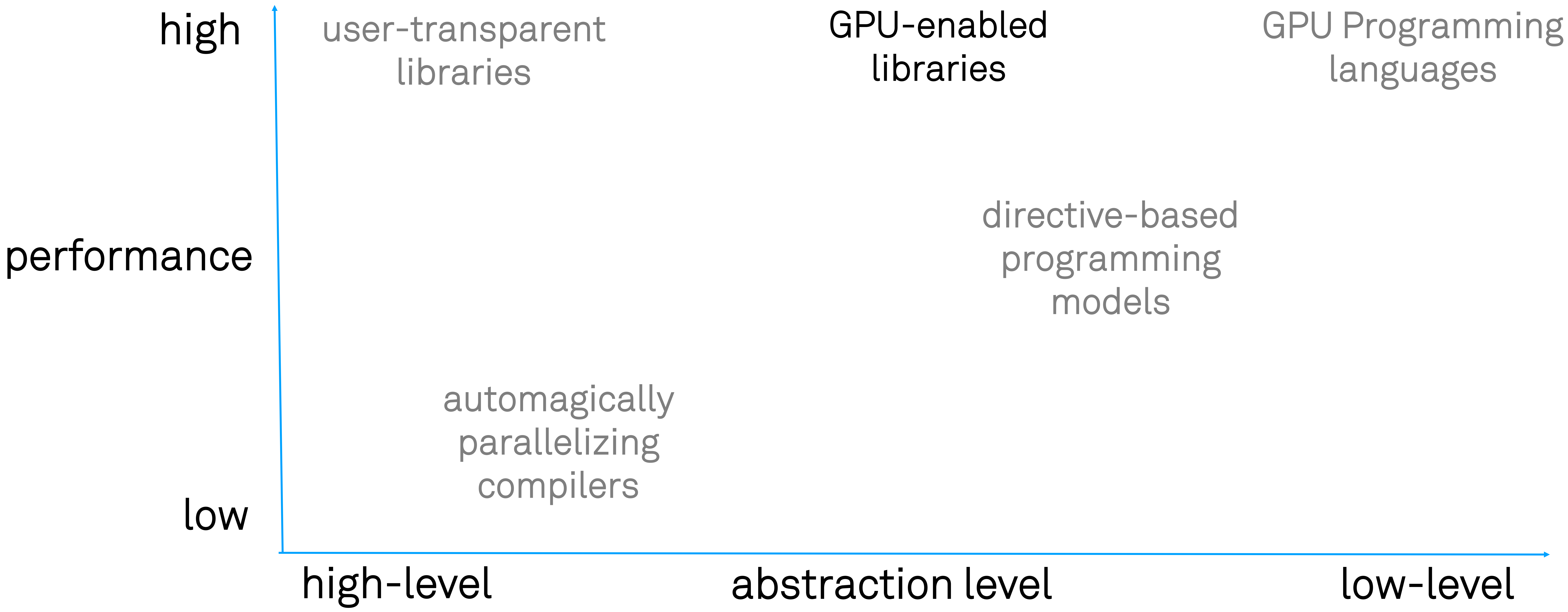


Registers

Shared memory

Global memory
Constant memory

GPU Programming techniques


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-enabled Libraries

- 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

Coffee Break

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 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.

CUDA Programming Model

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

The host part:

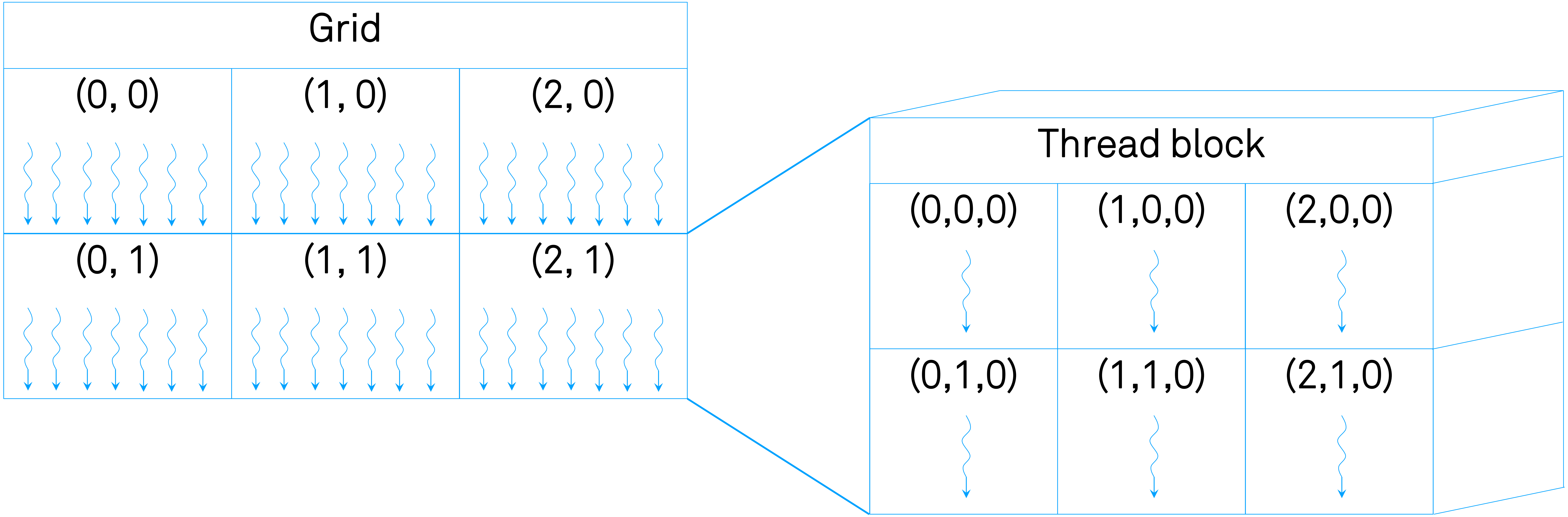
- Allocates memory and transfers data between host and device memory, and starts GPU functions

The device part:

- Consists of functions that 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

Thread Hierarchy

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



Threads

- 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

- 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();
  ```

Setup hands-on session

- Open a terminal
- Login using ssh on the server

- Clone the git repository by typing:

  -
    ```
    git clone https://github.com/benvanwerkhoven/gpu-course.git
    ```

- Change to directory vector_add within the newly created directory gpu-course

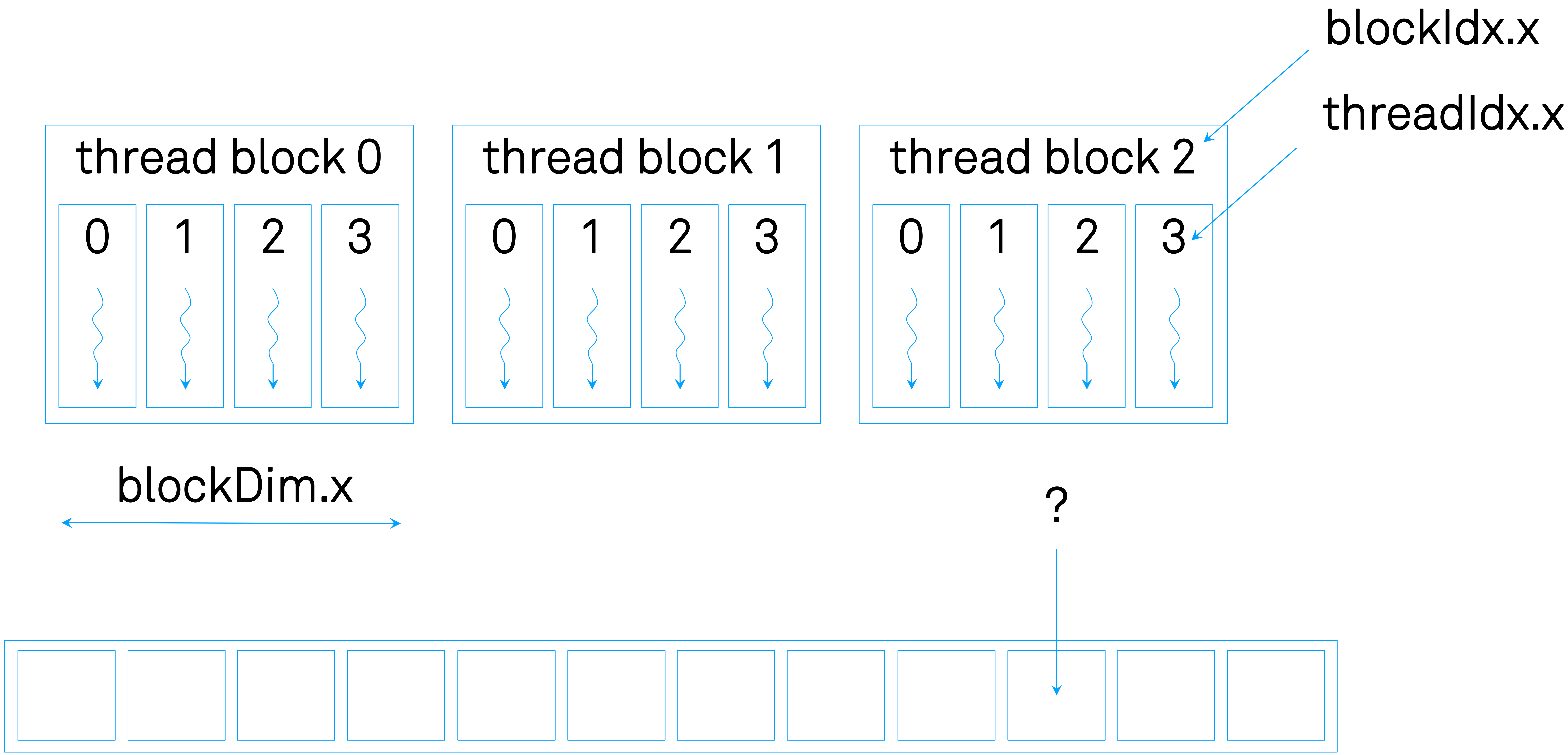
First hands-on session

- You can choose to use the C hostcode or the Python hostcode version for this exercise
- Using C:
  - Compile by typing `make`, run by typing `./vector_add`
- Using Python:
  - Run by typing `./vector_add.py`

- 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://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels
    - 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

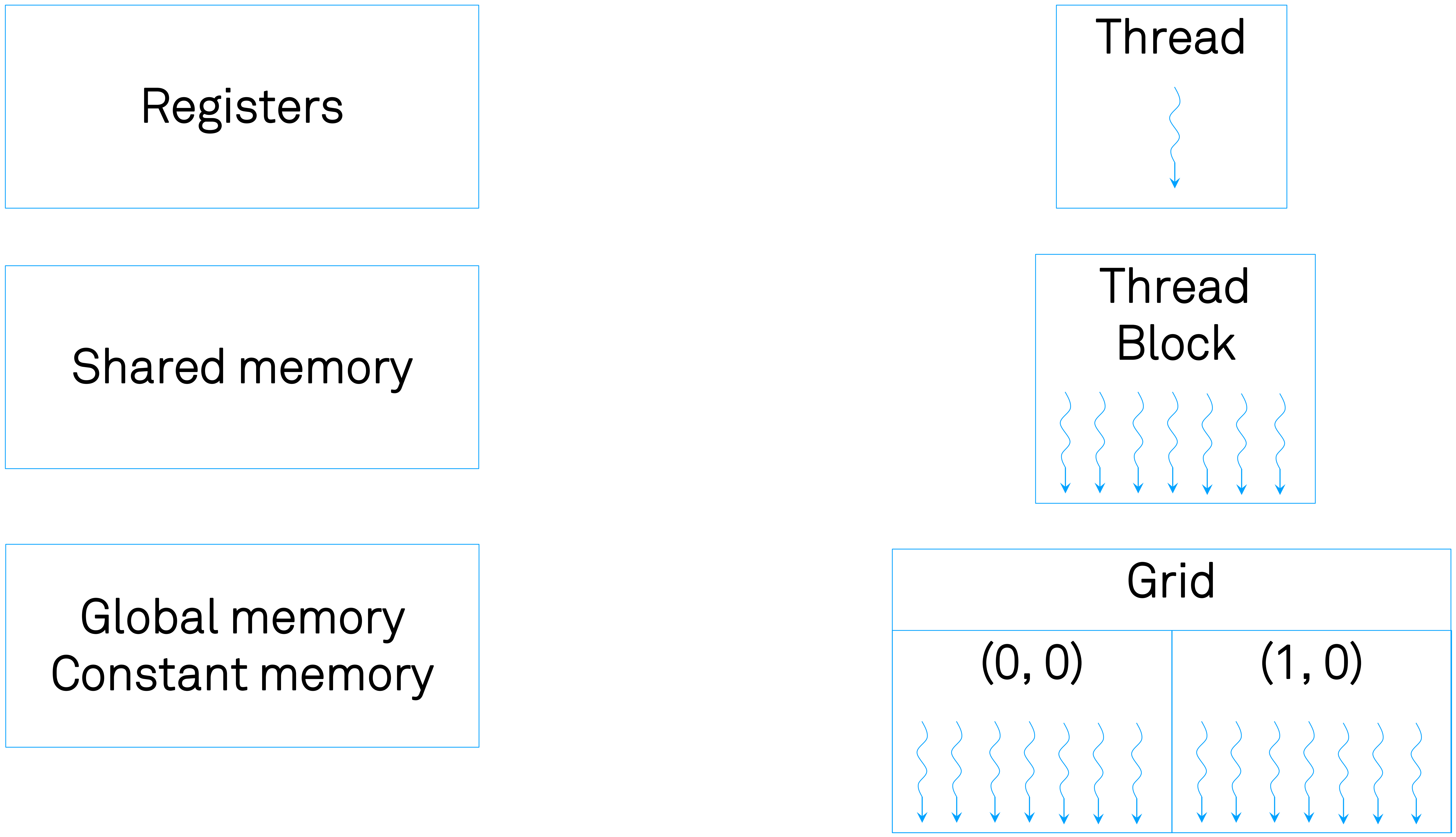
Hint


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

Lunch break

CUDA Memories

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`

2nd Hands-on

- Go to directory pnpoly, look at the source files

- Store the vertices in constant memory space:
  - Declare a **`float2`** array of size **`VERTICES`** as a global variable (choose a unique using the **`__constant__`** qualifier
  - Make sure the constant memory array is used inside the kernels, instead of the currently used '**`vertices`**' array in global memory, just leave it unused in the kernel (if you change the kernel arguments you have to change the hostcode as well)

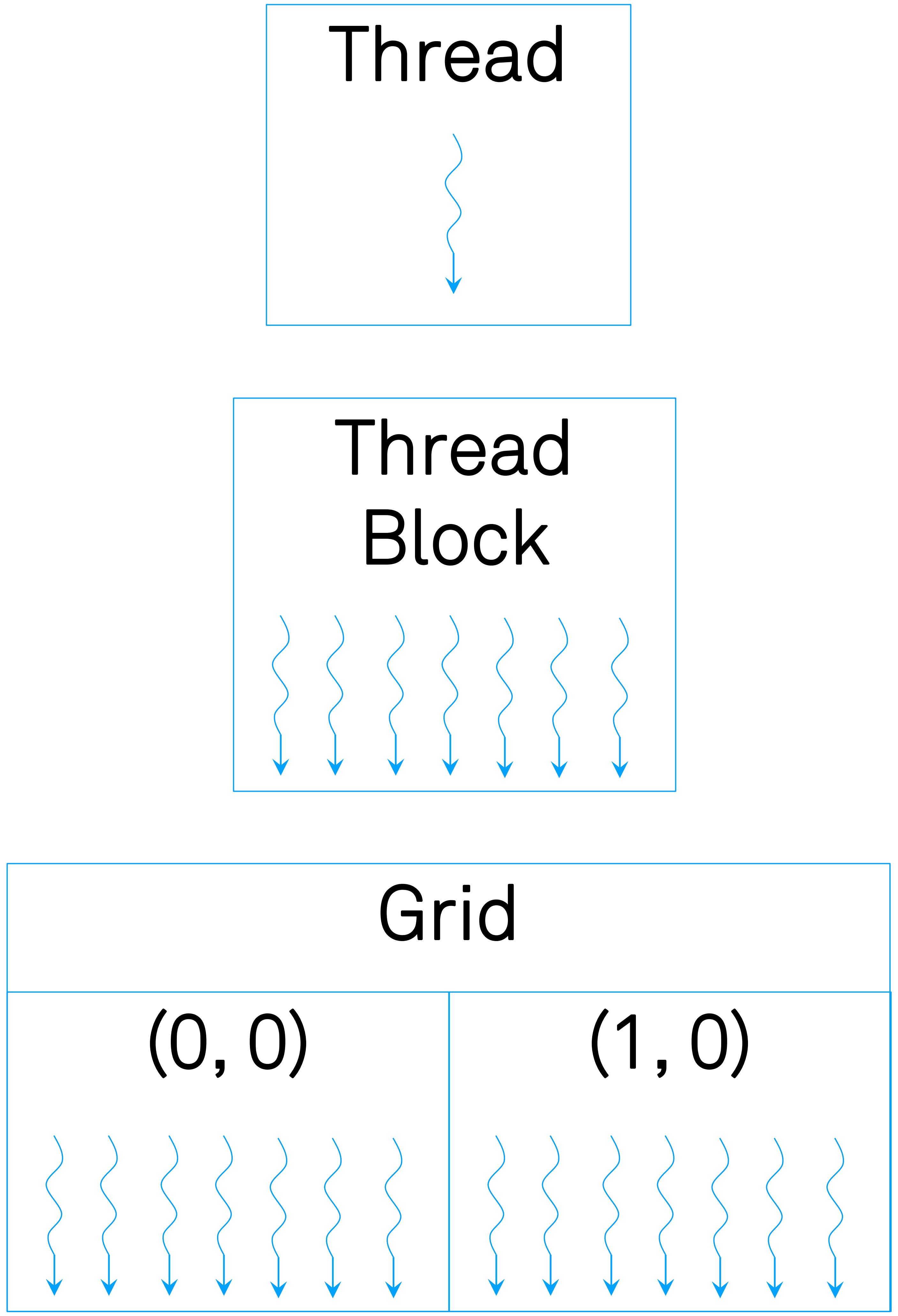
- In C:
  - Add a new **`cudaMemcpyToSymbol()`** after the **`cudaMemcpy()`** for vertices and make sure it copies to the right place. Please leave the global memory copy of vertices in place, the reference kernel uses it.
  - See CUDA documentation on **`cudaMemcpyToSymbol`**
- In Python:
  - Python users can use **`memcpy_htod()`,** but need to find the symbol to copy to
  - See PyCuda documentation on **`get_global`**

CUDA memory hierarchy

Registers

Shared memory

Global memory
Constant memory



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
    ...
```

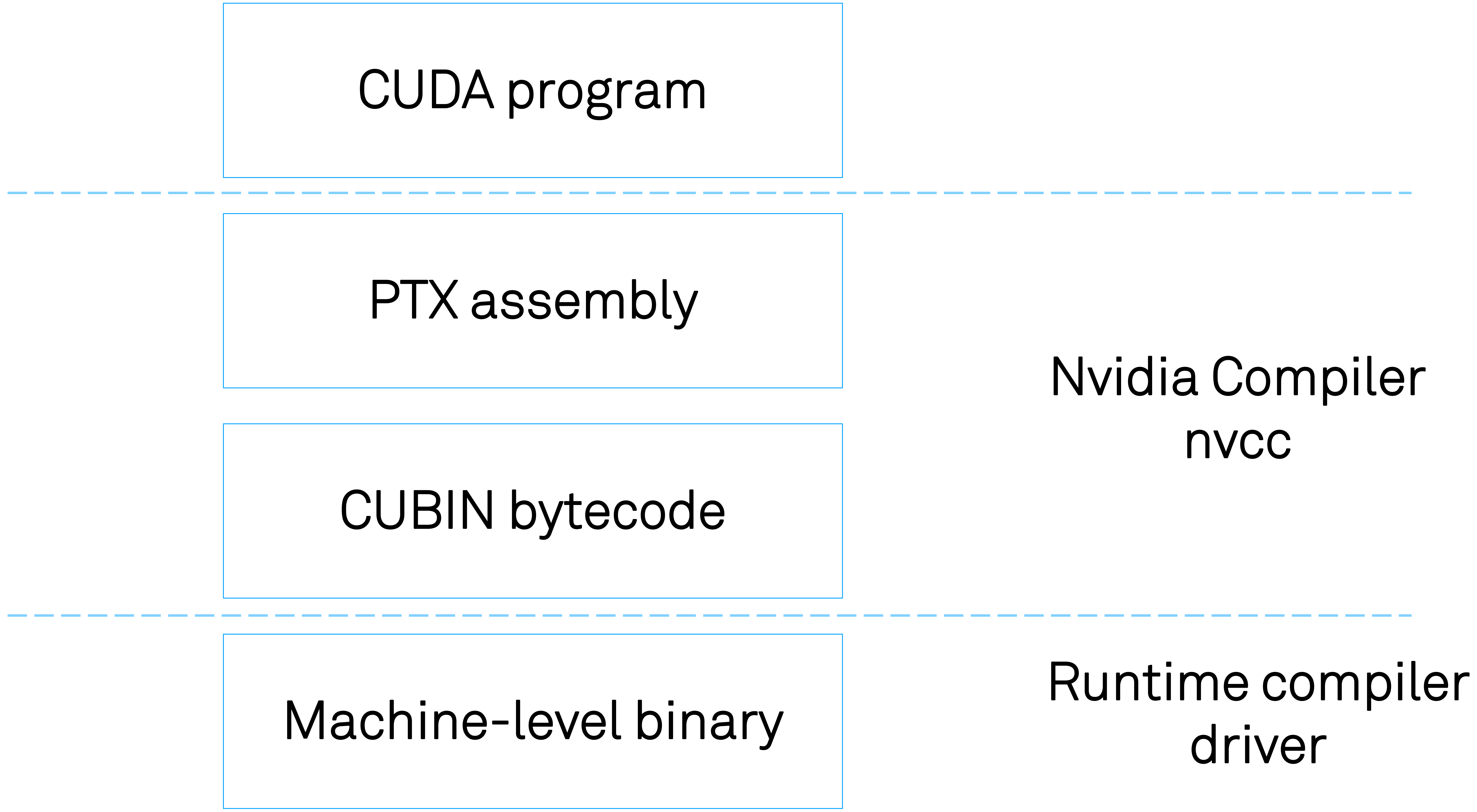
- Shared memory
  - Variables have to be declared using `__shared__` qualifier, size known at compile time
  - In the scope of a thread block, all threads in a thread block see the same piece of memory
  - Not initialized, threads have to fill shared memory with meaningful values
  - Not persistent, 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
    }
}
```

CUDA Program execution

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	Team
Warp	CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE	Worker	SIMD Chunk
Thread	Work item	Vector	Thread or SIMD

- 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 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)
    ```

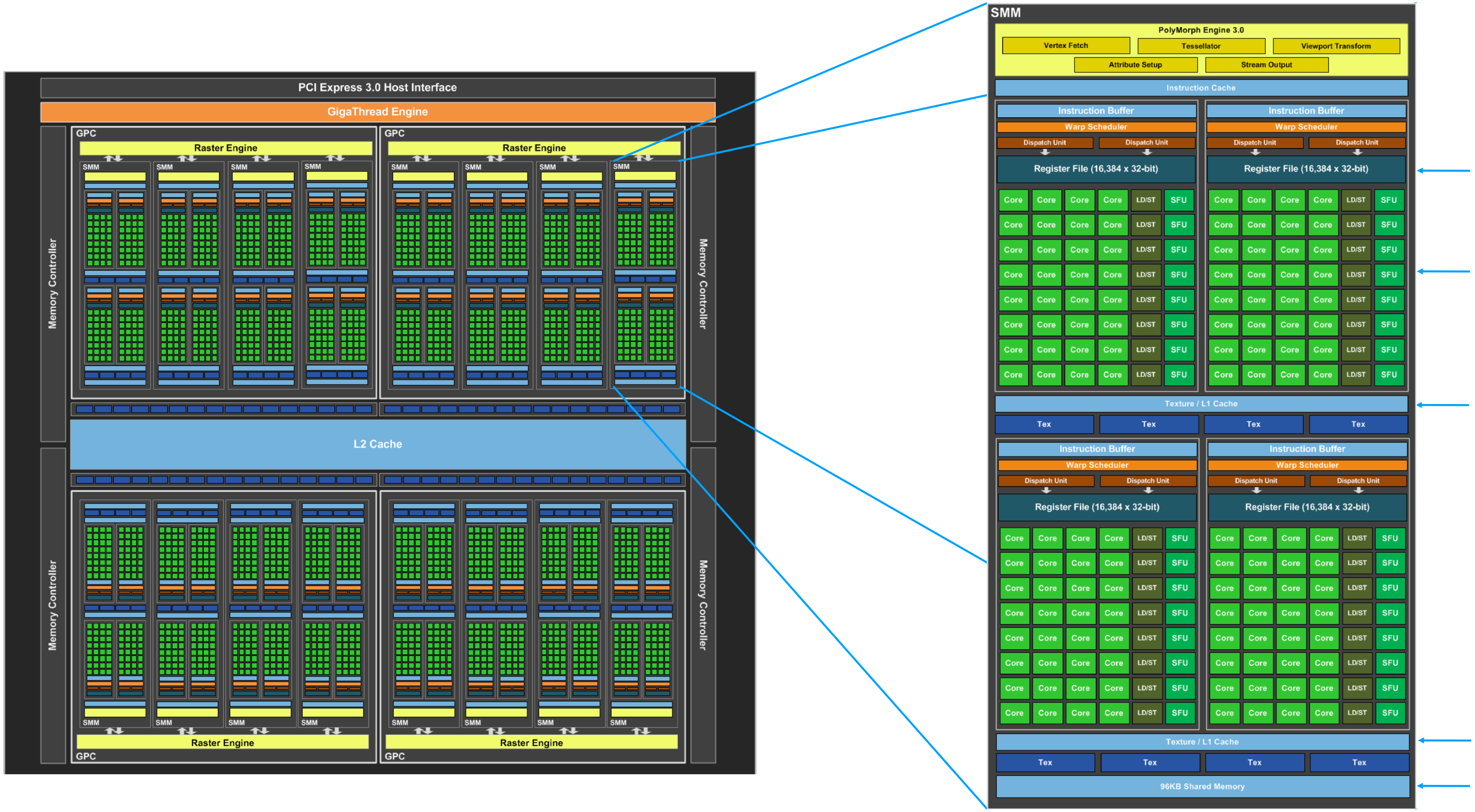
  - This is less efficient, but not always bad.
  - Avoid data-dependent conditional branching if possible

- 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



Register file

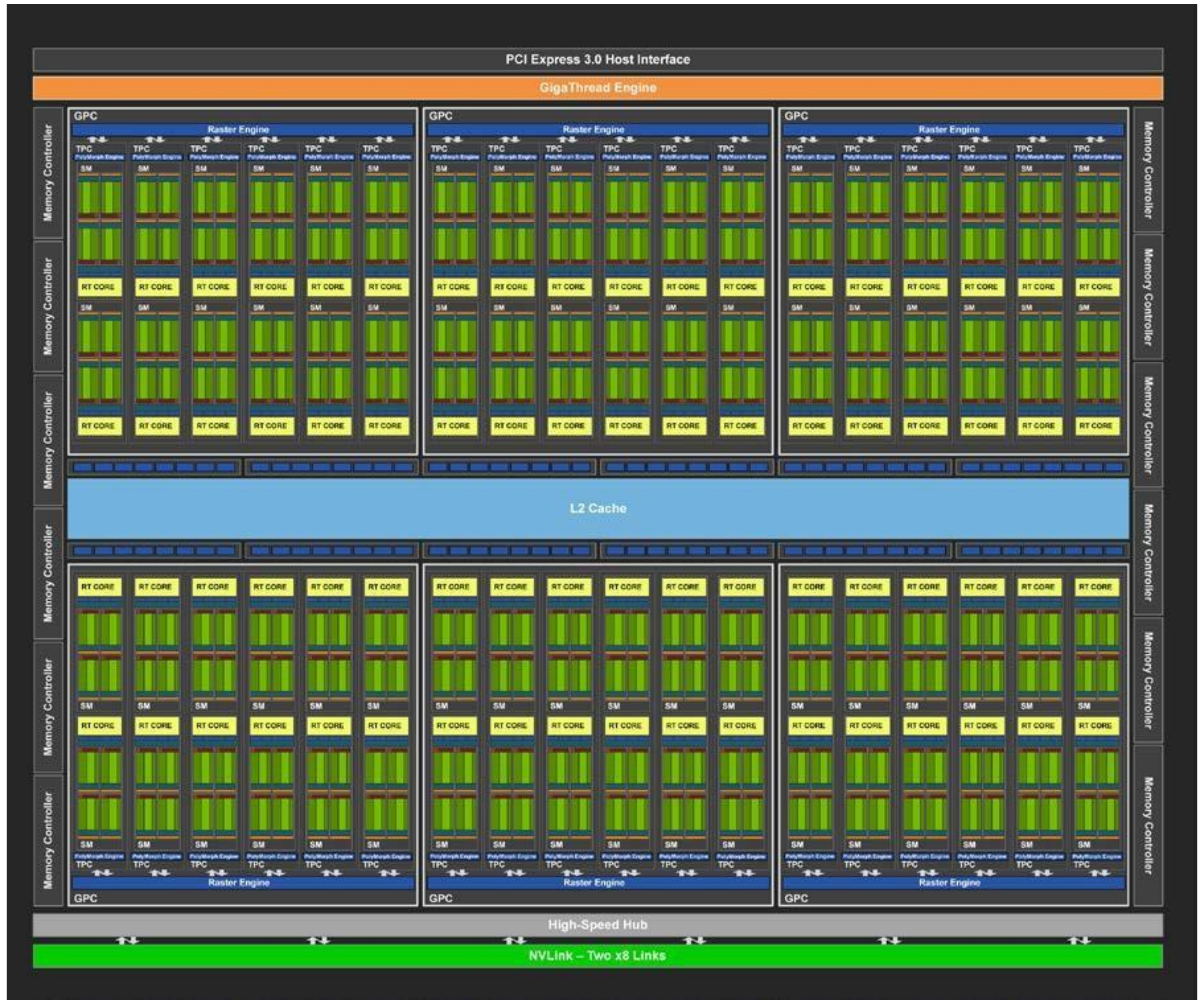
block of 32 cores

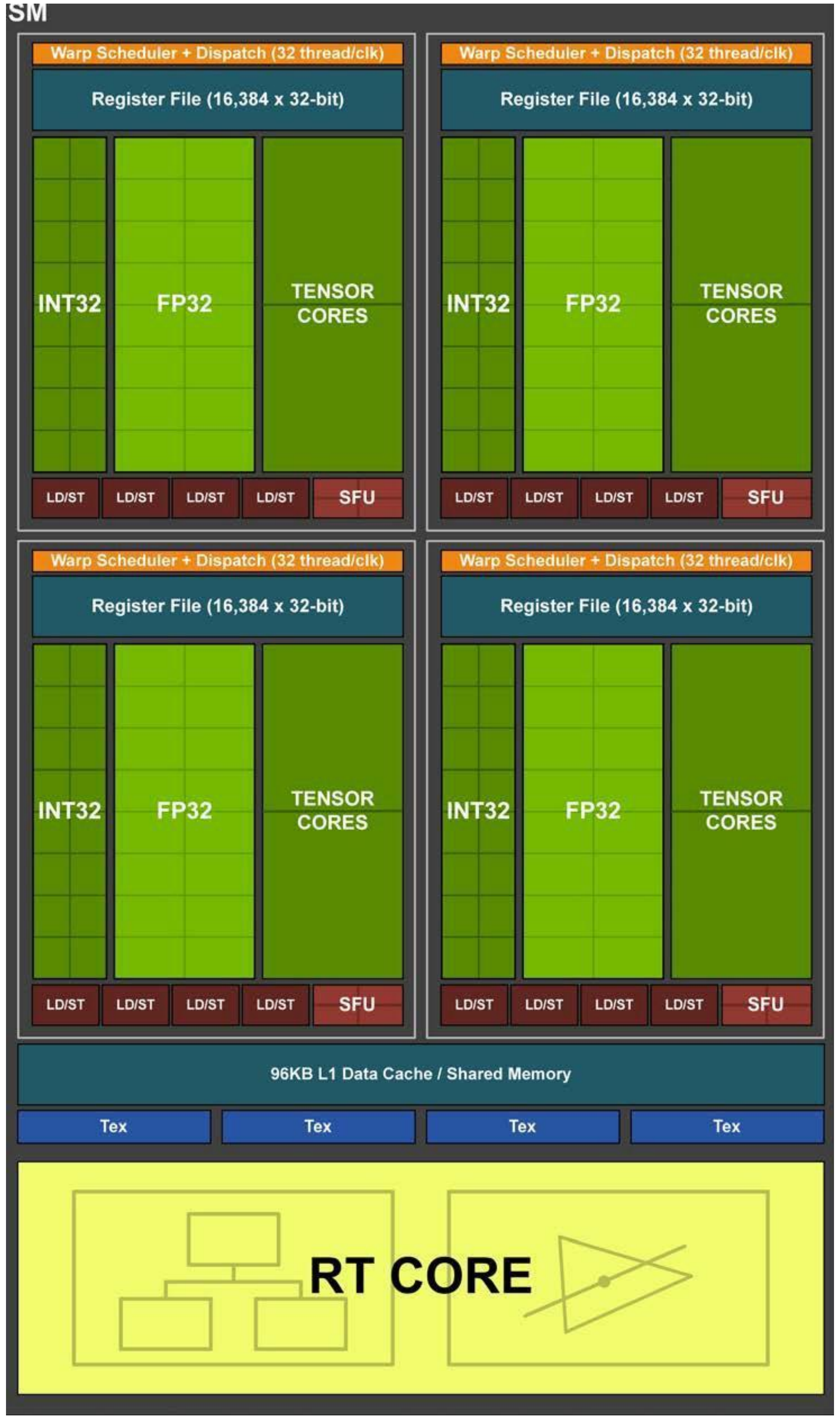
L1 Cache

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

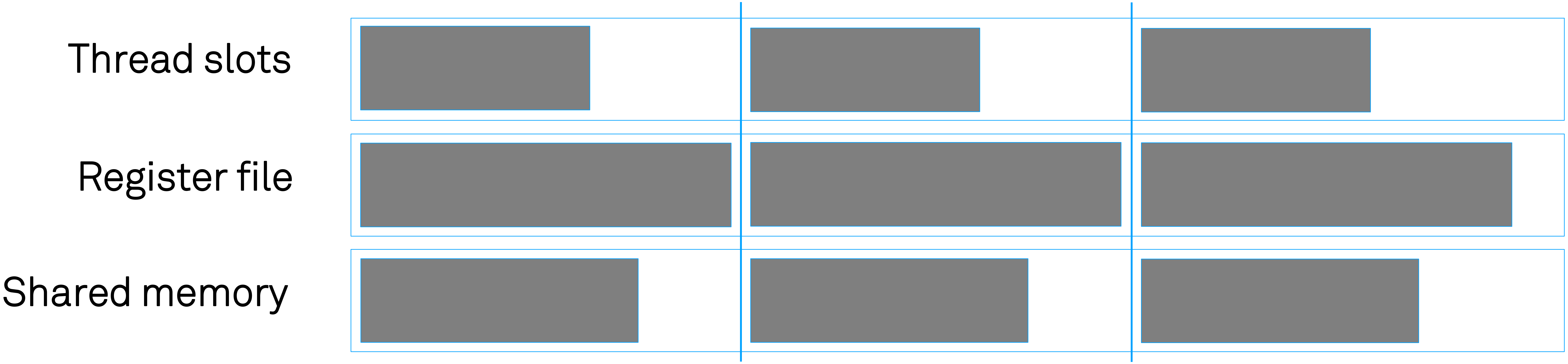



SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Resource partitioning

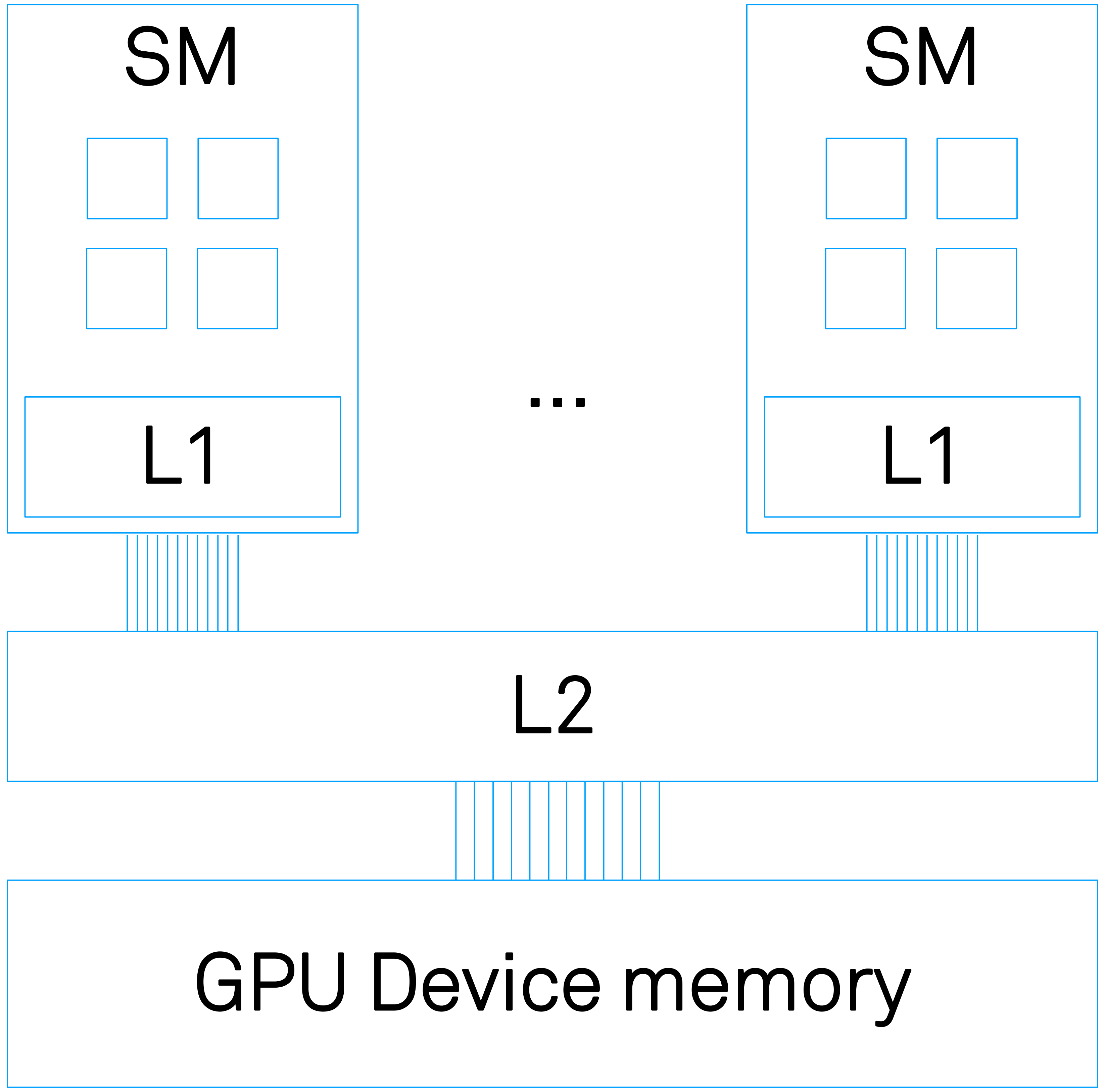
- The GPU consists of several (1 to 68) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Register file, Shared memory, Thread Slots, and Thread Block slots

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



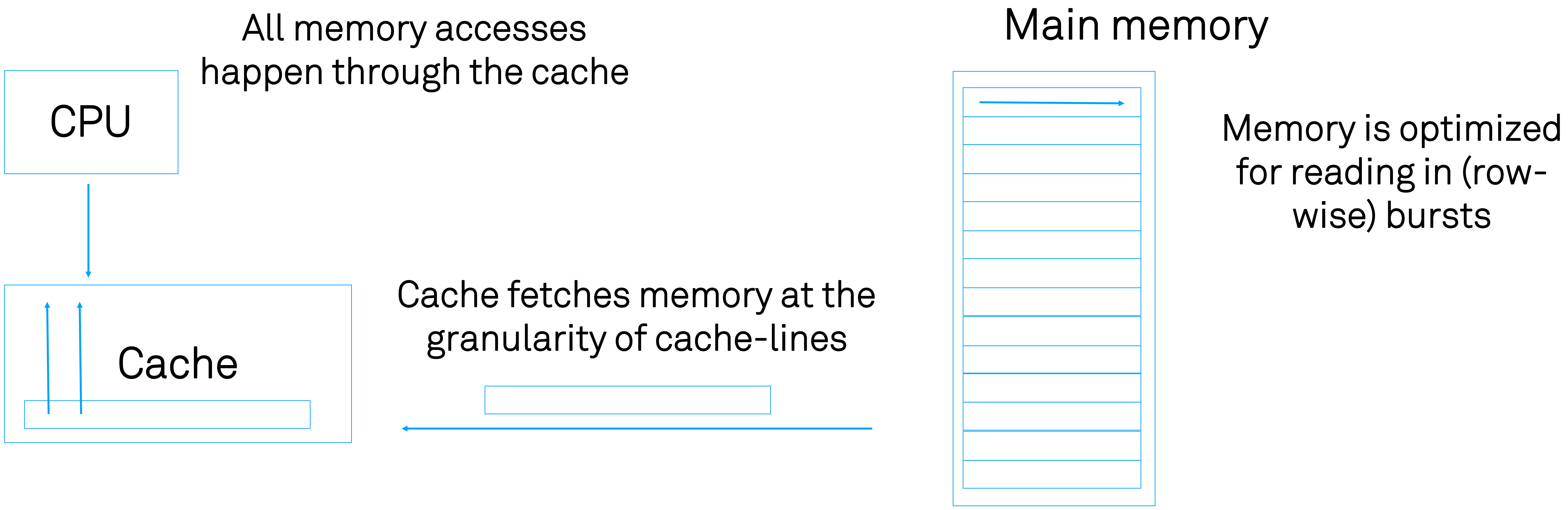
Global Memory access

- 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



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

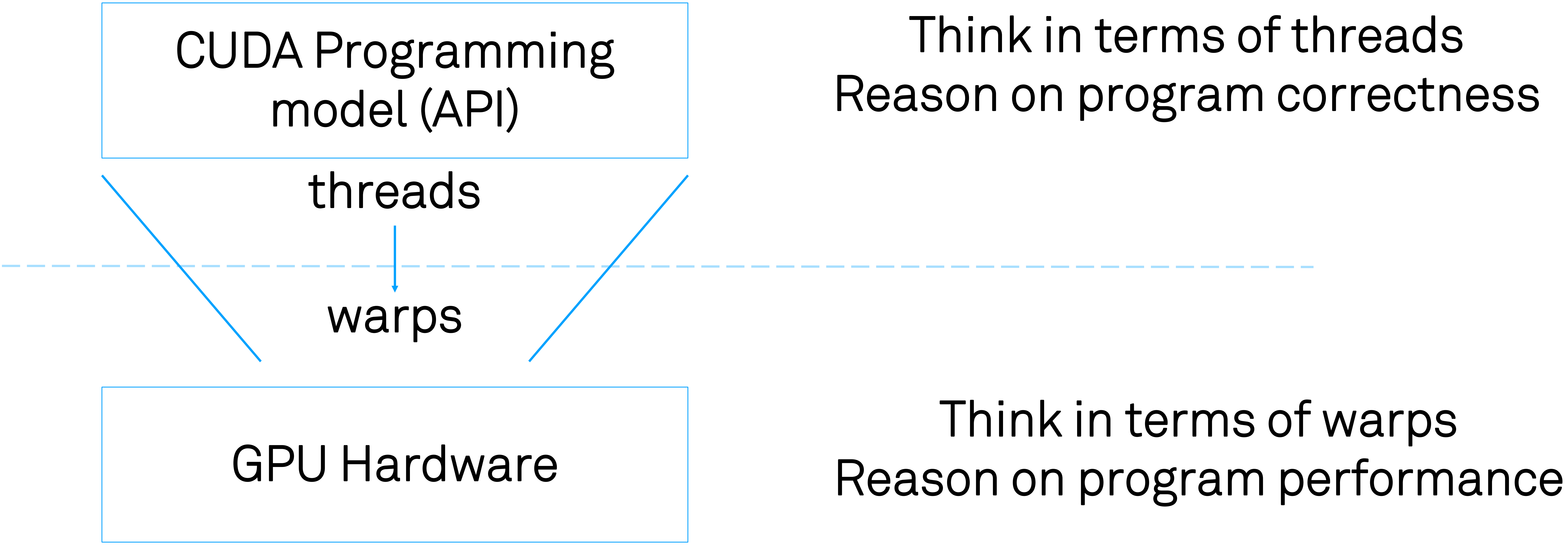
- Moving data around is more expensive than computing on it

- 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



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

Hands-on / Assignment

Hands-on / Assignment

- Go to directory reduction, look at the source files
- Make sure you understand everything in the code

- Task:
  - Implement the kernel such that shared memory is used to sum the per-thread partial sums into a single per-thread block partial sum

- 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 Summation