



# GPU Introduction

## JSC OpenACC Course 2018

29 October 2018 | Andreas Herten | Forschungszentrum Jülich

# Outline

Introduction

GPU History

Architecture Comparison

Jülich Systems

App Showcase

The GPU Platform

3 Core Features

Memory

Asynchronicity

SIMT

High Throughput

Summary

Programming GPUs

Libraries

GPU programming models

CUDA

# History of GPUs

A short but parallel story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]

# History of GPUs

A short but parallel story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

# History of GPUs

A short but parallel story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA

# History of GPUs

A short but parallel story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA
- 2009 OpenCL

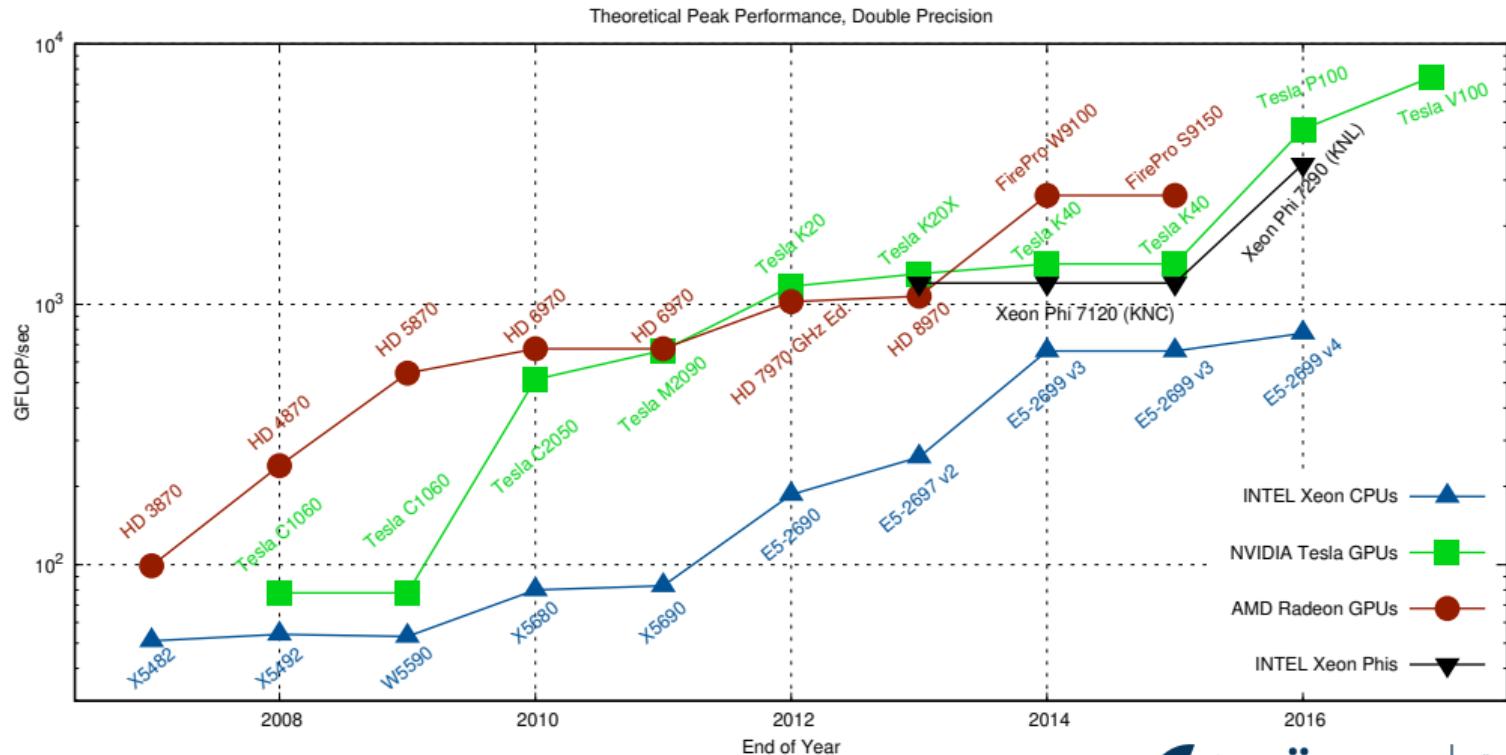
# History of GPUs

A short but parallel story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA
- 2009 OpenCL
- 2018 Top 500: 20 % with GPUs (#1, #3) [4], Green 500: 7 of top 10 with GPUs [5]

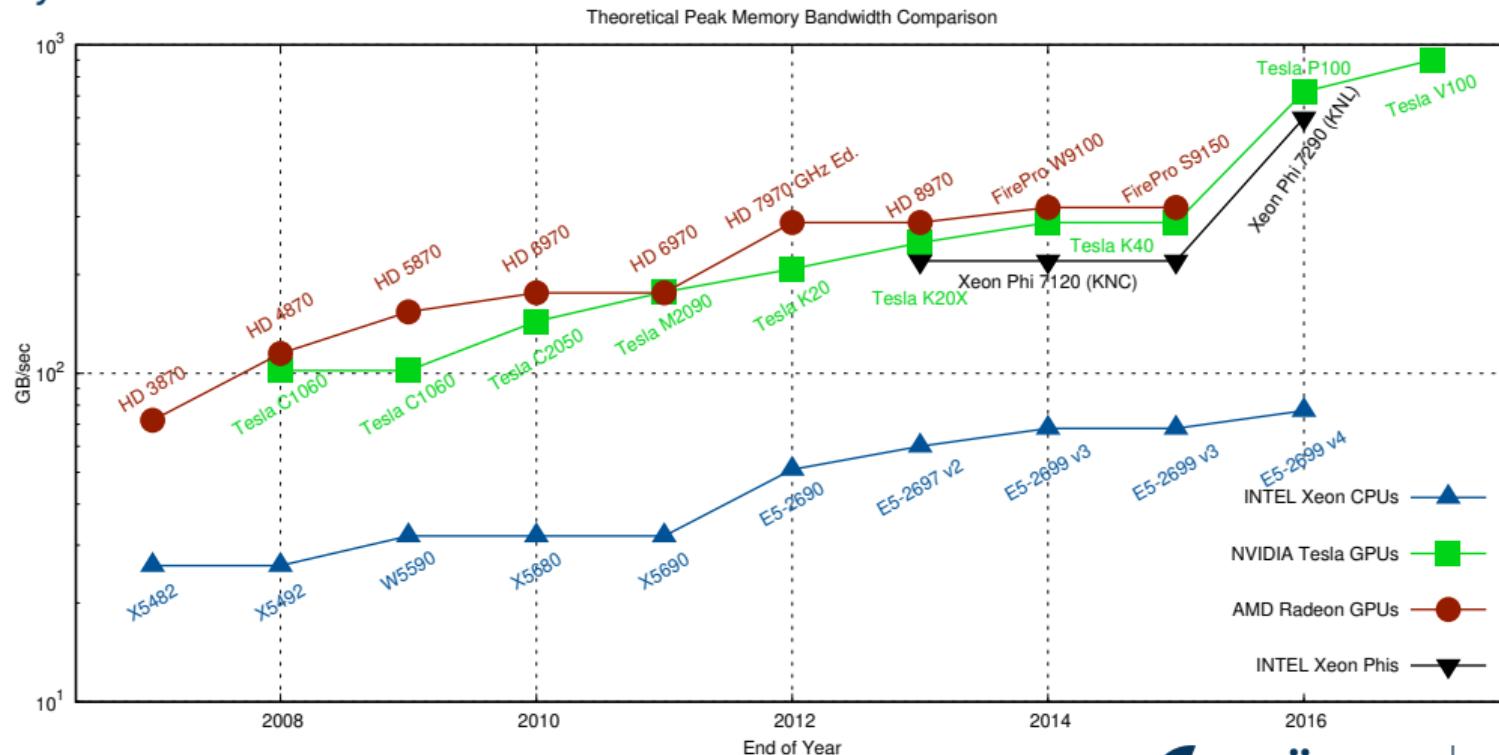
# Status Quo Across Architectures

## Performance



# Status Quo Across Architectures

## Memory Bandwidth





## JURECA – Jülich's Multi-Purpose Supercomputer

- 1872 nodes with Intel Xeon E5 CPUs ( $2 \times 12$  cores)
- 75 nodes with 2 NVIDIA Tesla K80 cards (look like 4 GPUs)
- JURECA Booster: 1640 nodes with Intel Xeon Phi *Knights Landing*
- 1.8 (CPU) + 0.44 (GPU) + 5 (KNL) PFLOP/s peak performance (#29)
- Mellanox EDR InfiniBand



## JUWELS – Jülich's New Large System

- 2500 nodes with Intel Xeon CPUs ( $2 \times 24$  cores)
- 48 nodes with 4 NVIDIA Tesla V100 cards
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance

# Getting GPU-Acquainted

## Some Applications

TASK

Location of Code:

1-Introduction-.../Tasks/getting-started/

See Instructions.md for hints.

# Getting GPU-Acquainted

## Some Applications

TASK

GEMM

N-Body

Location of Code:  
1-Introduction-.../Tasks/getting-started/

See Instructions.md for hints.

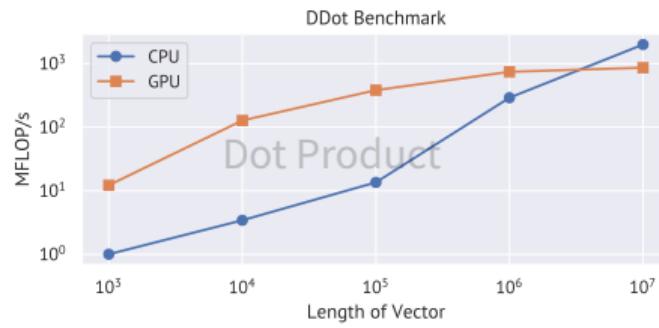
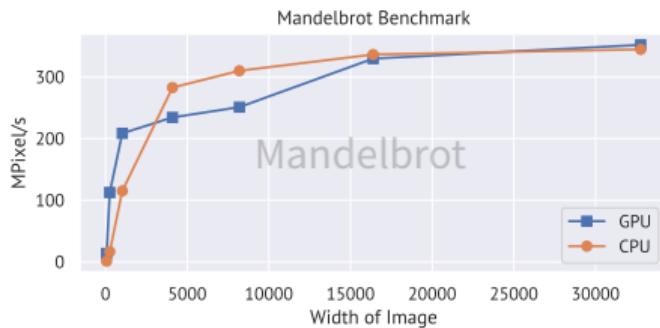
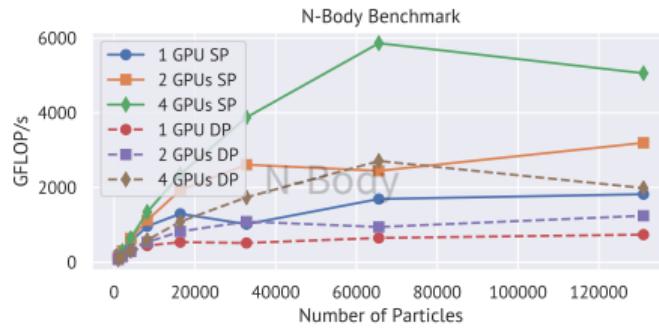
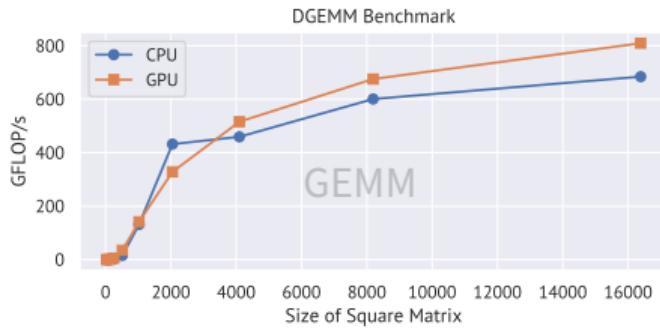
Mandelbrot

Dot Product

# Getting GPU-Acquainted

TASK

## Some Applications



# The GPU Platform

# CPU vs. GPU

A matter of specialties



# CPU vs. GPU

A matter of specialties



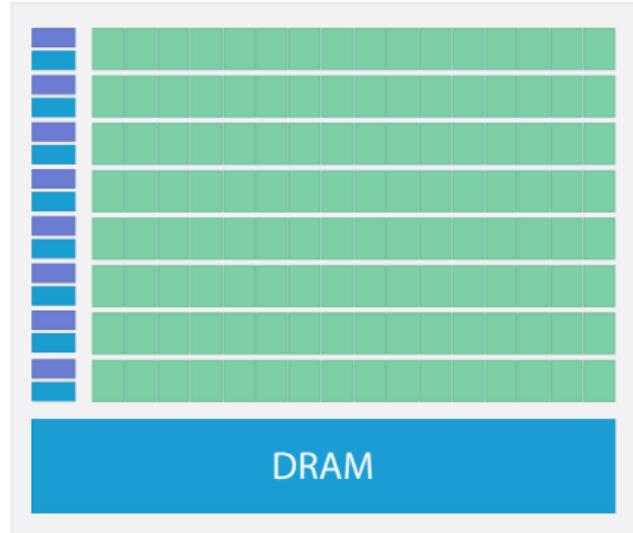
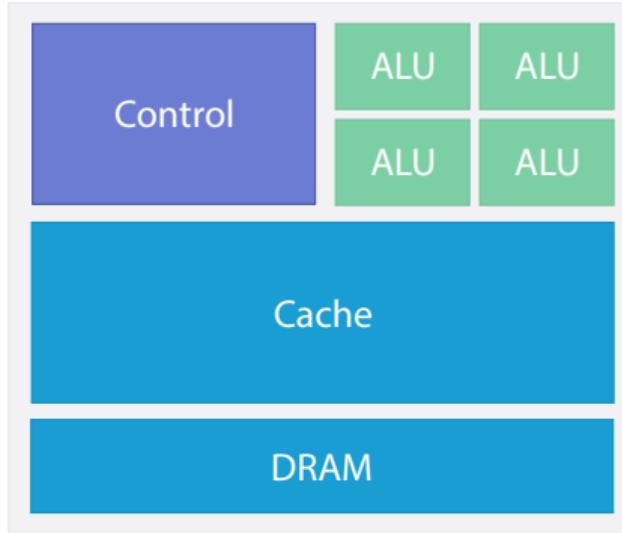
Transporting one



Transporting many

# CPU vs. GPU

## Chip



# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

SIMT

Asynchronicity

Memory

# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

SIMT

Asynchronicity

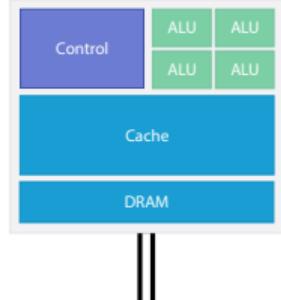
Memory

# Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card  
→ Separate device from CPU

Host



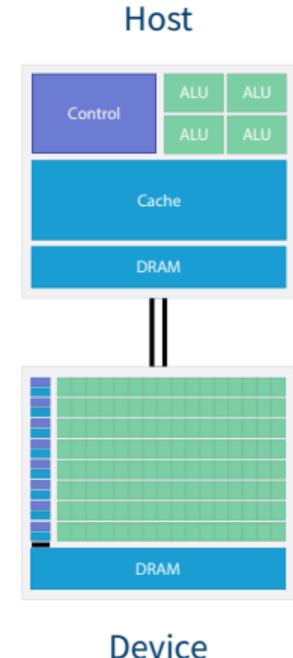
Device

# Memory

GPU memory ain't no CPU memory

*Unified Virtual Addressing*

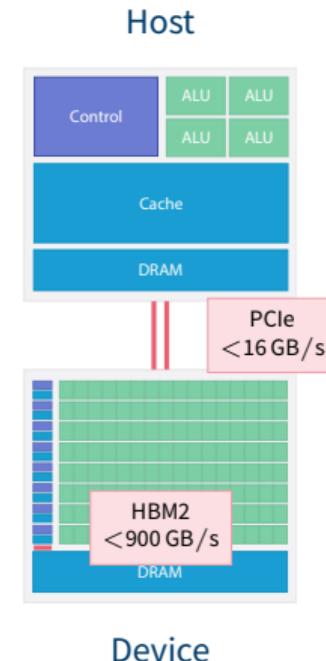
- GPU: accelerator / extension card
  - Separate device from CPU
- Separate memory, but UVA**



# Memory

GPU memory ain't no CPU memory

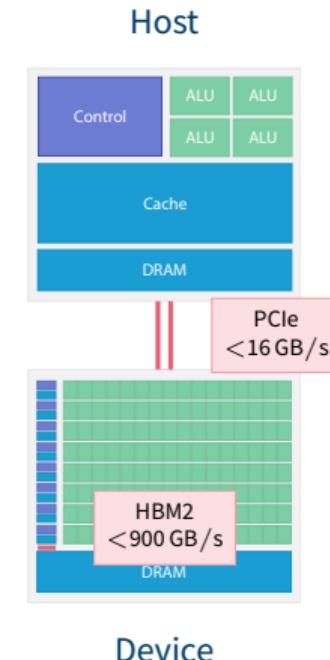
- GPU: accelerator / extension card
- Separate device from CPU
- Separate memory, but UVA**



# Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card
  - Separate device from CPU
  - Separate memory, but UVA**
  - Memory transfers need special consideration!
- Do as little as possible!*



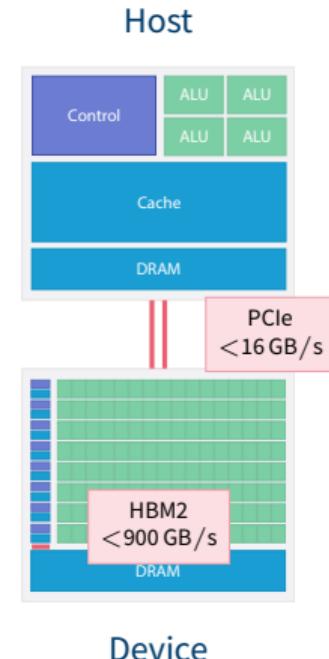
# Memory

GPU memory ain't no CPU memory

Unified Memory



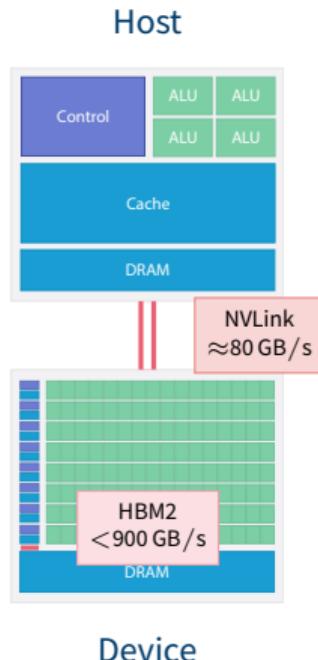
- GPU: accelerator / extension card  
→ Separate device from CPU
- Separate memory, but UVA and UM**
- Memory transfers need special consideration!  
*Do as little as possible!*
- Formerly: Explicitly copy data to/from GPU  
Now: Done automatically (performance...?)



# Memory

GPU memory ain't no CPU memory

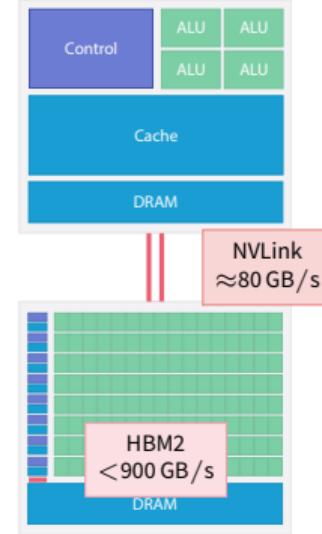
- GPU: accelerator / extension card  
→ Separate device from CPU  
**Separate memory, but UVA and UM**
- Memory transfers need special consideration!  
*Do as little as possible!*
- Formerly: Explicitly copy data to/from GPU  
Now: Done automatically (performance...?)



# Memory

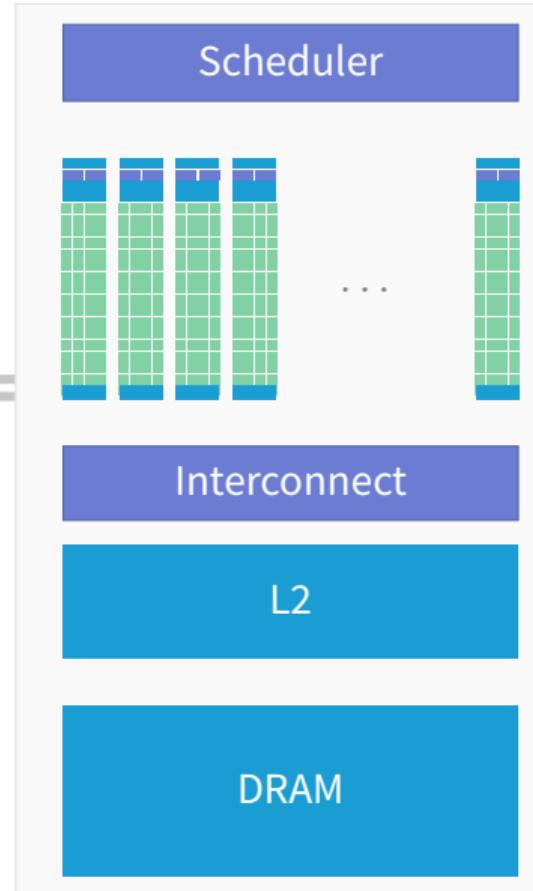
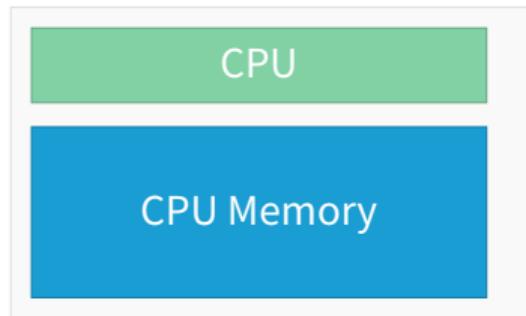
GPU memory ain't no CPU memory

- GPU: accelerator / extension card  
→ Separate device from CPU  
**Separate memory, but UVA and UM**
- Memory transfers need special consideration!  
*Do as little as possible!*
- Formerly: Explicitly copy data to/from GPU  
Now: Done automatically (performance...?)
- P100: 16 GB RAM, 720 GB/s; V100: 16 (32) GB RAM, 900 GB/s



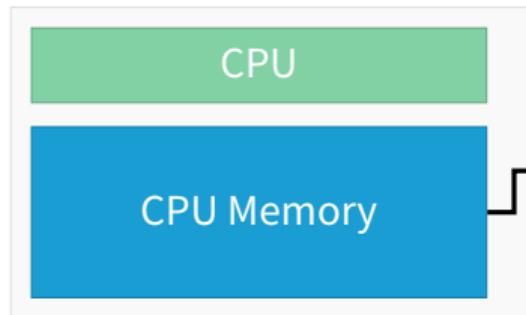
# Processing Flow

CPU → GPU → CPU

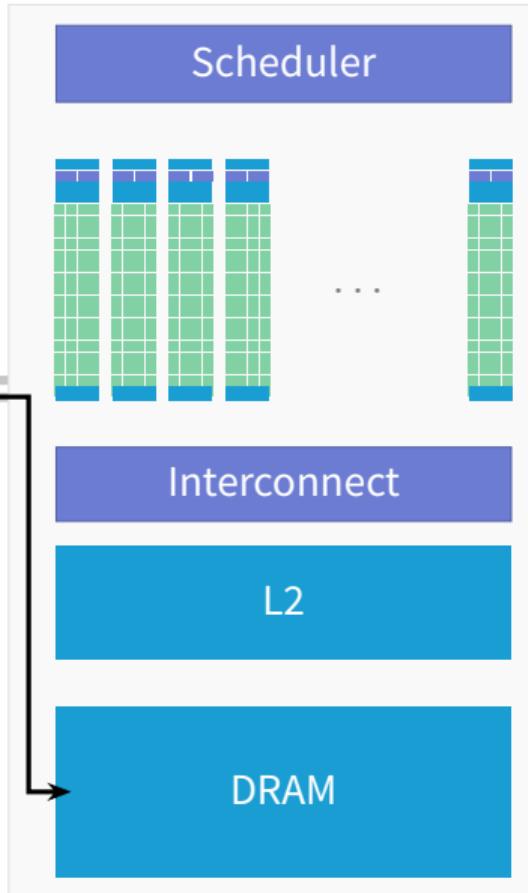


# Processing Flow

CPU → GPU → CPU

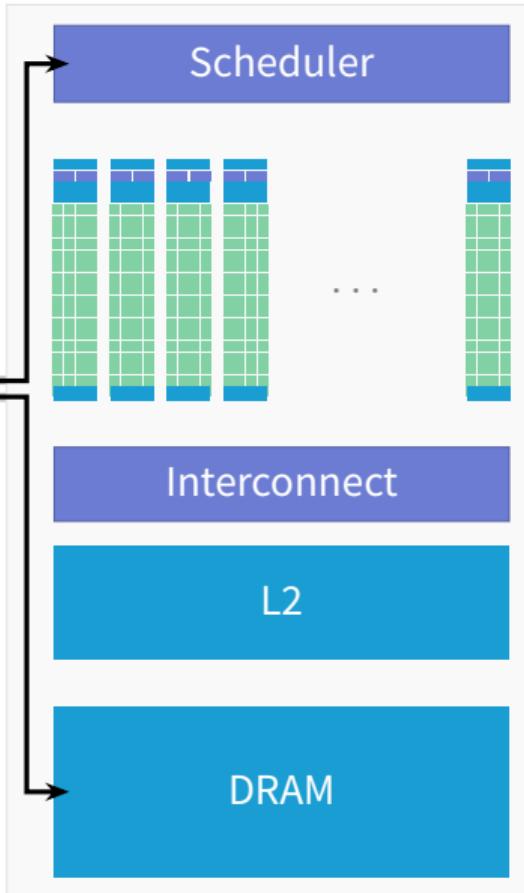
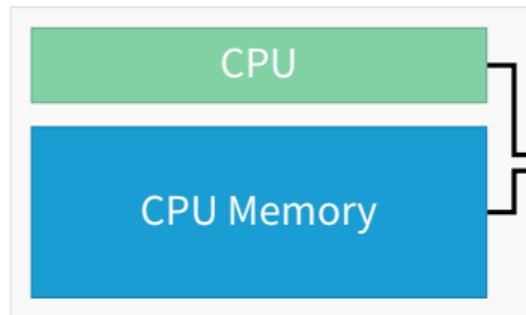


- 1 Transfer data from CPU memory to GPU memory



# Processing Flow

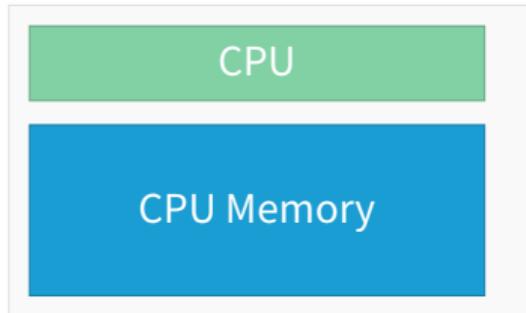
CPU → GPU → CPU



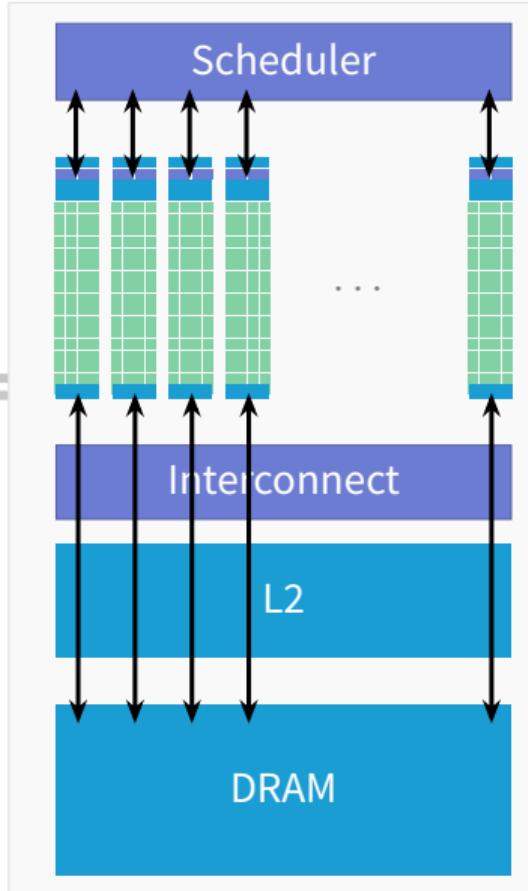
- 1 Transfer data from CPU memory to GPU memory, transfer program

# Processing Flow

CPU → GPU → CPU

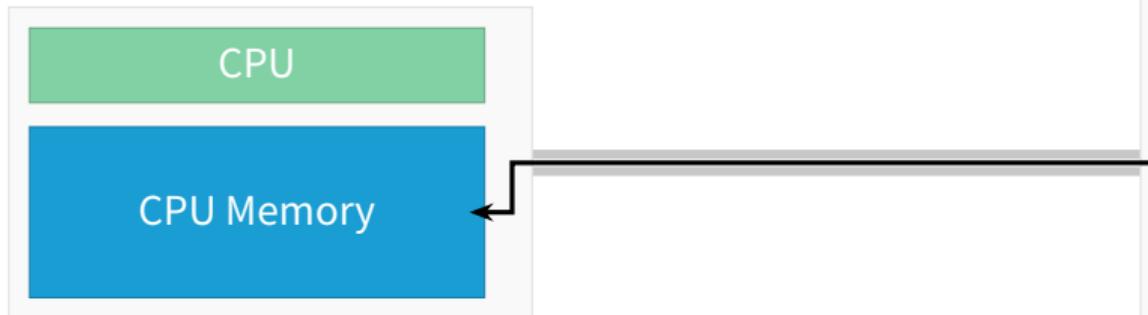


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back

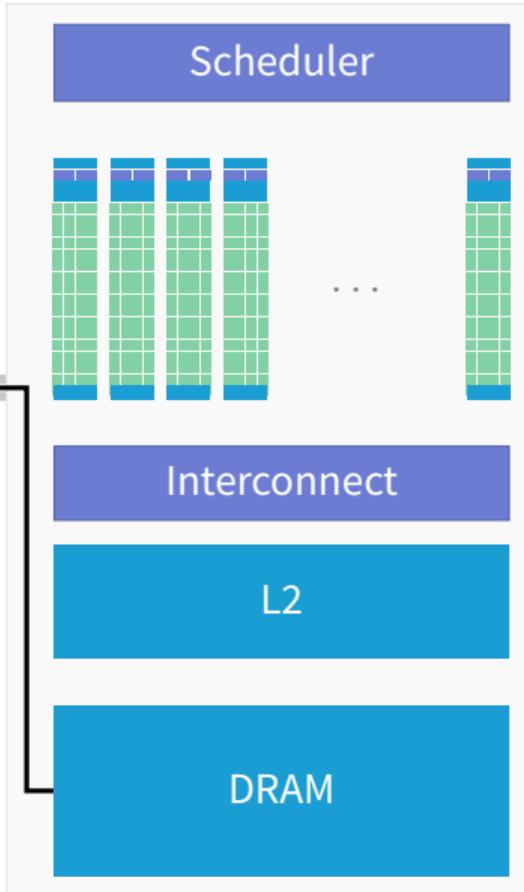


# Processing Flow

CPU → GPU → CPU



- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back
- 3 Transfer results back to host memory



# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

SIMT

Asynchronicity

Memory

# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

SIMT

Asynchronicity

Memory

# Async

## Following different streams

- Problem: Memory transfer is comparably slow  
Solution: Do something else in meantime (**computation**)!
- Overlap tasks
- Copy and compute engines run separately (*streams*)



- GPU needs to be fed: Schedule many computations
- CPU can do other work while GPU computes; synchronization

# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

SIMT

Asynchronicity

Memory

# GPU Architecture

## Overview

Aim: Hide Latency  
*Everything else follows*

**SIMT**

**Asynchronicity**

**Memory**

# SIMT

**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (**SIMD**)

*Scalar*

$$\begin{array}{rcl} A_0 & + & B_0 \\ \hline A_1 & + & B_1 \\ \hline A_2 & + & B_2 \\ \hline A_3 & + & B_3 \\ \hline \end{array} = \begin{array}{l} C_0 \\ C_1 \\ C_2 \\ C_3 \end{array}$$

# SIMT

**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (**SIMD**)

*Vector*

$$\begin{array}{c} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{c} \\ \\ \\ \end{array} = \begin{array}{c} \\ \\ \\ \end{array}$$

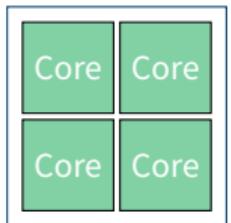
# SIMT

**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (**SIMD**)
  - Simultaneous Multithreading (**SMT**)

*Vector*

$$\begin{array}{c} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{c} \\ \\ \\ \end{array} = \begin{array}{c} \\ \\ \\ \end{array}$$



# SIMT

**SIMT = SIMD  $\oplus$  SMT**

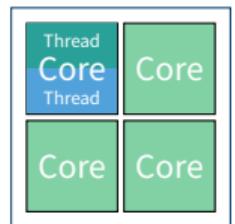
- CPU:

- Single Instruction, Multiple Data (**SIMD**)
- Simultaneous Multithreading (**SMT**)

*Vector*

$$\begin{array}{ccc} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{c} \\ \\ \\ \end{array} = \begin{array}{ccc} & & \\ & & \\ & & \\ & & \end{array}$$

*SMT*



# SIMT

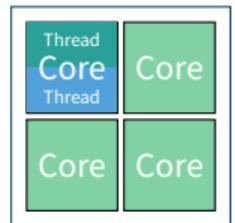
**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (**SIMD**)
  - Simultaneous Multithreading (**SMT**)
- GPU: Single Instruction, Multiple Threads (**SIMT**)

*Vector*

$$\begin{array}{ccc} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{c} \\ \\ \\ \end{array} = \begin{array}{ccc} & & \\ & & \\ & & \\ & & \end{array}$$

*SMT*



# SIMT

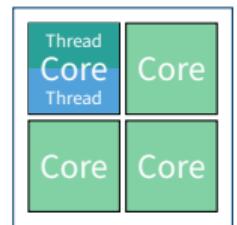
**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (SIMD)
  - Simultaneous Multithreading (SMT)
- GPU: Single Instruction, Multiple Threads (SIMT)

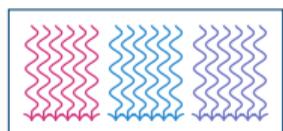
*Vector*

$$\begin{array}{ccc} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{ccc} & & \\ & & \end{array} = \begin{array}{ccc} & & \\ & & \end{array}$$

*SMT*



*SIMT*



# SIMT

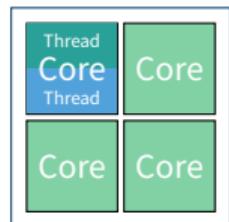
**SIMT = SIMD  $\oplus$  SMT**

- CPU:
  - Single Instruction, Multiple Data (**SIMD**)
  - Simultaneous Multithreading (**SMT**)
- GPU: Single Instruction, Multiple Threads (**SIMT**)
  - CPU core  $\approx$  GPU multiprocessor (**SM**)
  - Working unit: set of threads (32, a *warp*)
  - Fast switching of threads (large register file)
  - Branching    if 

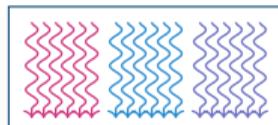
*Vector*

$$\begin{array}{ccc} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{array} + \begin{array}{ccc} & & \\ & & \end{array} = \begin{array}{ccc} & & \\ & & \end{array}$$

*SMT*



*SIMT*

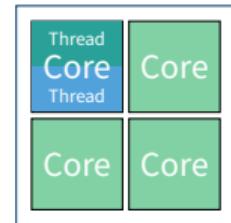




Vector

$$\begin{array}{c} A_0 \\ A_1 \\ A_2 \\ A_3 \end{array} + \begin{array}{c} B_0 \\ B_1 \\ B_2 \\ B_3 \end{array} = \begin{array}{c} C_0 \\ C_1 \\ C_2 \\ C_3 \end{array}$$

SMT



Graphics: Nvidia Corporation [9]

SIMT

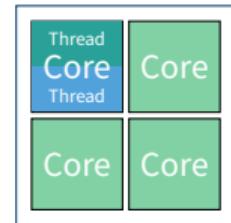




Vector

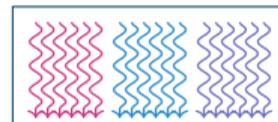
$$\begin{array}{c} A_0 \\ A_1 \\ A_2 \\ A_3 \end{array} + \begin{array}{c} B_0 \\ B_1 \\ B_2 \\ B_3 \end{array} = \begin{array}{c} C_0 \\ C_1 \\ C_2 \\ C_3 \end{array}$$

SMT



Graphics: Nvidia Corporation [9]

SIMT



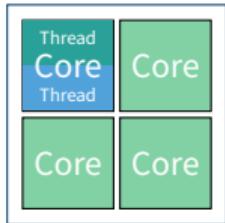
## Multiprocessor



Vector

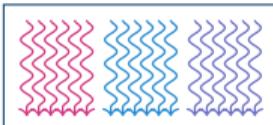
$$\begin{matrix} A_0 & & B_0 & & C_0 \\ A_1 & + & B_1 & = & C_1 \\ A_2 & & B_2 & & C_2 \\ A_3 & & B_3 & & C_3 \end{matrix}$$

SMT



Graphics: Nvidia Corporation [9]

SIMT



# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

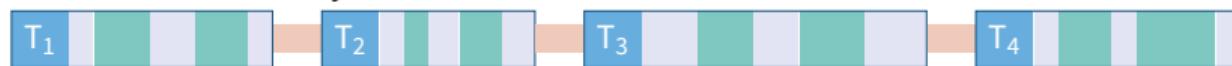
# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency



- Thread/Warp
- Processing
- Context Switch
- Ready
- Waiting

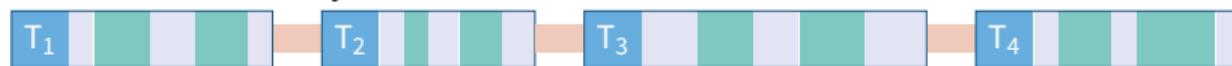
# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

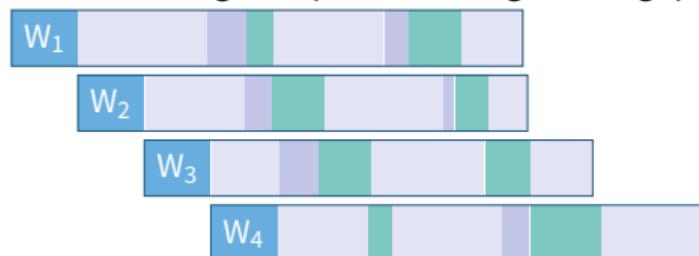
CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency



GPU Streaming Multiprocessor: High Throughput



- Thread/Warp
- Processing
- Context Switch
- Ready
- Waiting

# CPU vs. GPU

Let's summarize this!



## Optimized for low latency

- + Large main memory
- + Fast clock rate
- + Large caches
- + Branch prediction
- + Powerful ALU
- Relatively low memory bandwidth
- Cache misses costly
- Low performance per watt



## Optimized for high throughput

- + High bandwidth main memory
- + Latency tolerant (parallelism)
- + More compute resources
- + High performance per watt
- Limited memory capacity
- Low per-thread performance
- Extension card

# Programming GPUs

# Summary of Acceleration Possibilities

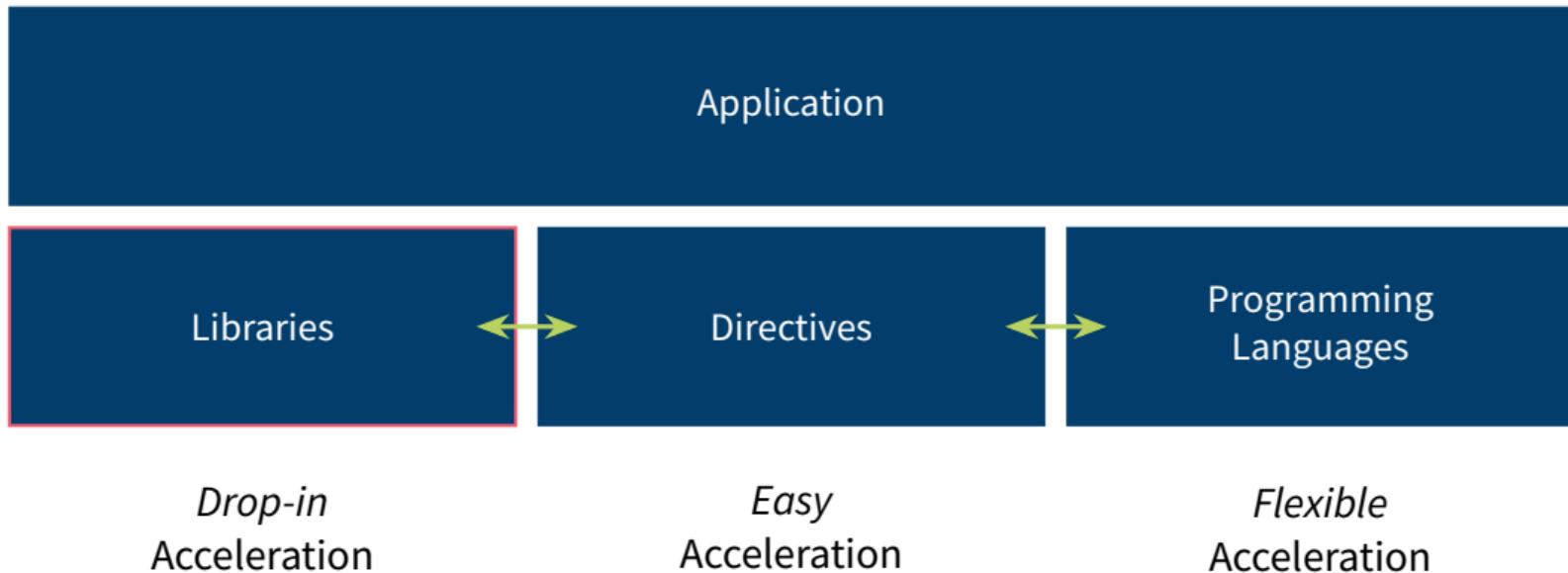


*Drop-in*  
Acceleration

*Easy*  
Acceleration

*Flexible*  
Acceleration

# Summary of Acceleration Possibilities



# Libraries

Programming GPUs is easy: **Just don't!**

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*

# Libraries

Programming GPUs is easy: Just don't!

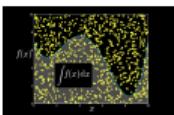
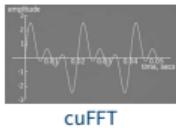
*Use applications & libraries*



# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



Numba



theano

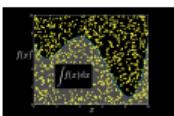
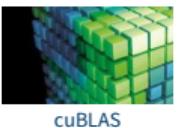


JÜLICH  
SUPERCOMPUTING  
CENTRE

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*

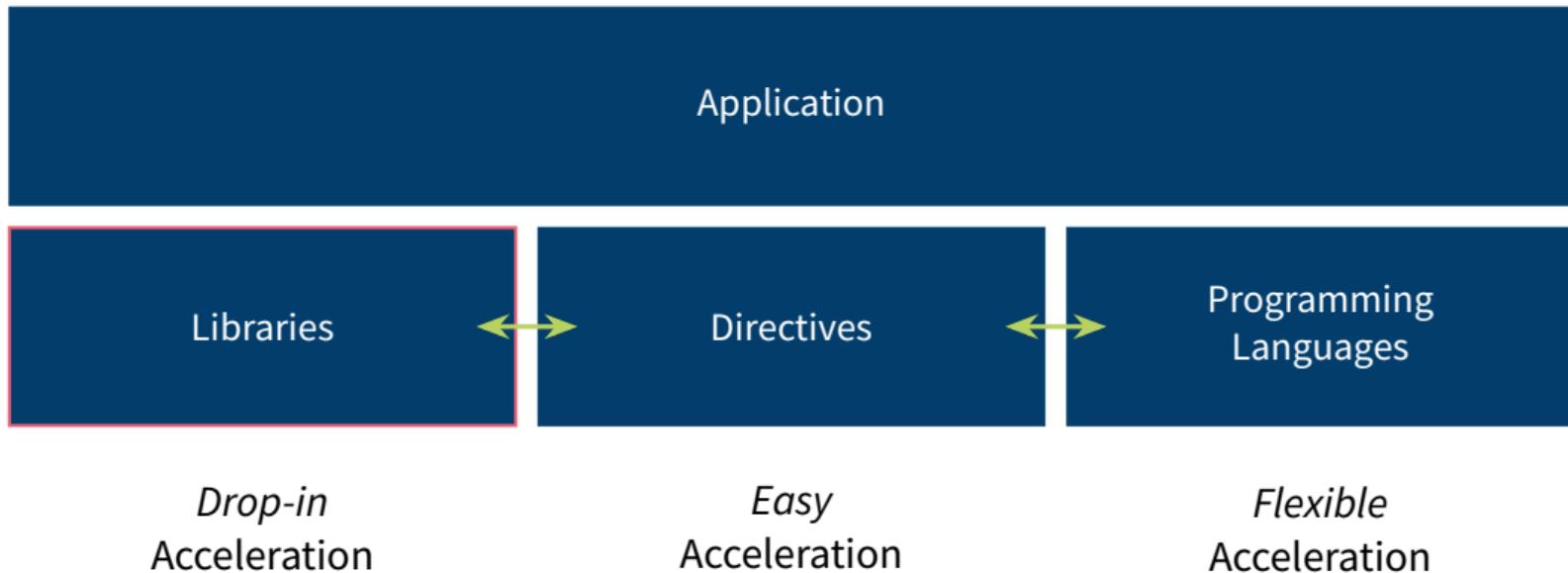


Numba

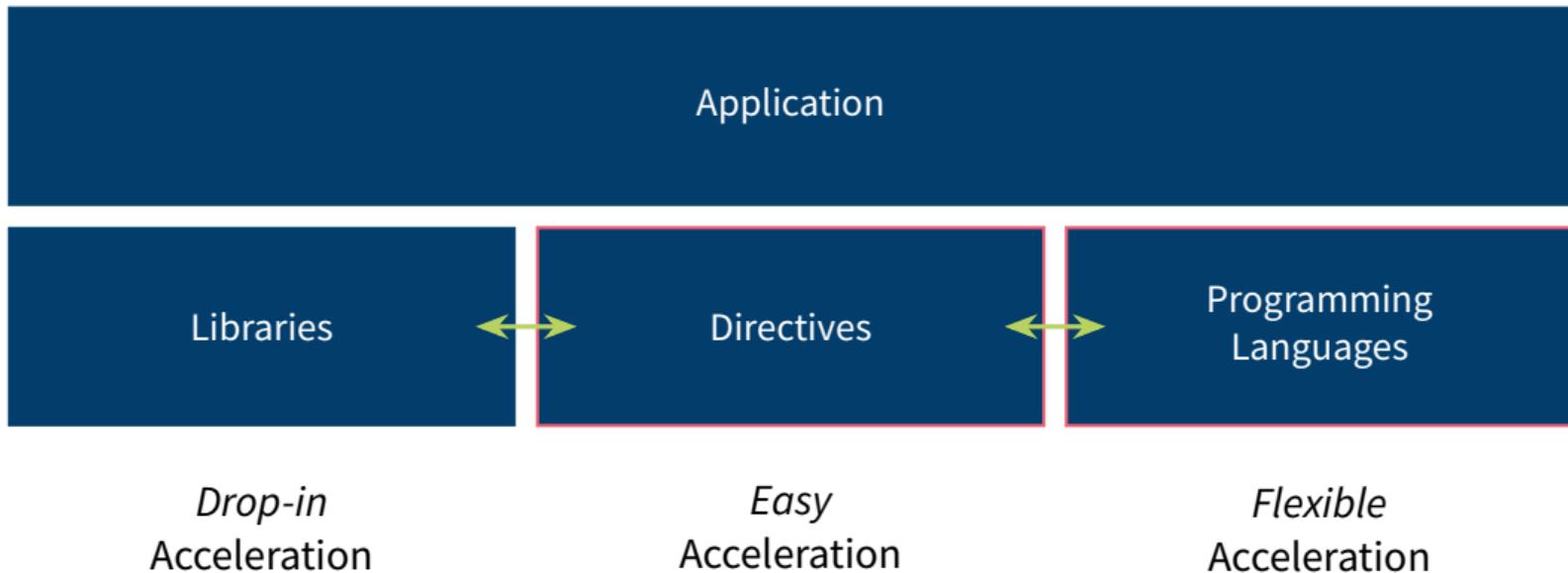
theano



# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



# ⚠ Parallelism

Libraries are not enough?

You need to write your own GPU code?

# Primer on Parallel Scaling

## Amdahl's Law

Possible maximum speedup for  
 $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

# Primer on Parallel Scaling

## Amdahl's Law

Possible maximum speedup for  
 $N$  parallel processors

Total Time  $t = t_{\text{serial}} + t_{\text{parallel}}$

$N$  Processors  $t(N) = t_s + t_p/N$

# Primer on Parallel Scaling

## Amdahl's Law

Possible maximum speedup for  
 $N$  parallel processors

**Total Time**  $t = t_{\text{serial}} + t_{\text{parallel}}$

**$N$  Processors**  $t(N) = t_s + t_p/N$

**Speedup**  $s(N) = t/t(N) = \frac{t_s+t_p}{t_s+t_p/N}$

# Primer on Parallel Scaling

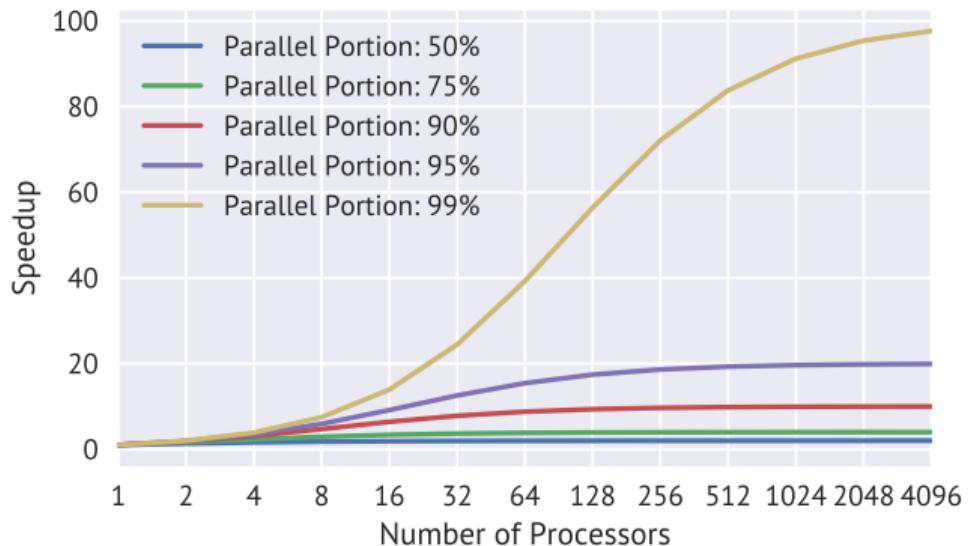
## Amdahl's Law

Possible maximum speedup for  $N$  parallel processors

Total Time  $t = t_{\text{serial}} + t_{\text{parallel}}$

$N$  Processors  $t(N) = t_s + t_p/N$

Speedup  $s(N) = t/t(N) = \frac{t_s+t_p}{t_s+t_p/N}$



# ⚠ Parallelism

Parallel programming is not easy!

Things to consider:

- Is my application **computationally intensive** enough?
- What are the levels of **parallelism**?
- How much **data** needs to be **transferred**?
- Is the **gain** worth the **pain**?

# Possibilities

Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*...

- OpenACC
- OpenMP
- Thrust
- PyCUDA
- CUDA Fortran
- CUDA
- OpenCL

# Possibilities

Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*...

- OpenACC
- OpenMP
- Thrust
- PyCUDA
- CUDA Fortran
- CUDA
- OpenCL

# Possibilities

Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*...

- OpenACC
- OpenMP
- Thrust
- PyCUDA
- CUDA Fortran
- CUDA
- OpenCL

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cudaMallocManaged(&x, n * sizeof(float));
cudaMallocManaged(&y, n * sizeof(float));

saxpy_cuda<<<2, 5>>>(n, a, x, y);

cudaDeviceSynchronize();
```

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}
```

Specify kernel

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}
```

Specify kernel

ID variables

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}
```

Specify kernel

ID variables

Guard against  
too many threads

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}
```

Specify kernel

ID variables

Guard against  
too many threads

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

Allocate GPU-capable  
memory

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

Specify kernel

ID variables

Guard against too many threads

Allocate GPU-capable memory

Call kernel  
2 blocks, each 5 threads

# CUDA SAXPY

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$  (single precision)

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

Specify kernel

ID variables

Guard against too many threads

Allocate GPU-capable memory

Call kernel 2 blocks, each 5 threads

Wait for kernel to finish

# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Thread



# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:
  - Threads



# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → Block



# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → **Block**
- **Block**

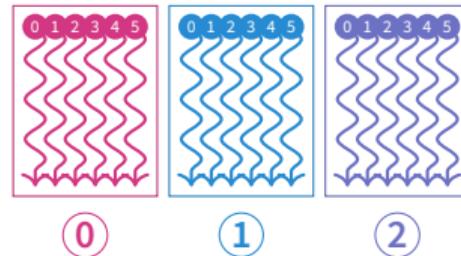


# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → Block
- Blocks

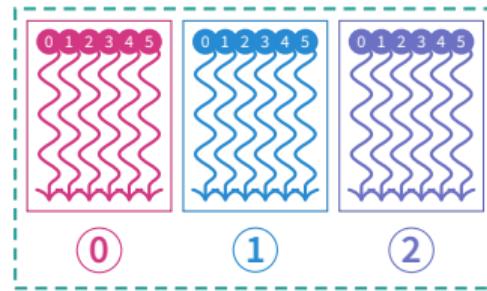


# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid

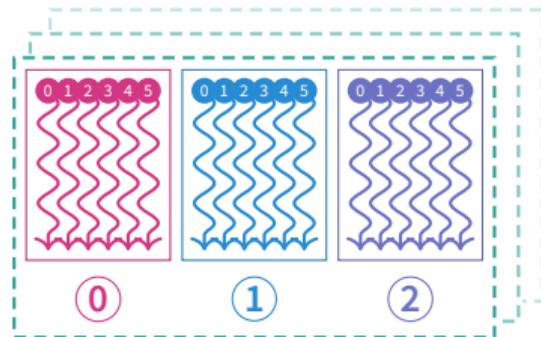


# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- Threads & blocks in 3D

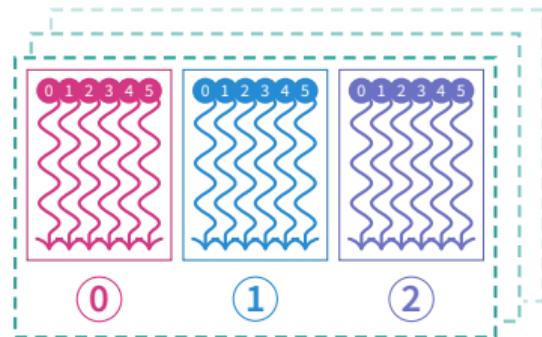


# CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- Threads & blocks in 3D

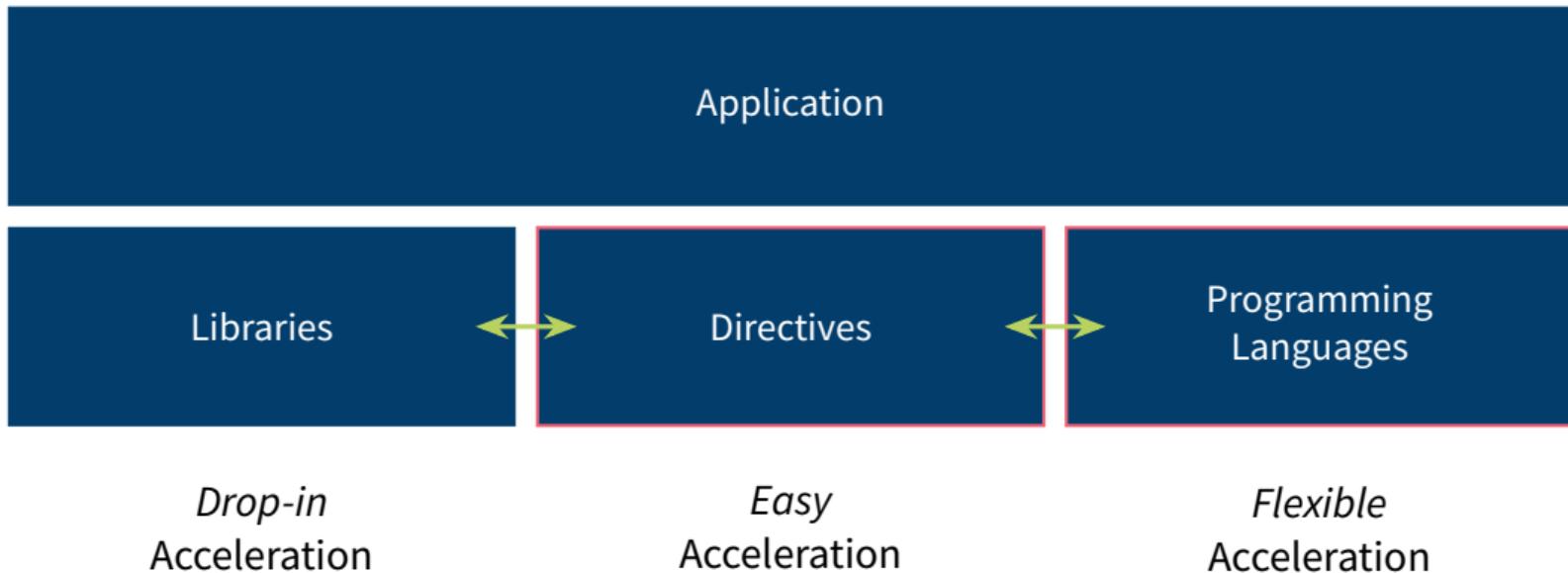


- Execution entity: **threads**

- Lightweight → fast switching!
- 1000s threads execute simultaneously → order non-deterministic!

- **OpenACC** takes care of threads and blocks for you!  
→ Block configuration is just an optimization!

# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



*Drop-in*  
Acceleration

*Easy*  
Acceleration

*Flexible*  
Acceleration

# Conclusions

- GPUs achieve performance by specialized hardware → **threads**
  - Faster *time-to-solution*
  - Lower *energy-to-solution*
- GPU acceleration can be done by different means
- Libraries are the easiest, CUDA the fullest
- OpenACC good compromise

# Conclusions

- GPUs achieve performance by specialized hardware → **threads**
  - Faster *time-to-solution*
  - Lower *energy-to-solution*
- GPU acceleration can be done by different means
- Libraries are the easiest, CUDA the fullest
- **OpenACC** good compromise

# Conclusions

- GPUs achieve performance by specialized hardware → **threads**
  - Faster *time-to-solution*
  - Lower *energy-to-solution*
- GPU acceleration can be done by different means
- Libraries are the easiest, CUDA the fullest
- **OpenACC** good compromise

Thank you  
for your attention!  
[a.herten@fz-juelich.de](mailto:a.herten@fz-juelich.de)

# Appendix

## Glossary

## References

# Glossary I

- API** A programmatic interface to software by well-defined functions. Short for application programming interface. [95](#)
- ATI** Canada-based [GPUs](#) manufacturing company; bought by AMD in 2006. [3](#), [4](#), [5](#), [6](#), [7](#)
- CUDA** Computing platform for [GPUs](#) from NVIDIA. Provides, among others, CUDA C/C++. [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [69](#), [70](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [77](#), [78](#), [79](#), [80](#), [81](#), [82](#), [83](#), [84](#), [85](#), [86](#), [87](#), [90](#), [91](#), [92](#), [95](#)
- JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. [94](#)
- JURECA** A multi-purpose supercomputer with 1800 nodes at JSC. [10](#)

## Glossary II

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. [11](#)

NVIDIA US technology company creating GPUs. [3](#), [4](#), [5](#), [6](#), [7](#), [10](#), [11](#), [94](#), [96](#)

OpenACC Directive-based programming, primarily for many-core machines. [1](#), [69](#), [70](#), [71](#),  
[79](#), [80](#), [81](#), [82](#), [83](#), [84](#), [85](#), [86](#), [87](#)

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures ([CPU](#), [GPU](#), DSP, FPGA). The alternative to [CUDA](#). [3](#), [4](#), [5](#), [6](#), [7](#), [69](#), [70](#),  
[71](#)

OpenGL The *Open Graphics Library*, an [API](#) for rendering graphics across different hardware architectures. [3](#), [4](#), [5](#), [6](#), [7](#)

OpenMP Directive-based programming, primarily for multi-threaded machines. [69](#), [70](#), [71](#)

# Glossary III

**SAXPY** Single-precision  $A \times X + Y$ . A simple code example of scaling a vector and adding an offset. 72, 73, 74, 75, 76, 77, 78

**Tesla** The GPU product line for general purpose computing computing of NVIDIA. 10, 11

**Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 69, 70, 71

# References: Images, Graphics I

- [1] Igor Ovsyannykov. *Yarn*. Freely available at Unsplash. URL:  
<https://unsplash.com/photos/hvILKk7SlH4>.
- [6] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL:  
<https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/> (pages 8, 9).
- [7] Mark Lee. *Picture: kawasaki ninja*. URL:  
<https://www.flickr.com/photos/pochacco20/39030210/> (pages 16, 17).
- [8] Shearings Holidays. *Picture: Shearings coach 636*. URL:  
<https://www.flickr.com/photos/shearings/13583388025/> (pages 16, 17).

# References: Images, Graphics II

- [9] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: <https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf> (pages 46–48).
- [10] Wes Breazzell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/> (pages 56–60).

# References: Literature I

- [2] Kenneth E. Hoff III et al. “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. ISBN: 0-201-48560-5. DOI: [10.1145/311535.311567](https://doi.org/10.1145/311535.311567). URL: <http://dx.doi.org/10.1145/311535.311567> (pages 3–7).
- [3] Chris McClanahan. “History and Evolution of GPU Architecture”. In: *A Survey Paper* (2010). URL: <http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf> (pages 3–7).
- [4] Jack Dongarra et al. *TOP500*. Nov. 2016. URL: <https://www.top500.org/lists/2016/11/> (pages 3–7).

# References: Literature II

- [5] Jack Dongarra et al. *Green500*. Nov. 2016. URL:  
<https://www.top500.org/green500/lists/2016/11/> (pages 3–7).