High-Level GPU Course

November 8 2017
Rob van Nieuwpoort & Ben van Werkhoven

netherlands eScience center

by SURF & NWO

Schedule

- 10:00 – 10:45 Introduction to GPU Computing
- 10:45 – 11:30 High-level intro to GPU Programming
- 11:30 – 12:00 Setup 1st Hands-on Session

- 12:00 – 13:00 Lunch break
- 13:00 – 14:00 Continue working on 1st hands-on

- 14:00 – 15:00 Introduction to CUDA programming and 2nd hands-on
- 15:00 – 16:00 CUDA Program execution

- Additional material for self-study: performance modeling & analysis

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

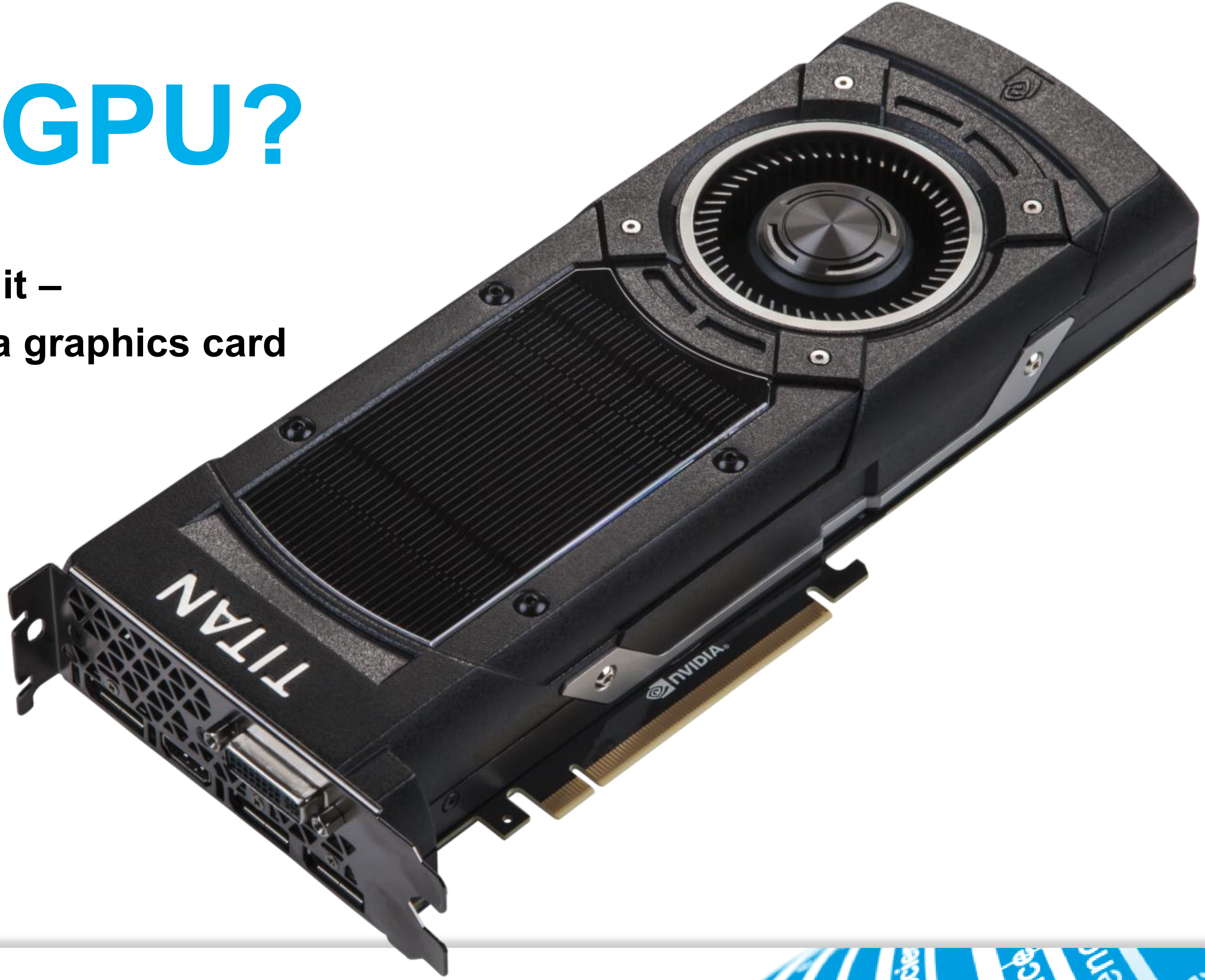
- In code samples on the slides we sometimes leave out ‘{‘ and ‘}’ to save space

Introduction to GPU Computing

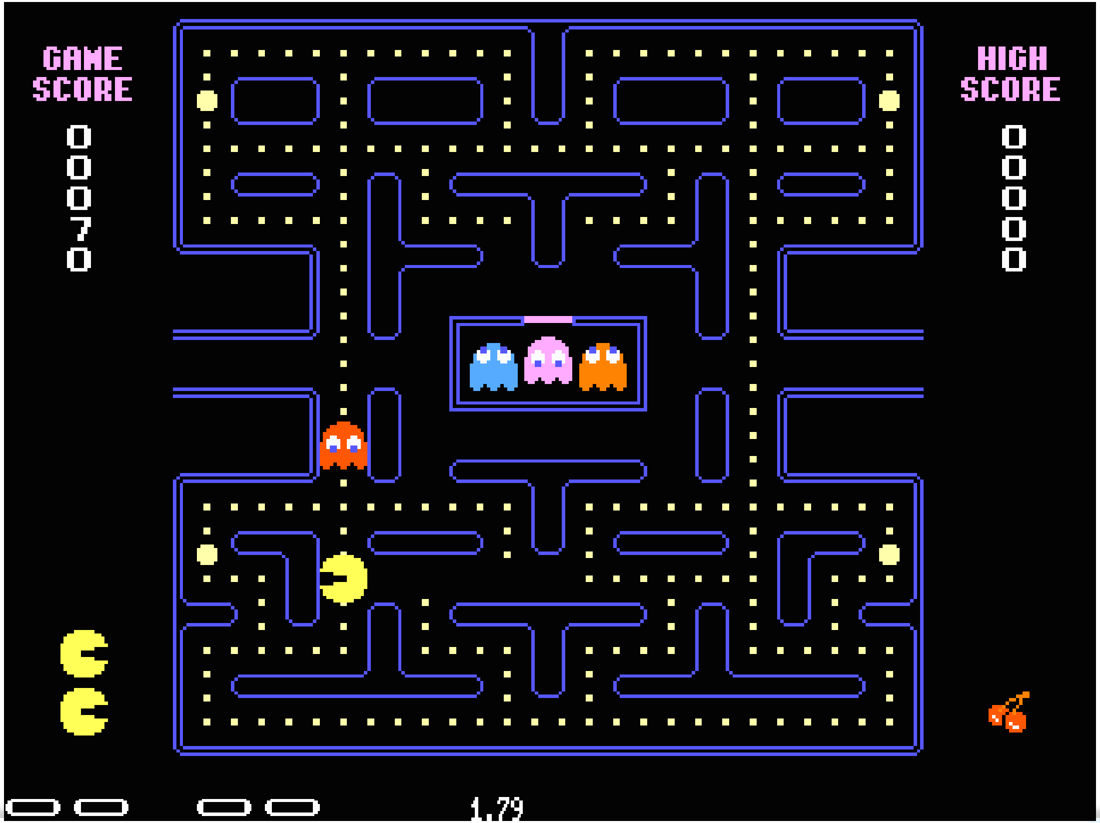
What is a GPU?

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

- GPGPU



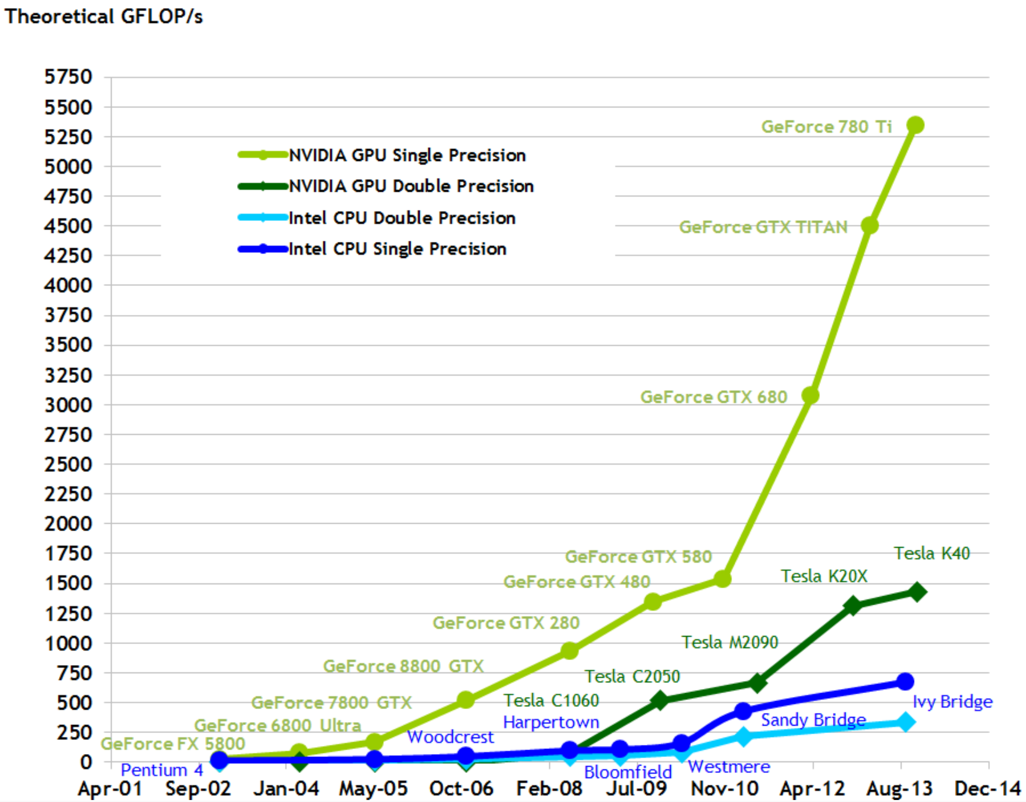
Graphics in 1980



Graphics in 2000

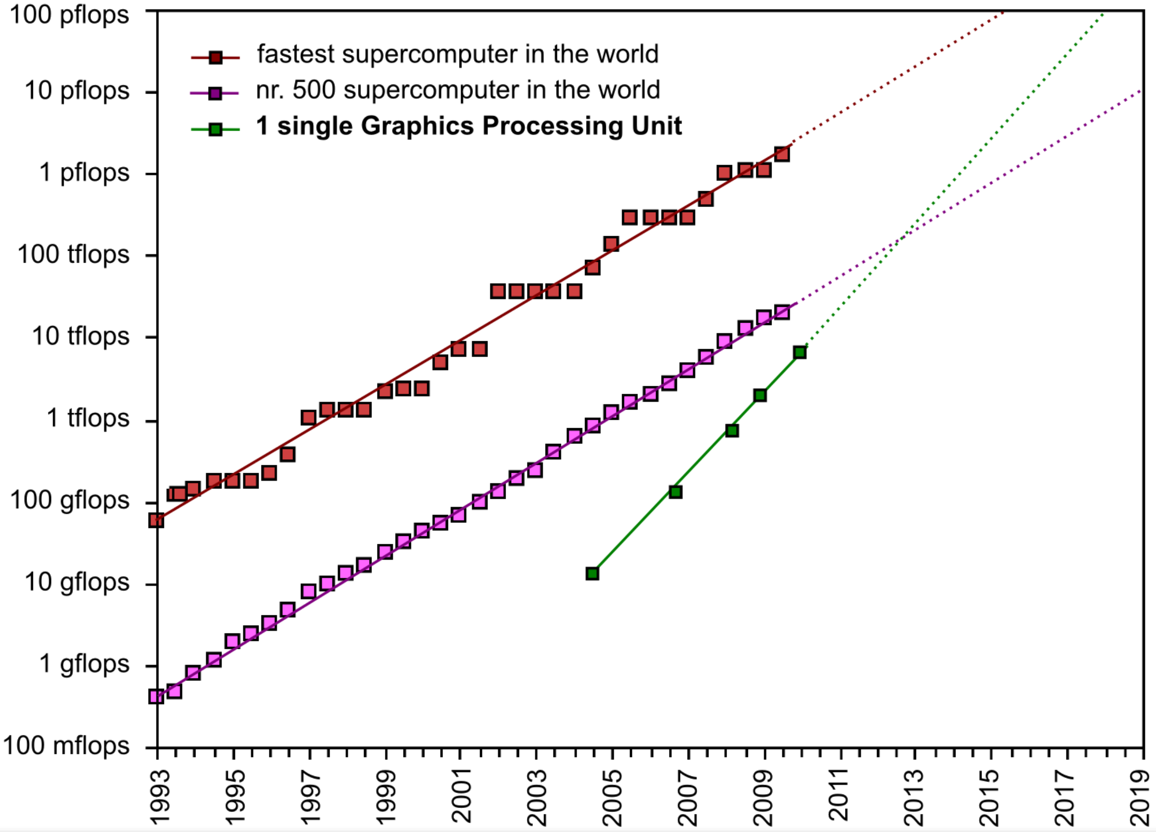
Graphics now

Compute performance per chip



(According to Nvidia)

GPUs vs supercomputers ?

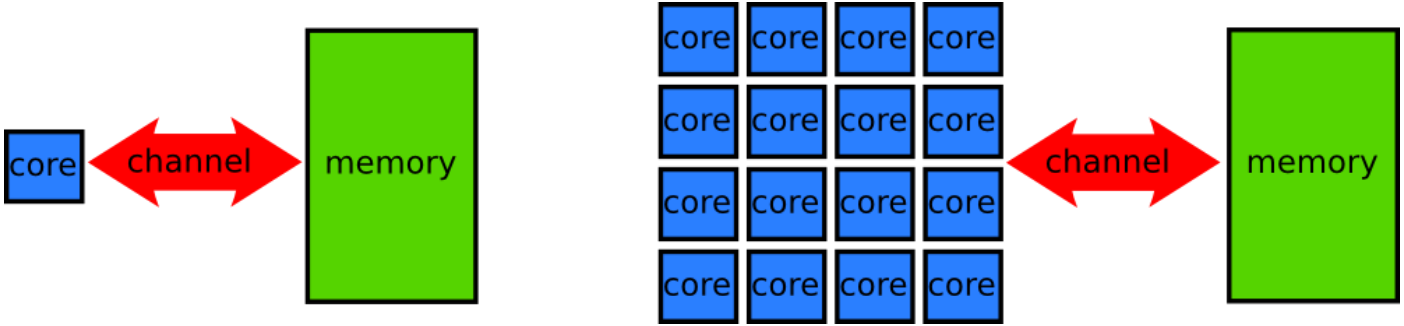


netherlands eScience center

Oak Ridge’s Titan

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

It's all about the memory



Many-core architectures

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

Threads

- The smallest sequence of programmed instructions that can be managed independently by a scheduler
- Lightweight process
- Multiple threads in a process share the same memory and data structures
- Used for doing things in parallel
  - single-core: time slicing
  - multi-core: truly parallel

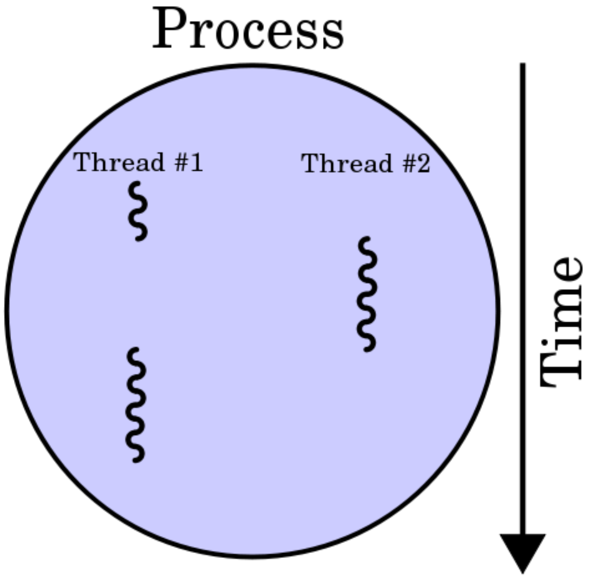
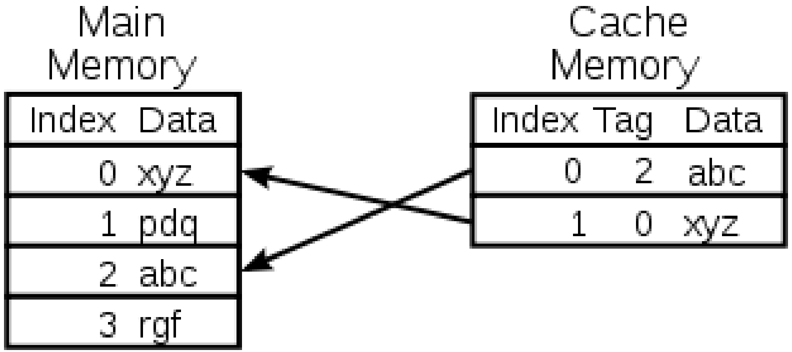


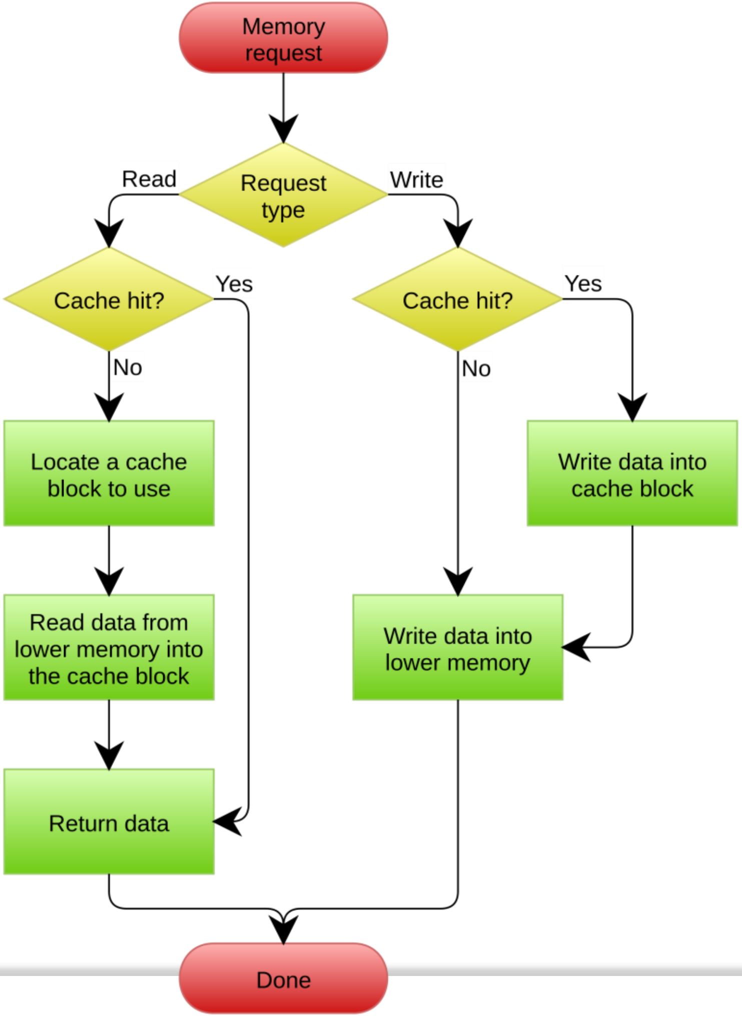
image courtesy Cburnett

Cache

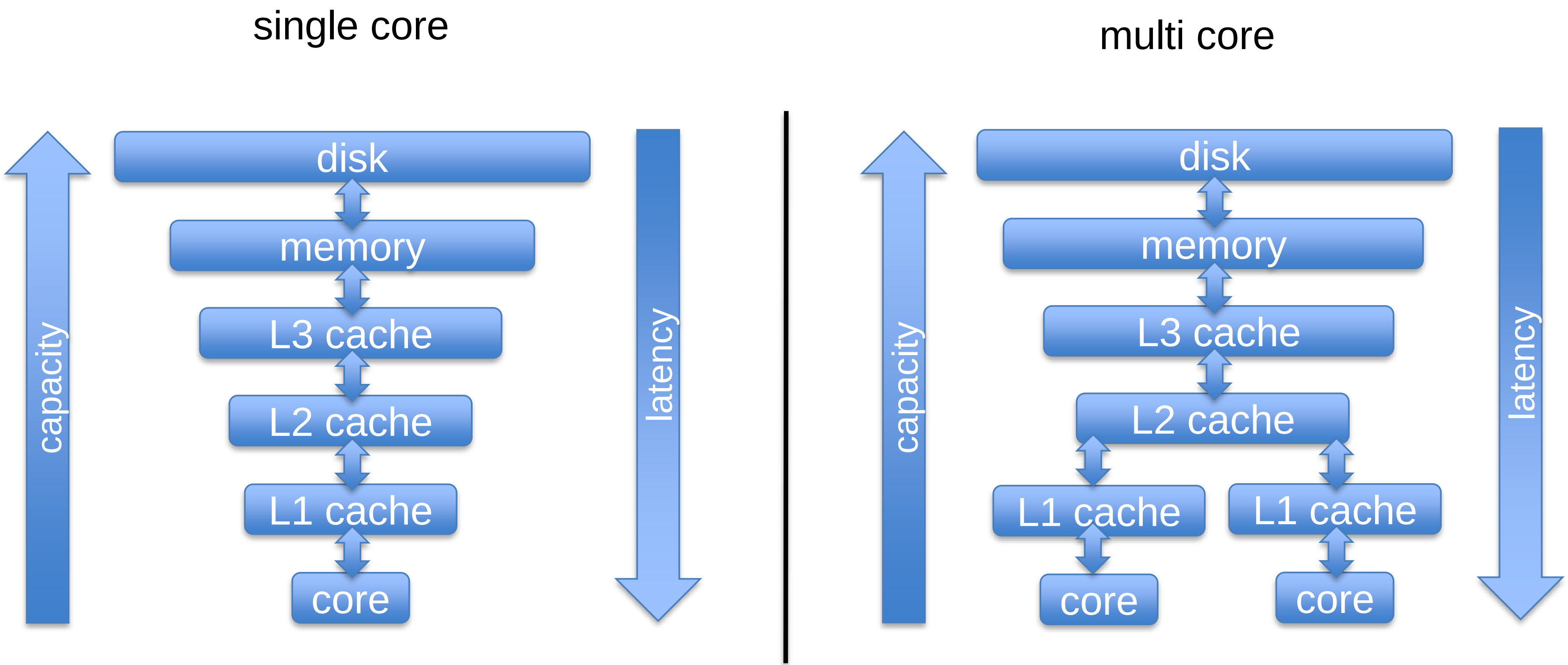
- A small fast piece of memory
- Transparently stores data so that future requests for that same data can be served faster.



Cache


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done

Cache

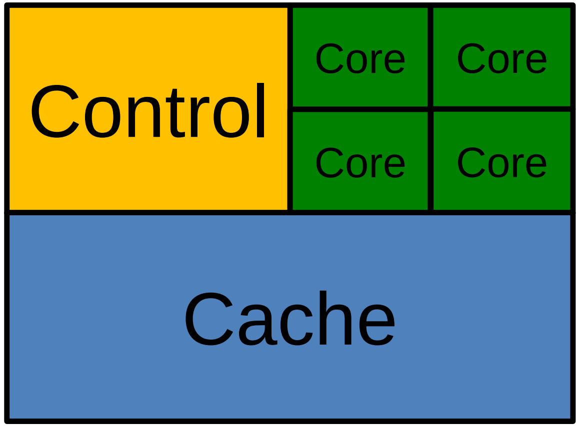

single core
multi core
capacity
disk
memory
L3 cache
L2 cache
L1 cache
core
latency
capacity
disk
memory
L3 cache
L2 cache
L1 cache
L1 cache
core
core
latency

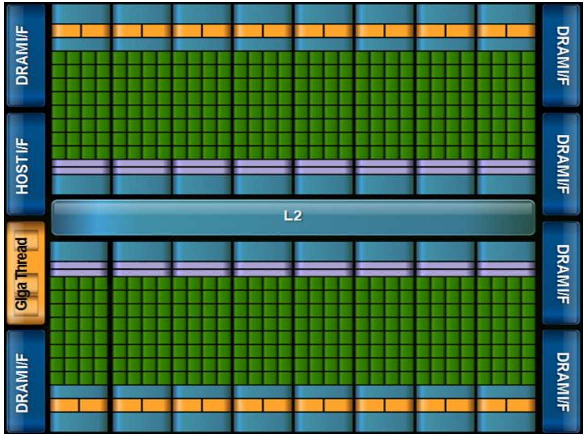
CPU vs GPU Hardware

- Different goals produce different designs
  - CPU must be good at everything, parallel or not
  - GPU assumes work load is highly parallel

- CPU: minimize latency experienced by 1 thread
  - Big on-chip transparent caches
  - Sophisticated control logic

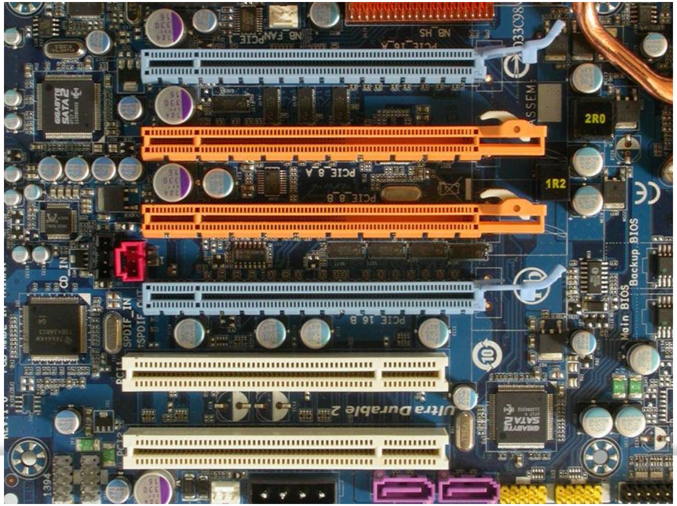
- GPU: maximize throughput of all threads
  - Multithreading can hide latency, so no big caches
  - Control logic
    - Much simpler
    - Less: share control logic across many threads

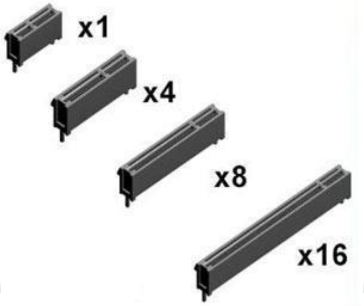


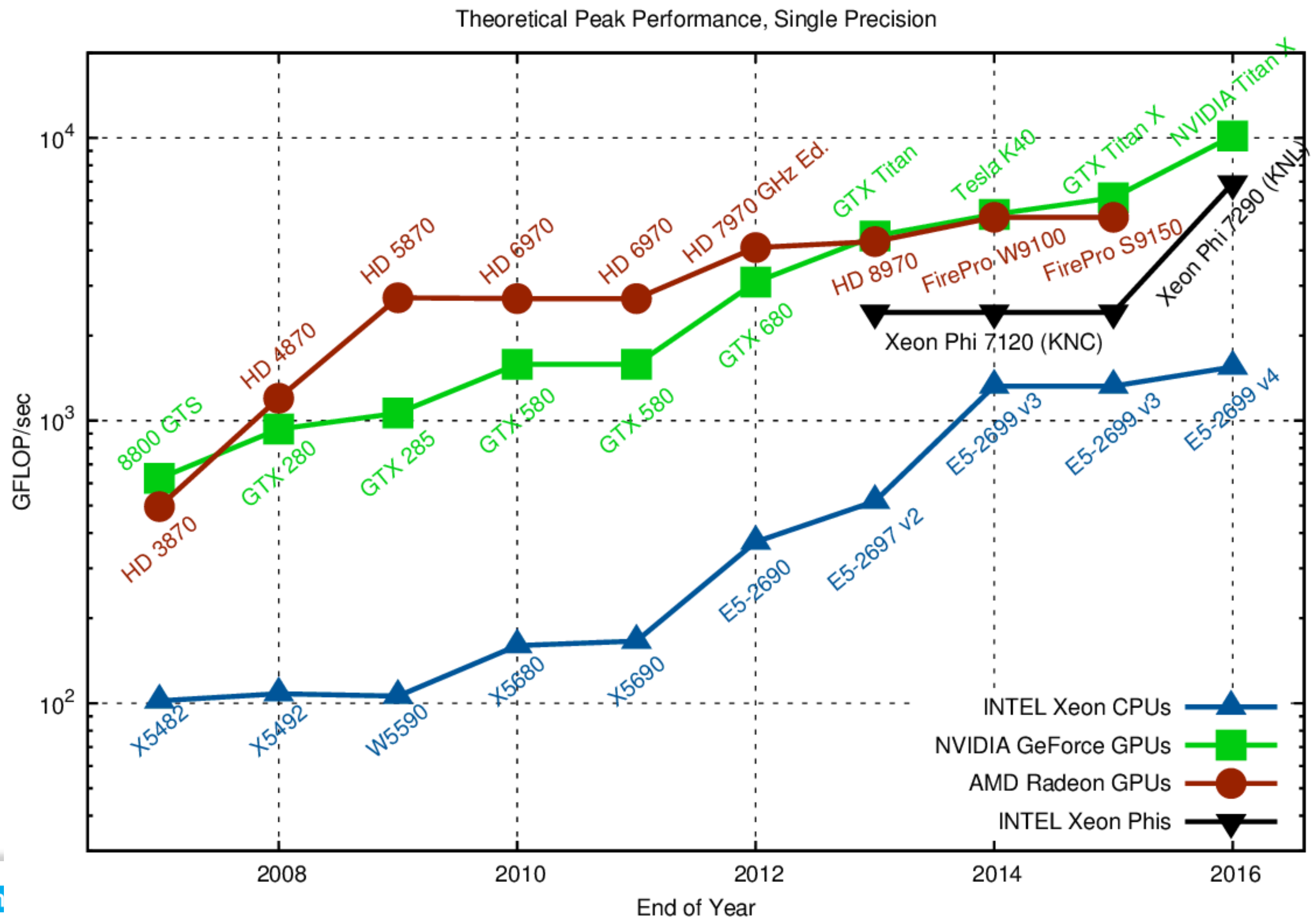


Integration into host system

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

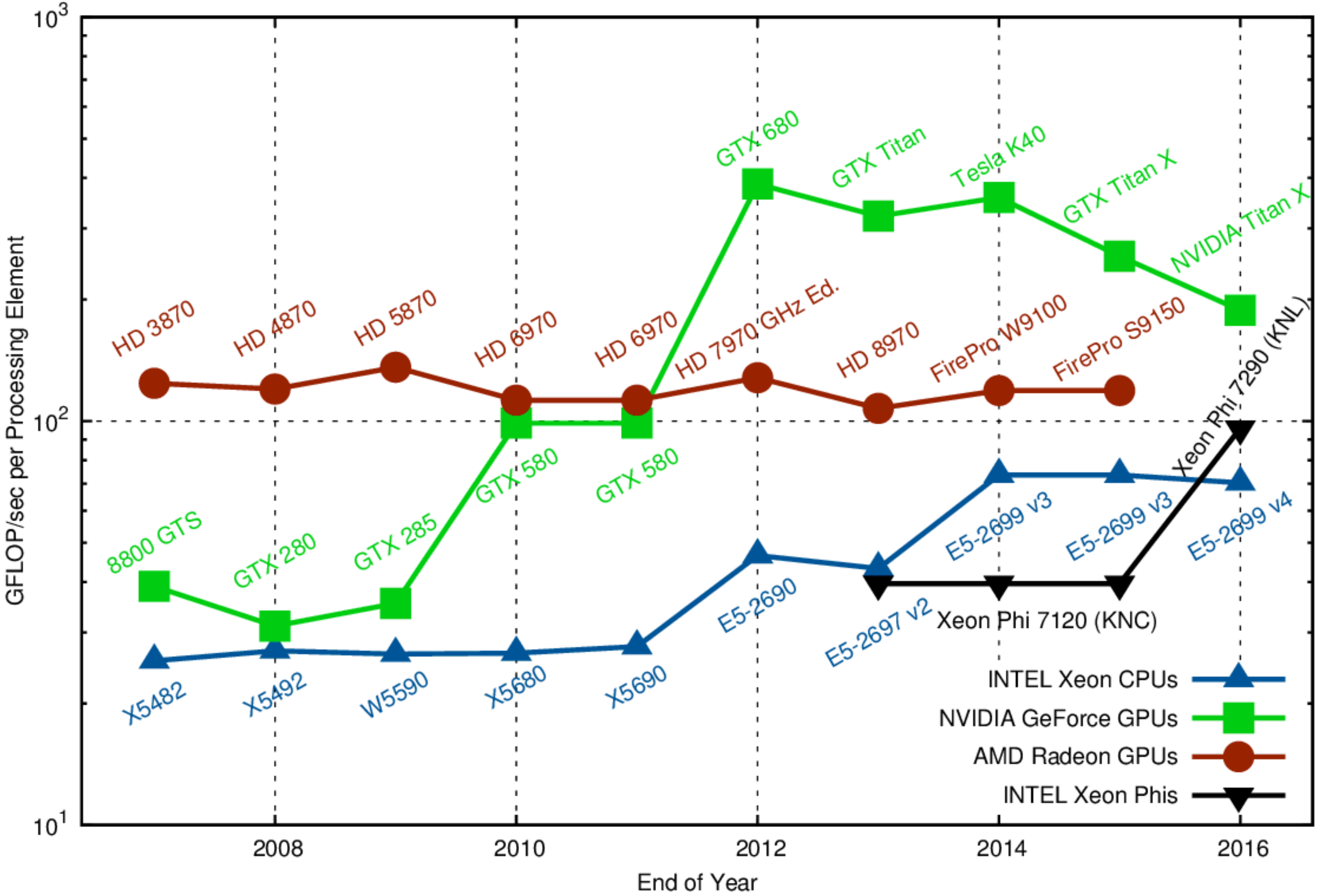


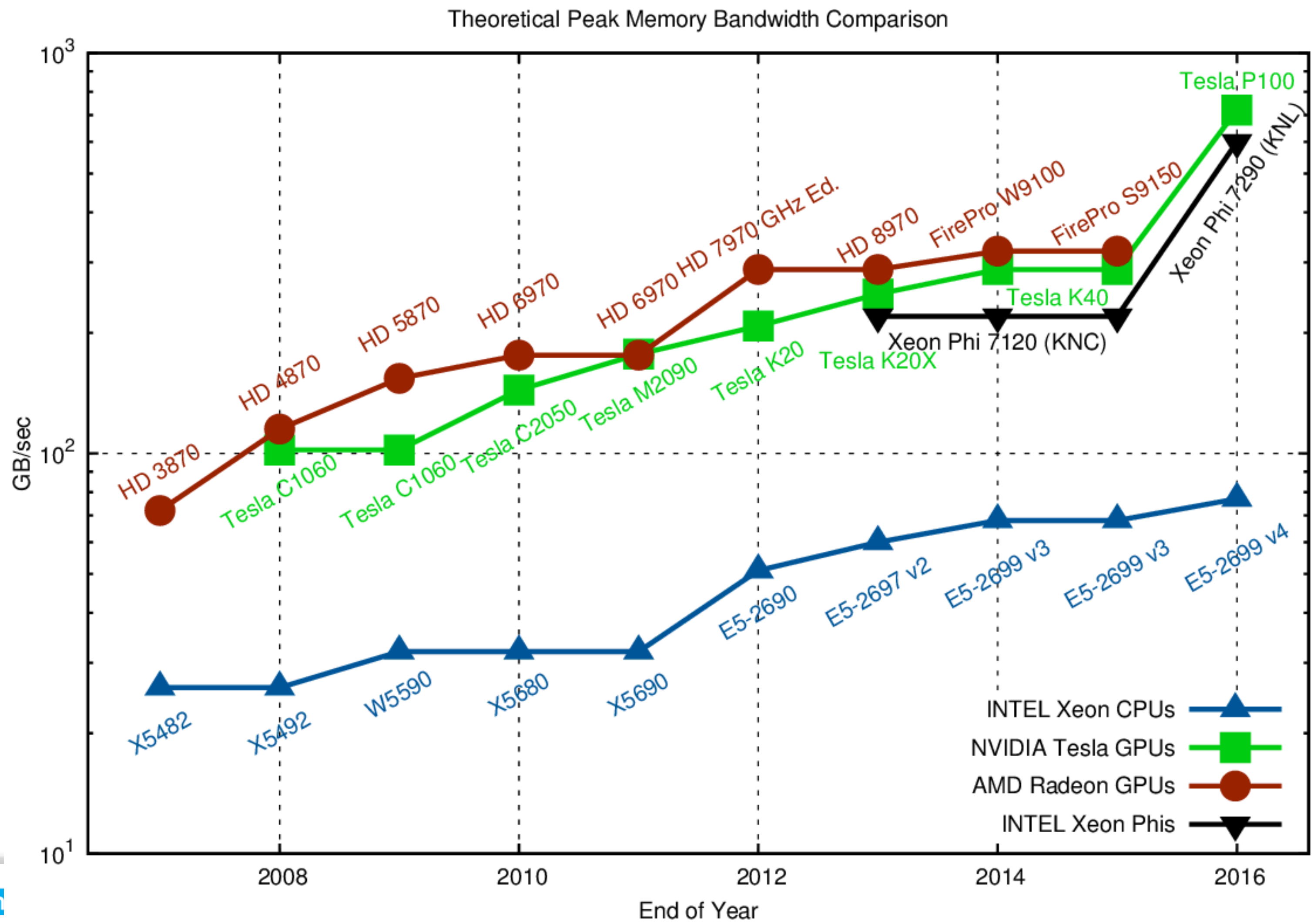


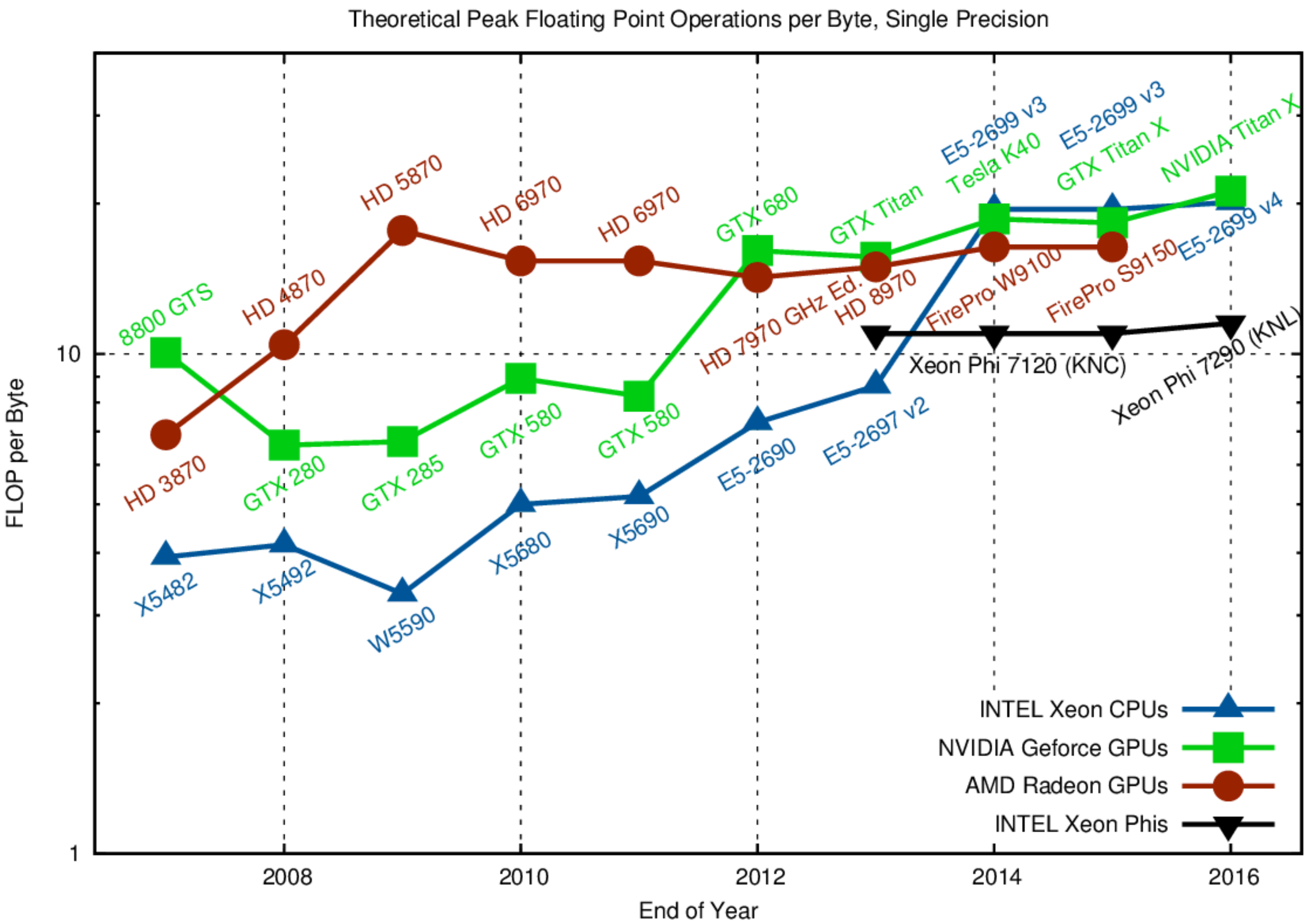


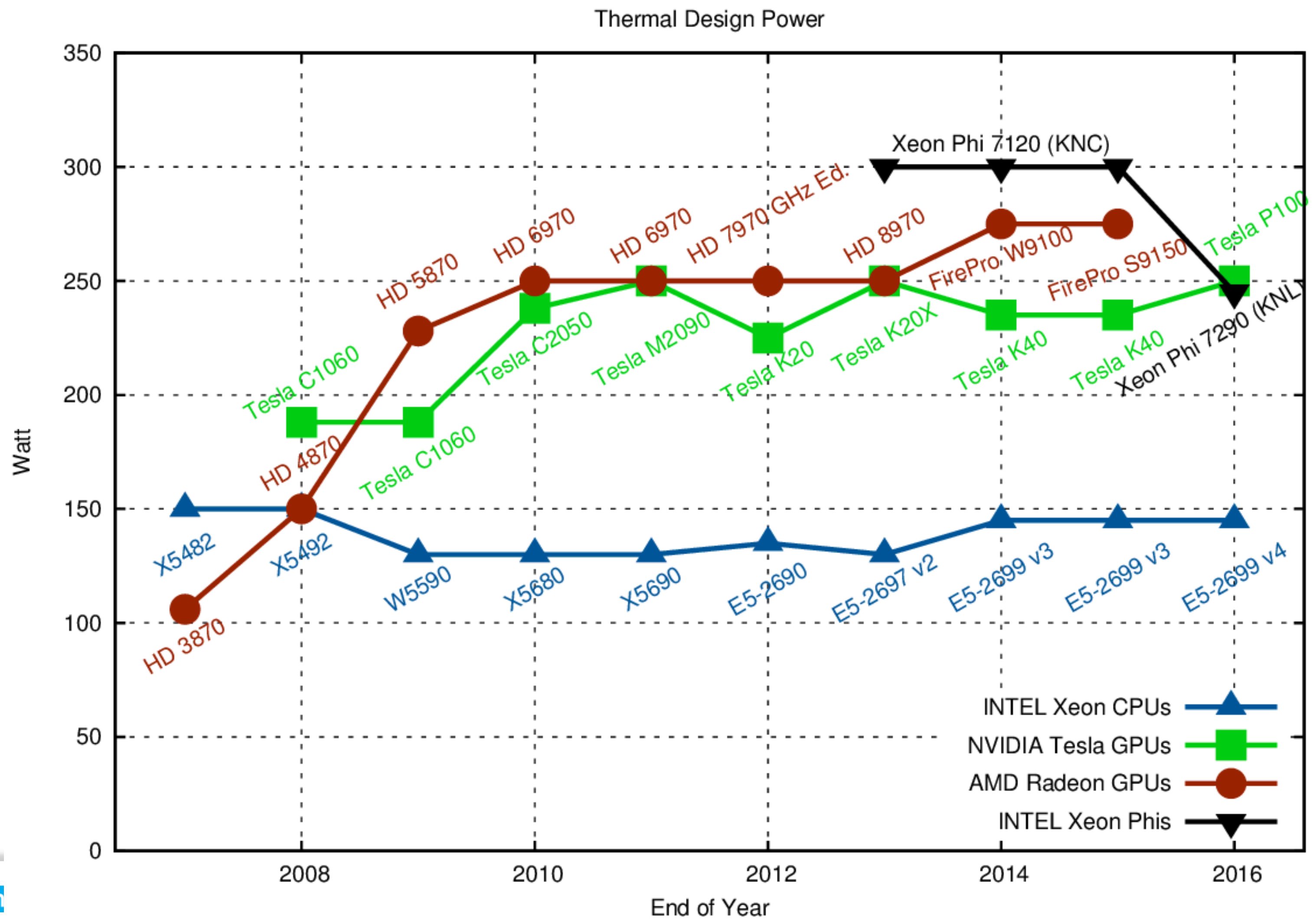
Theoretical Peak Performance, Single Precision
GFLOP/sec
End of Year
10^2
10^3
10^4
2008
2010
2012
2014
2016
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis
8800 GTS
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4

Theoretical Peak Performance per Core/Multiprocessor, Single Precision

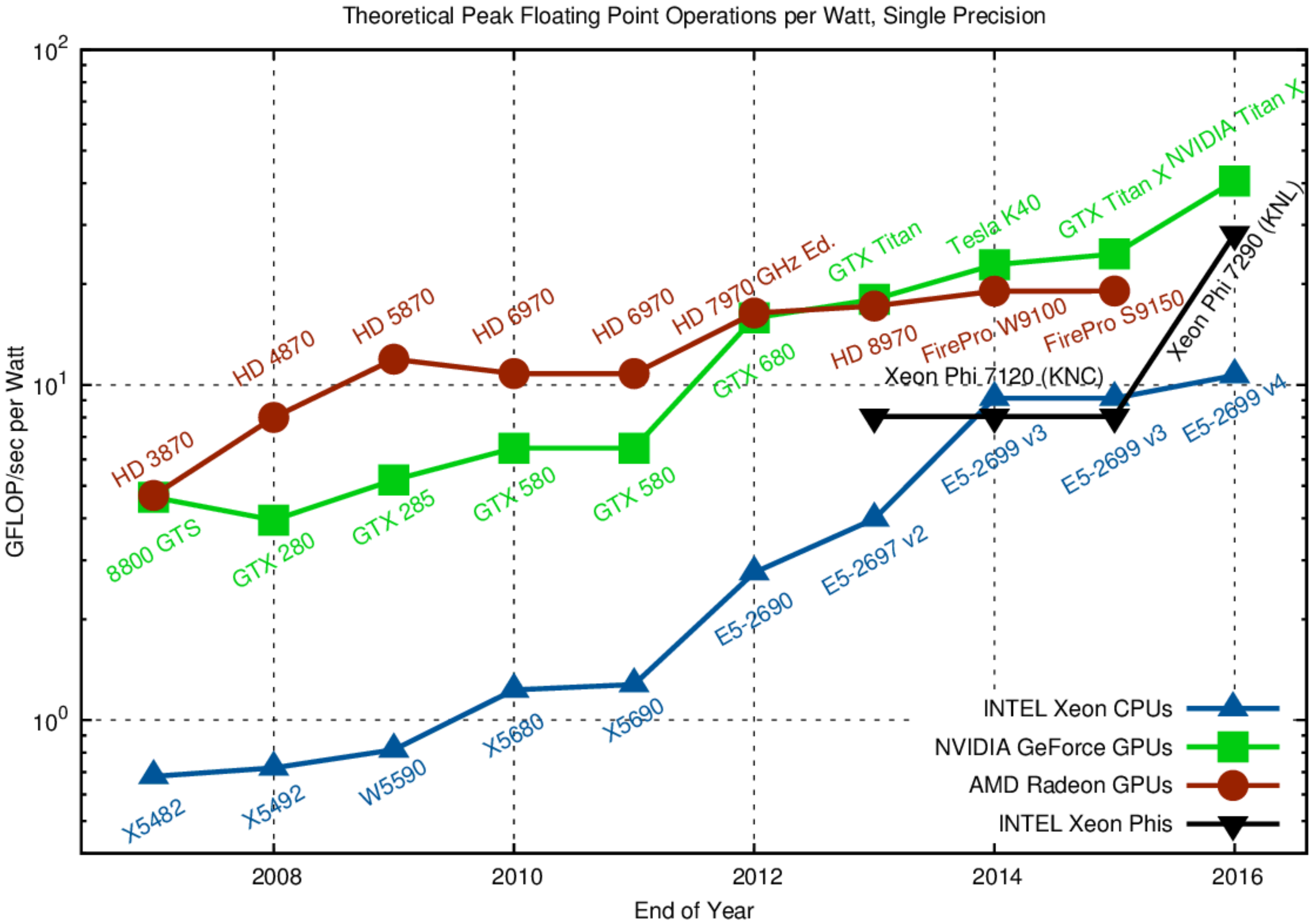



Theoretical Peak Memory Bandwidth Comparison
GB/sec
End of Year
10^1
10^2
10^3
2008
2010
2012
2014
2016
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
Tesla K40
Tesla P100
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Theoretical Peak Floating Point Operations per Byte, Single Precision
FLOP per Byte
End of Year
10
1
2008
2010
2012
2014
2016
8800 GTS
HD 4870
HD 5870
HD 6970
HD 6970
GTX 680
GTX Titan
E5-2699 v3
Tesla K40
E5-2699 v3
GTX Titan X
NVIDIA Titan X
E5-2699 v4
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
HD 3870
GTX 280
GTX 285
GTX 580
GTX 580
E5-2690
E5-2697 v2
X5482
X5492
W5590
X5680
X5690
INTEL Xeon CPUs
NVIDIA Geforce GPUs
AMD Radeon GPUs
INTEL Xeon Phis



Thermal Design Power
Watt
End of Year
0
50
100
150
200
250
300
350
2008
2010
2012
2014
2016
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
Tesla K40
Tesla K40
Tesla P100
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis



netherlands
eScien

Why GPUs?

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

When use GPU Computing?

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

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

A high-level introduction to GPU Programming

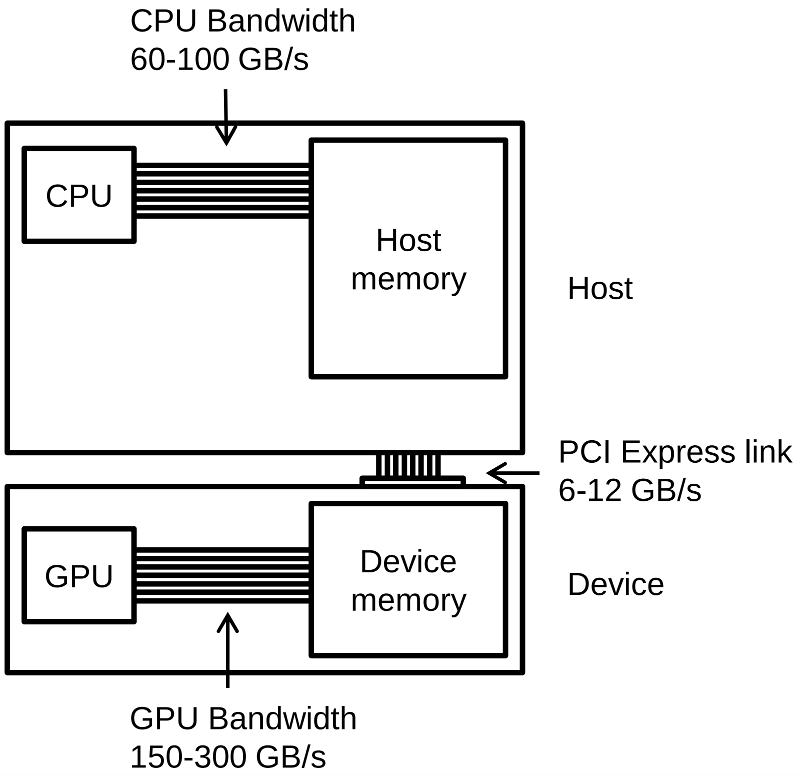
Why is GPU Programming different?

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

- Several language bindings for GPU Programming exist:
  - Python: PyCuda and PyOpenCL for CUDA and OpenCL programming
  - Java: JCuda and JOCL
  - Fortran: CudaFortran
  - Matlab: MexCuda (using mexfiles)
- However, these only cover the *host* part of the program. Kernels have to be written in a language that can be compiled to device code
- Solutions for the device part of the program:
  - 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)

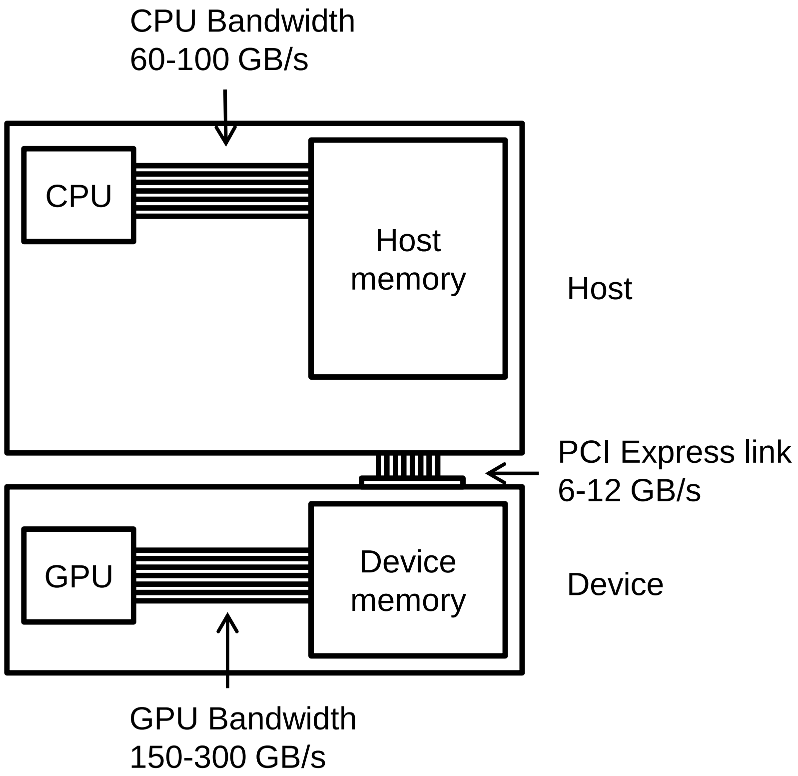
- In general it's best to keep working on transferred data for as long as possible

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

Summary

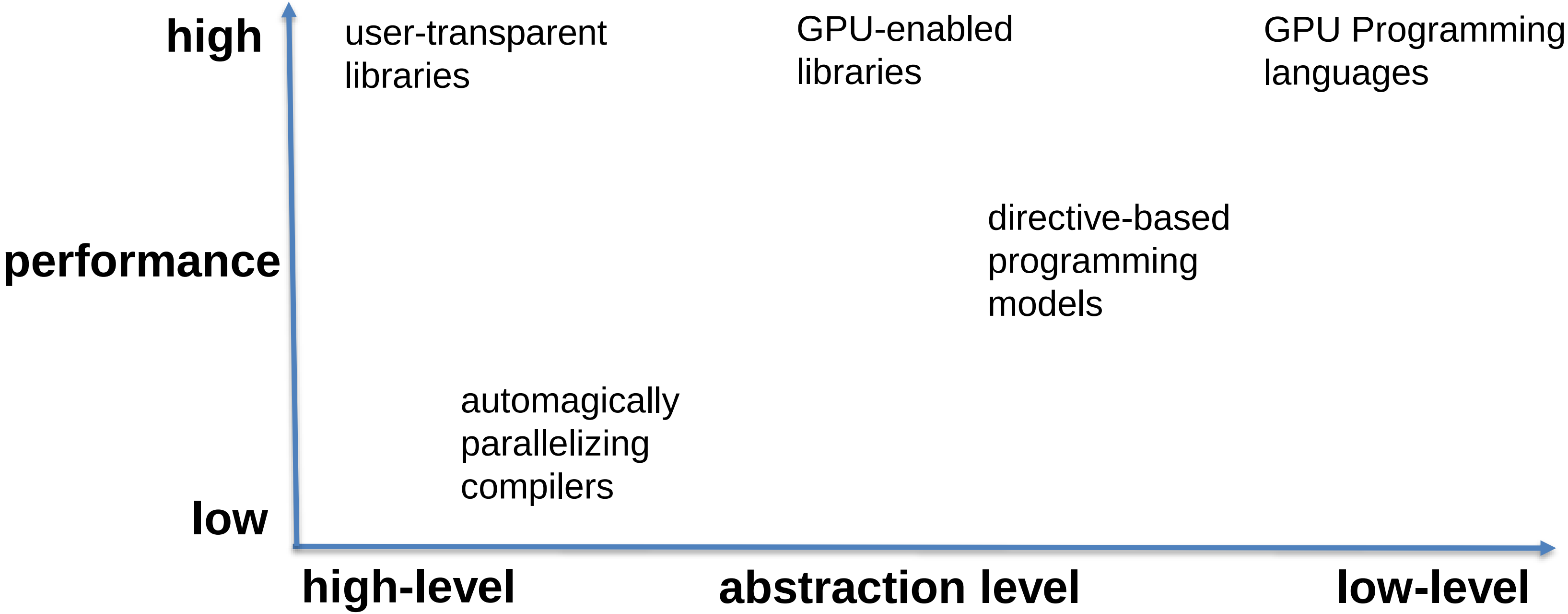
Main differences normal and GPU programming:

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**

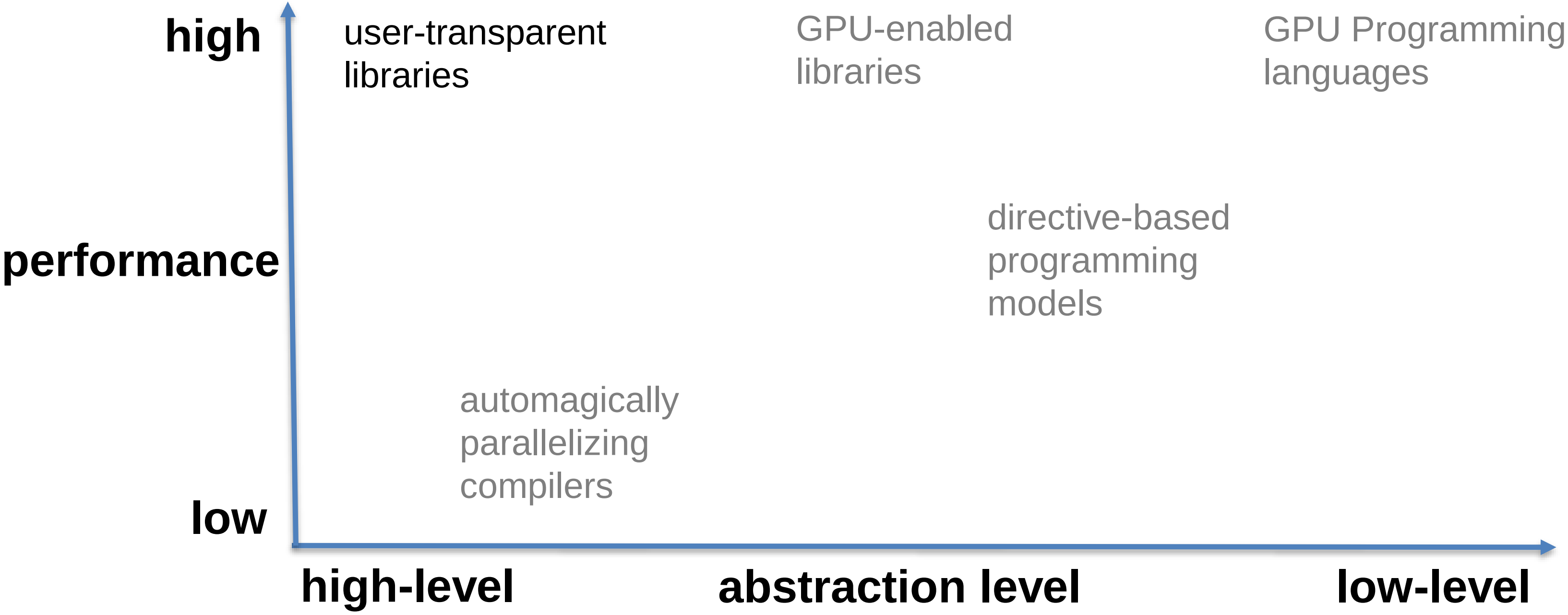


Overview of GPU programming technologies

GPU Programming techniques



GPU Programming techniques

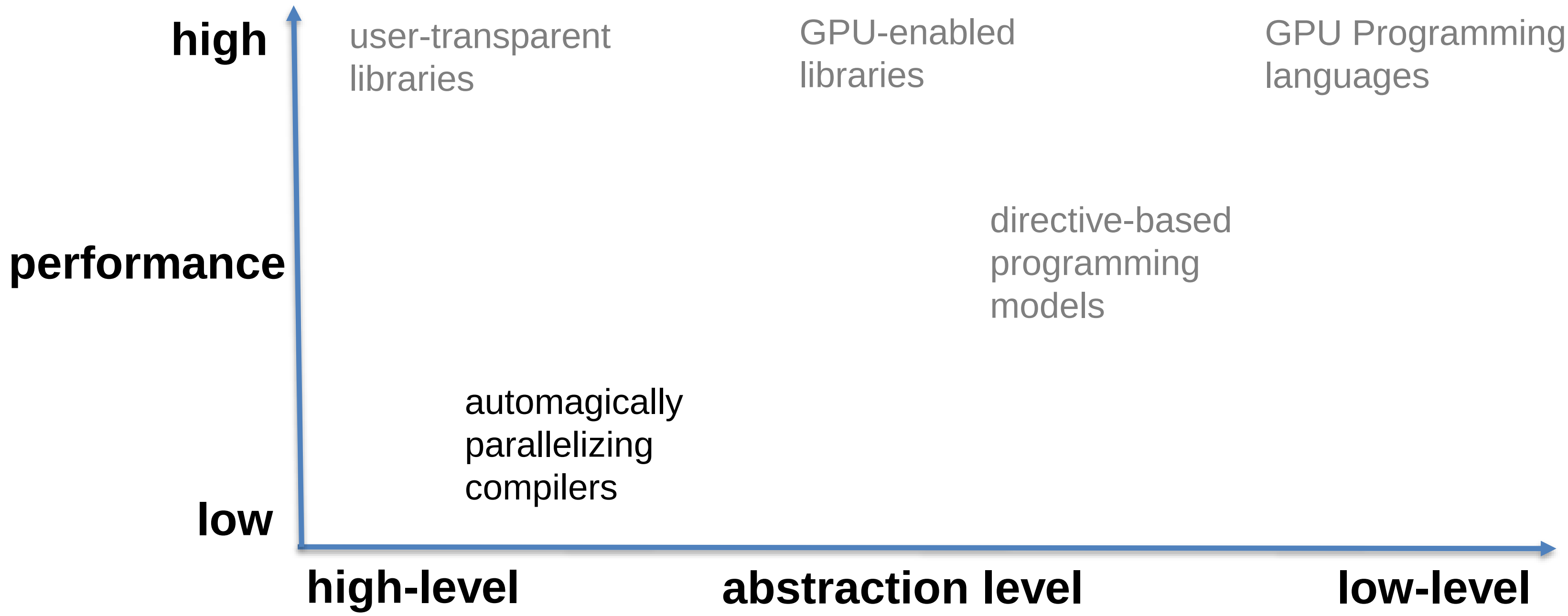


User-Transparent libraries

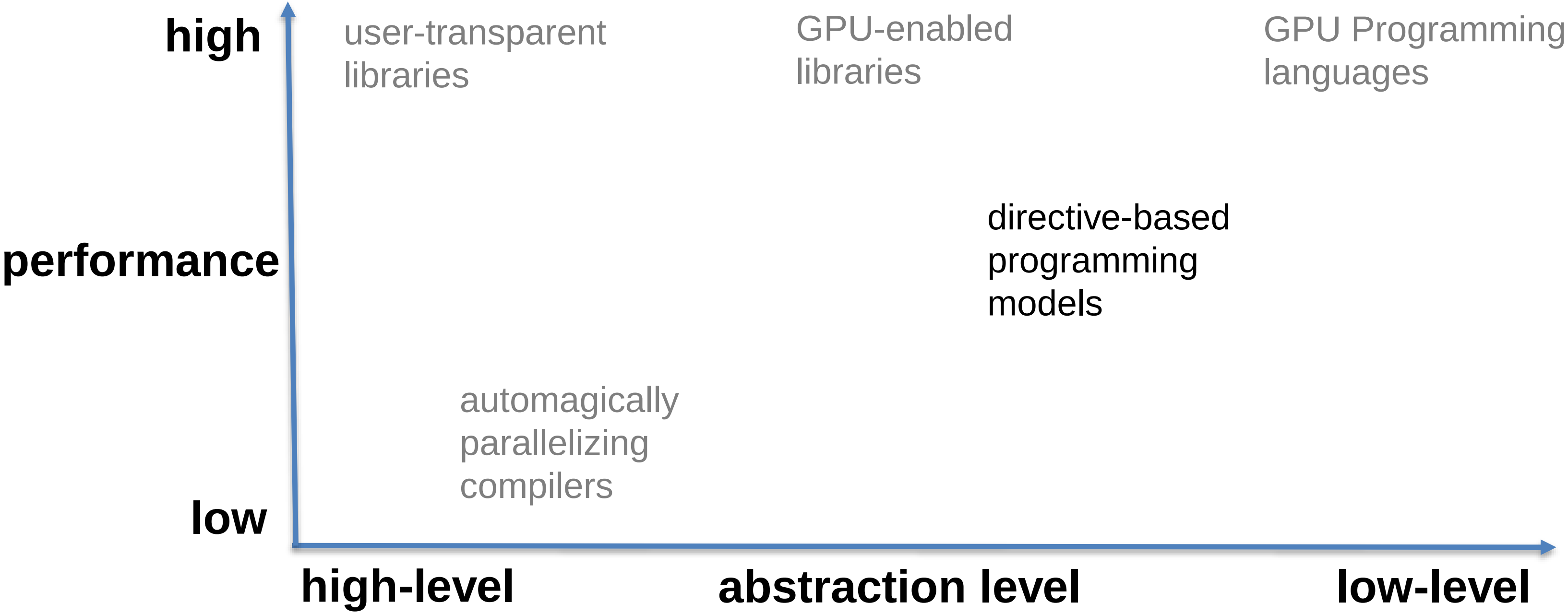
- Act as drop-in replacements for CPU libraries
- There aren’t that many, and their application is often limited

- 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



GPU Programming techniques



Directive-based approaches

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.

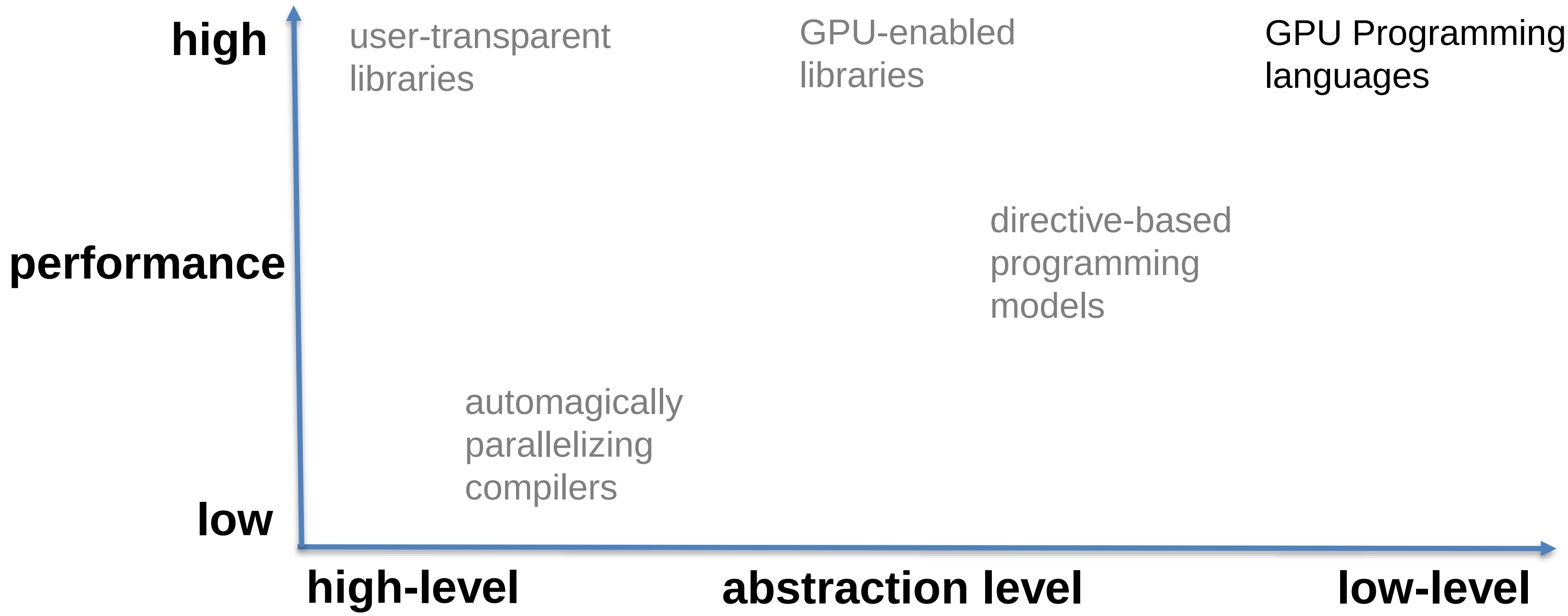
Compiler support

- Commercial compilers:
  - PGI (as of version 12.6), Cray, and CAPS

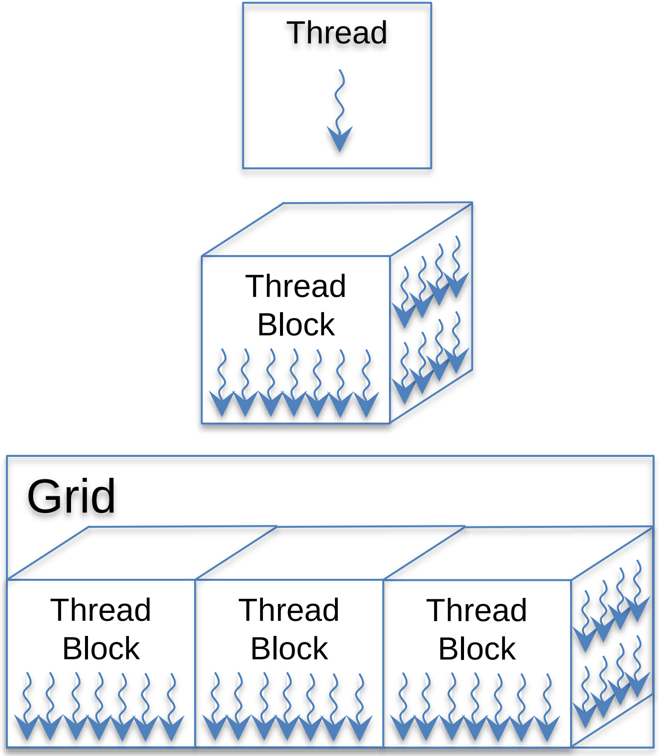
- ‘Research’ compilers (developed by universities):
  - OpenUH, OpenARC, accULL

- Open compilers:
  - GCC (OpenACC 1.0, as of version 5) (OpenACC 2.0 as of version 6)
    I believe OpenACC through GCC only works with Nvidia GPUs.
    For instructions on how to configure GCC for OpenACC see:
    http://scelementary.com/2015/04/25/openacc-in-gcc.html

GPU Programming techniques



Writing GPU kernels

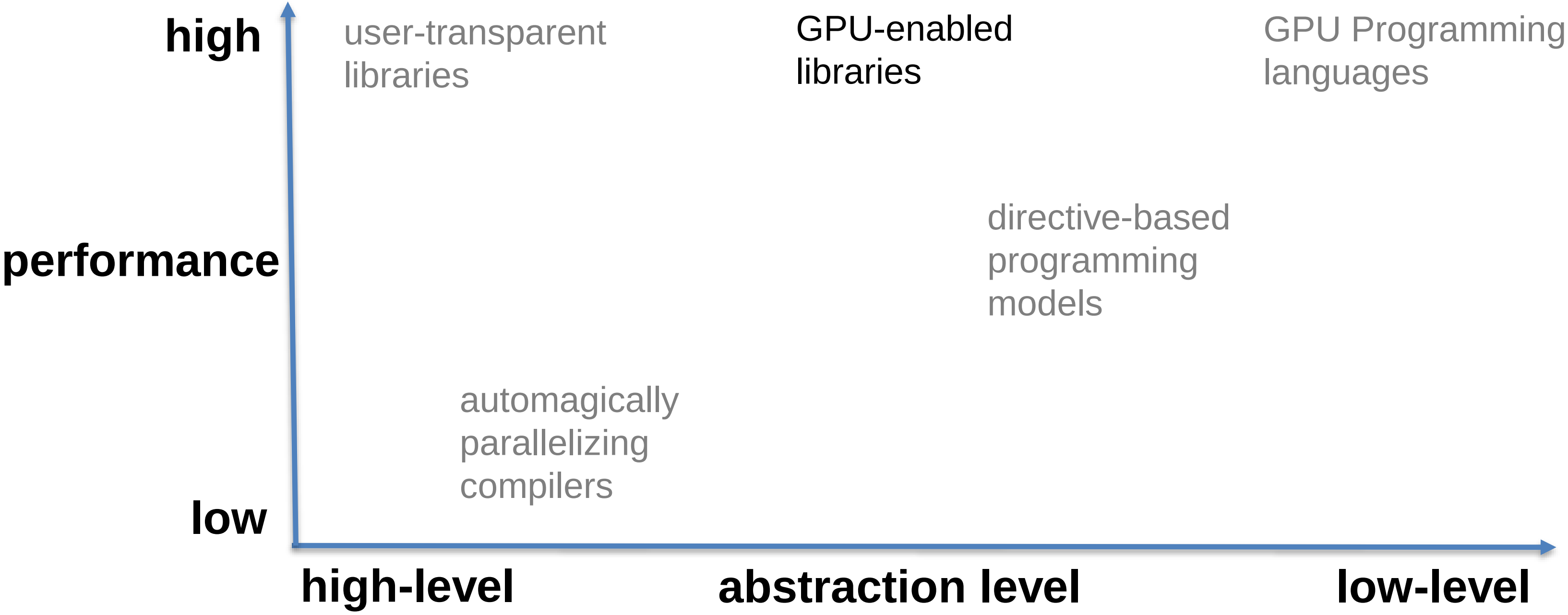


Registers

Shared memory

Global memory
Constant memory

GPU Programming techniques



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

Setup hands-on sessions

- **Open a terminal and connect to** `csngpu1.science.uva.nl` **and type:**
  - `module load matlab`
  - `` export MATLAB_EXECUTABLE=`which matlab` `` **(note: those are backticks)**
  - `export CUDA_CACHE_MAXSIZE=1073741824`
  - `git clone https://github.com/benvanwerkhoven/gpu-course.git`
  - `cd gpu-course/matlab/`
  - `jupyter notebook -no-browser`
- **Setup SSH tunnel to the server**
  - **Linux:** `ssh -N -f -L localhost:8000:localhost:8888 username@csngpu1.science.uva.nl`
    Putty: http://realprogrammers.com/how_to/set_up_an_ssh_tunnel_with_putty.html
    - **Destination** `localhost:8888`, **source port:** `8000`, **click Add, Save the session, connect**
- **Open your browser and navigate to** http://localhost:8000
  - **you need to copy the token printed by the notebook server to login**
- **Open the** `GPU_FFTs_in_Matlab.ipynb` **notebook**

Lunch break & hands-on

we continue at 14:00

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

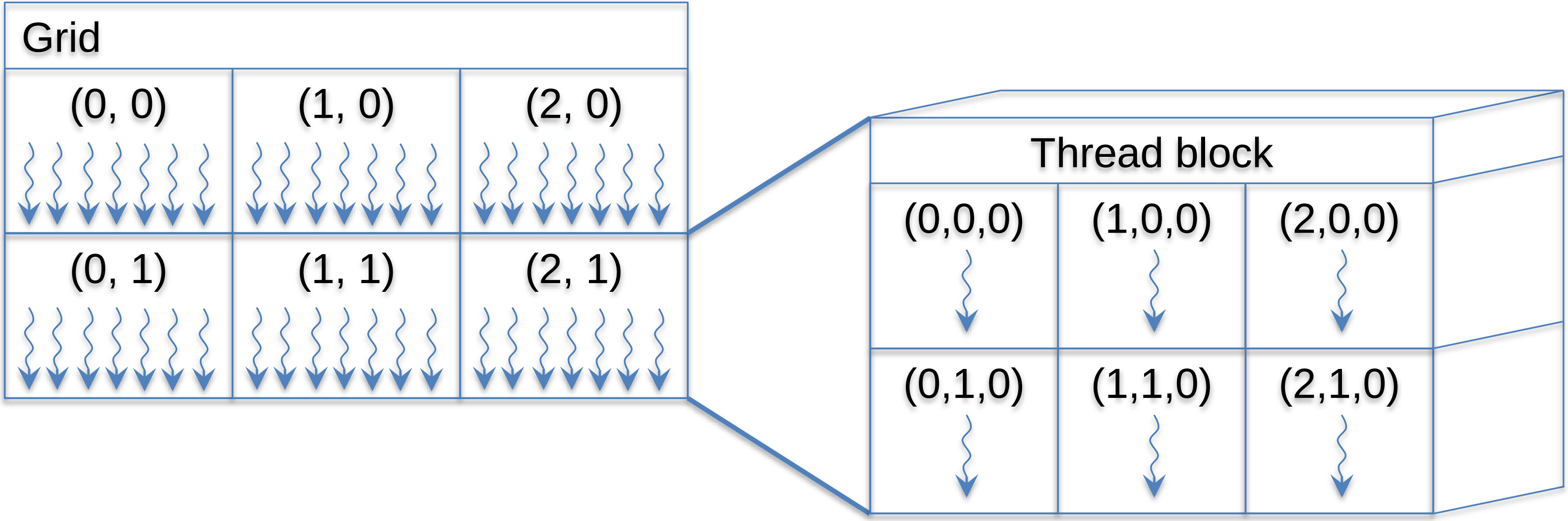
- The CUDA programming model separates a program into a host (CPU) and a device (GPU) part.

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

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

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 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();
  ```
  ```

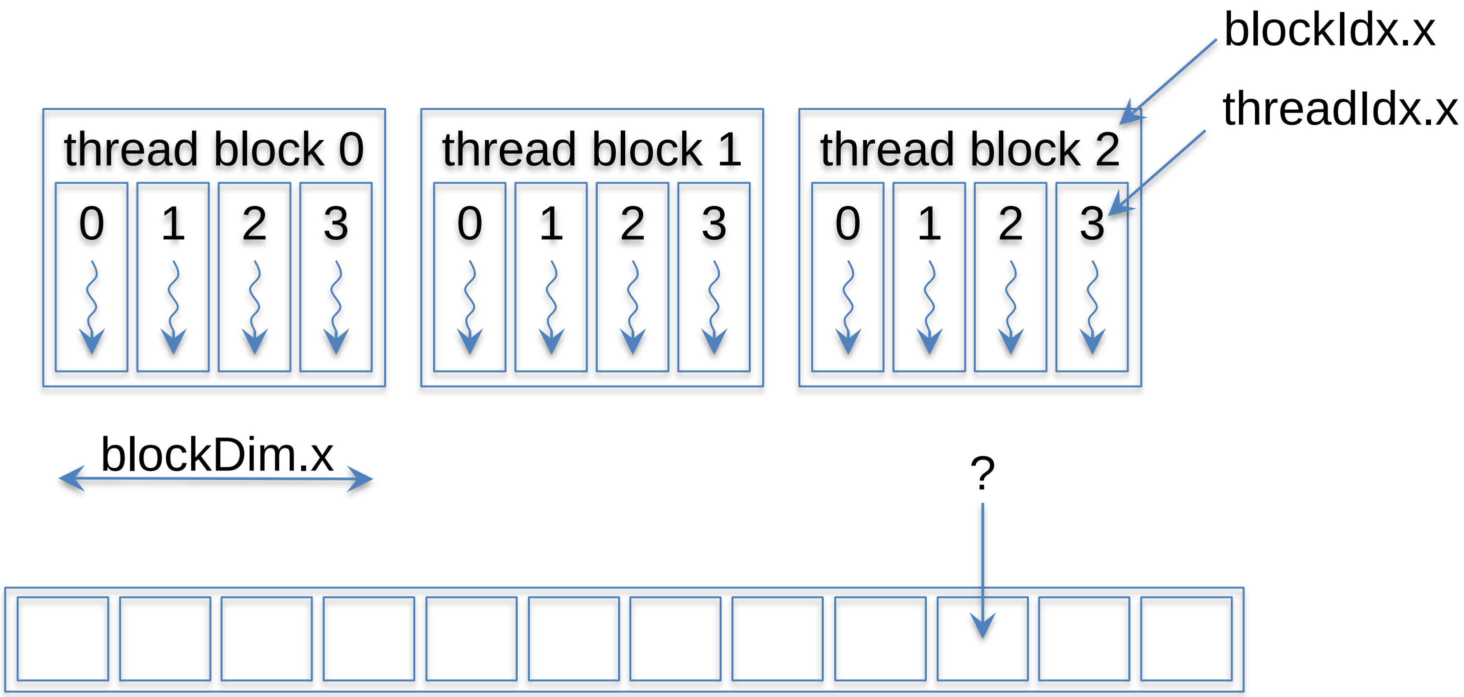
Second hands-on session

- Open the `Run_CUDA_kernels_in_Matlab.ipynb` notebook
- Task #1: Read through the notebook and execute all the cells

- Task #2: Fix the `vector_add` CUDA kernel!

- Hints:
  - Look at how the kernel is launched in the host program
  - `threadIdx.x` is the thread index within the thread block
  - `blockIdx.x` is the block index within the grid
  - `blockDim.x` is the dimension of the thread block

Hint


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

CUDA Program execution

Compilation

CUDA program

PTX assembly

CUBIN bytecode

Machine-level binary

Nvidia Compiler
nvcc

Runtime compiler
driver

How kernels 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 (in different registers)

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

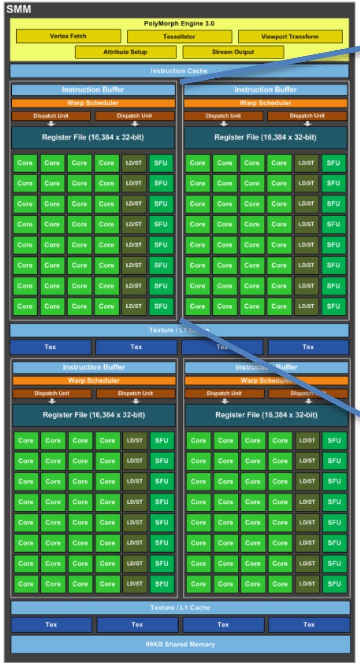
    ```
    if (a > 0.0f)
    ```

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

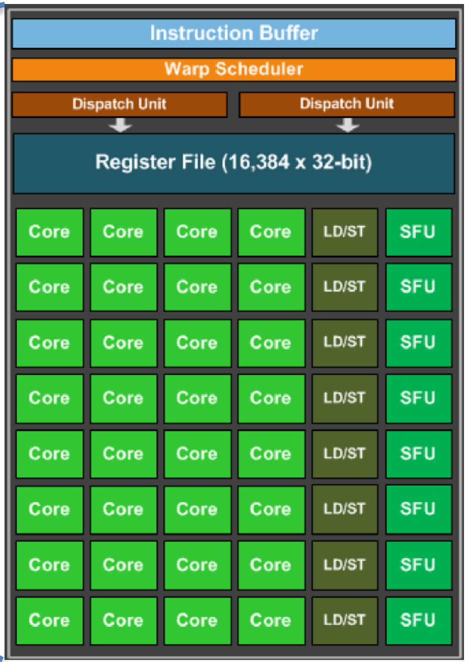
- Thread index-dependent branching is usually harmless, in particular when you respect the warp size

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

Inside the GPU



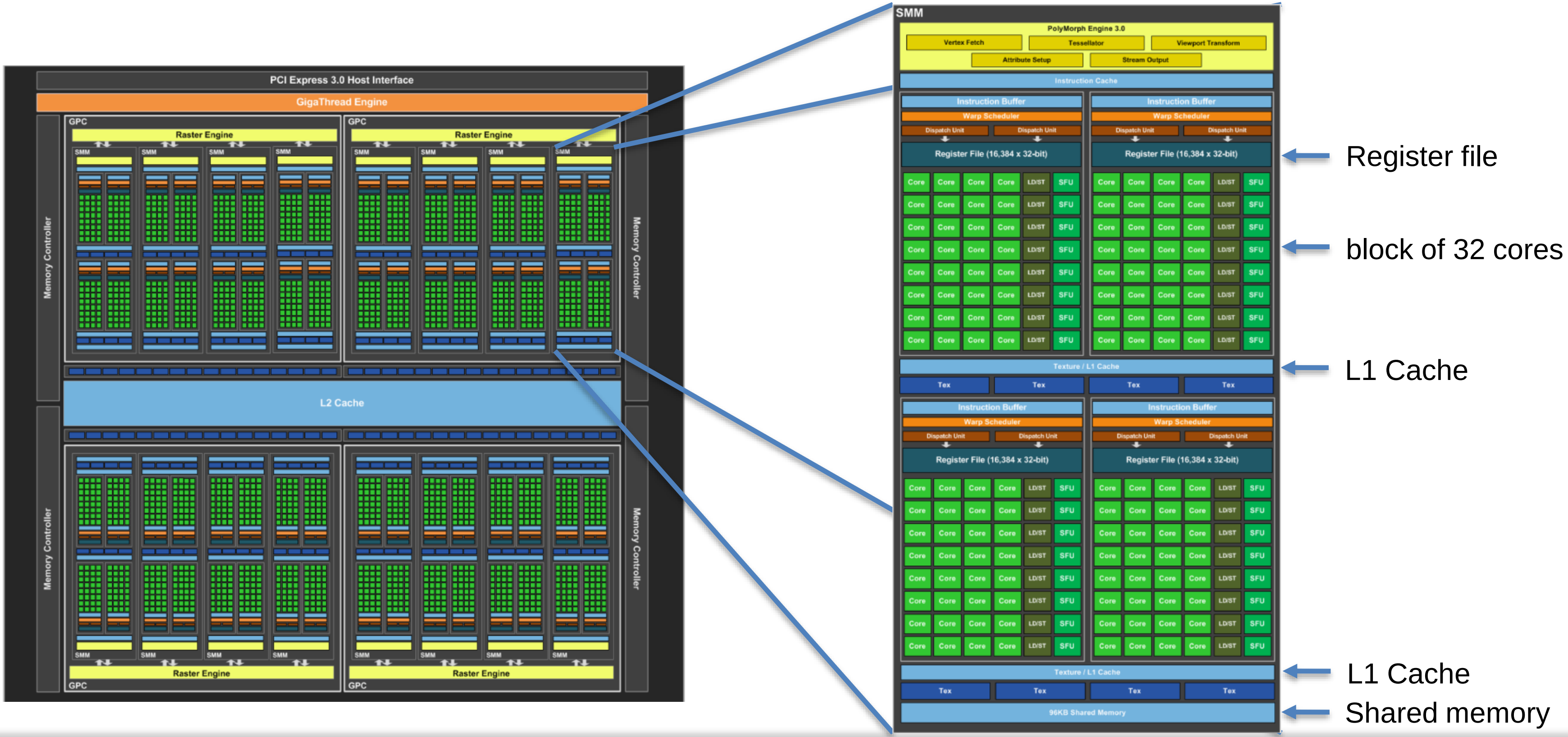
Streaming multiprocessor (SM)



32 core block

(Maxwell architecture)

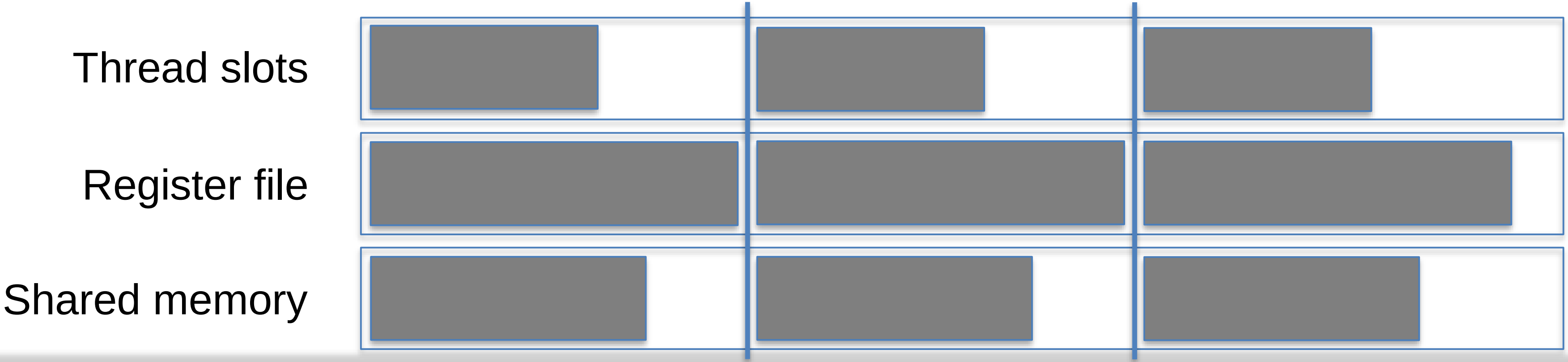
Inside the GPU



(Maxwell architecture)

Resource partitioning

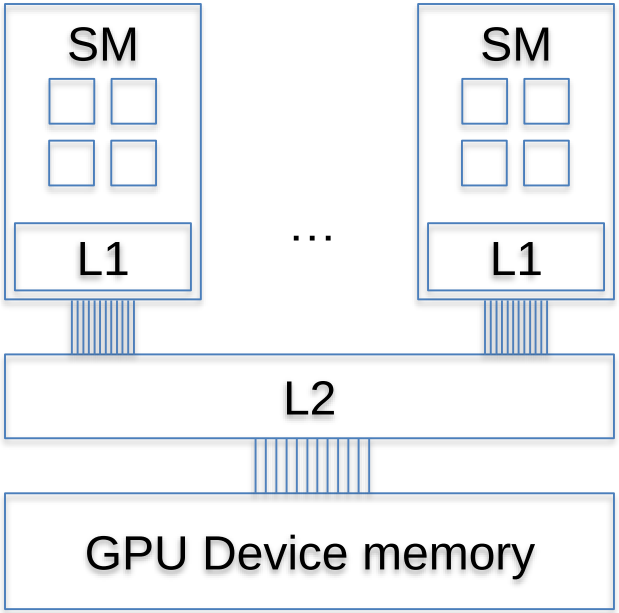
- The GPU consists of several (1 to 56) *streaming multiprocessors* (SMs)
- The SMs are fully independent
- Each SM contains several resources: Thread and Thread Block slots, Register file, and Shared memory
- SM Resources are dynamically partitioned among the thread blocks that execute concurrently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



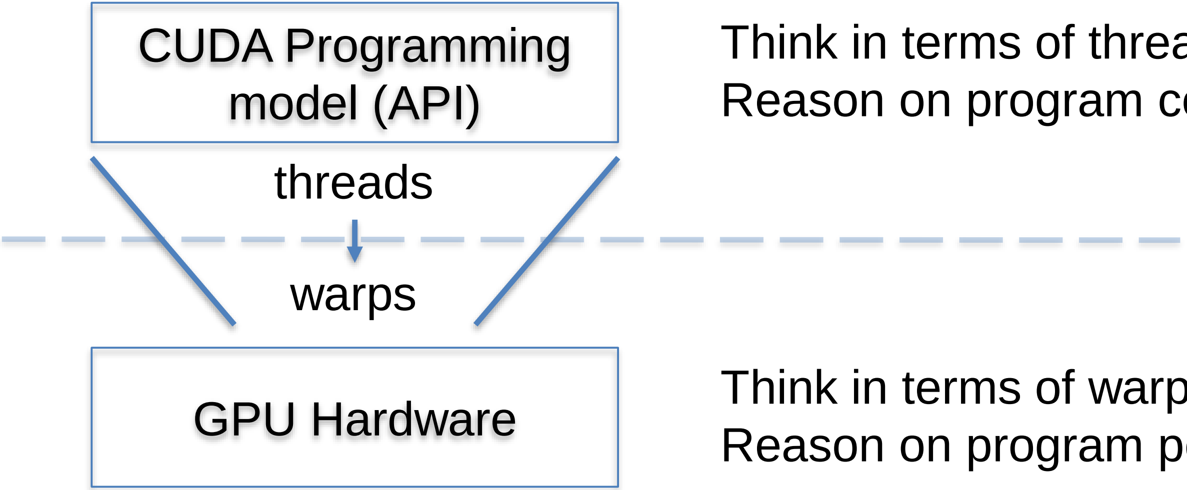
Global Memory access

- Global memory is cached at L2, and for some GPUs also in L1

- When a thread reads a value from global memory, think about:
  - The total number of values that are accessed by the warp that the thread belongs to
  - The cache line length and the number of cache lines that those values will belong to
  - Alignment of the data accesses to that of the cache lines



Overview



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance