



# CUDA INTRODUCTION *PART I*

## GSP GPU COURSE 2018

8 August 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

- About CUDA Alternatives

- Directives

- Thrust

- CUDA C/C++



# History of GPUs

## 50 Shaders of Gray

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

# History of GPUs

## 50 Shaders of Gray

- 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

## 50 Shaders of Gray

- 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

## 50 Shaders of Gray

- 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

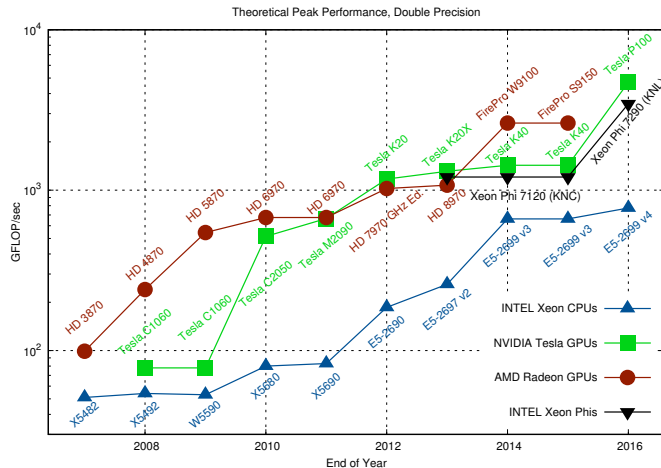
# History of GPUs

## 50 Shaders of Gray

- 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
- 2016 Top 500:  $> 1/10$  with GPUs [4], Green 500:  $\approx 2/3$  of top 50 with GPUs [5]

# Status Quo Across Architectures

## Performance

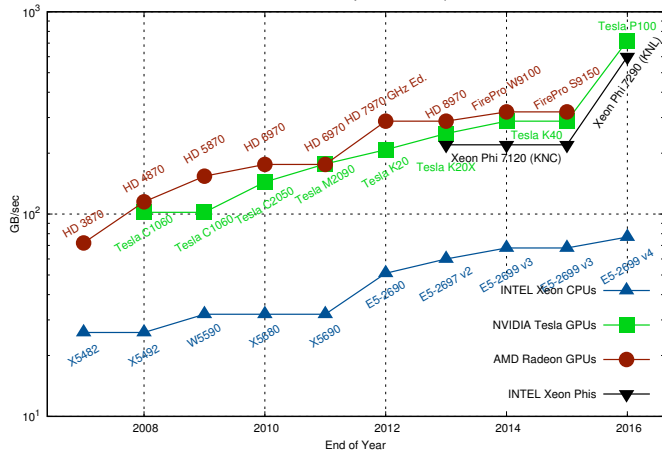


Graphic: Rupp [6]

# Status Quo Across Architectures

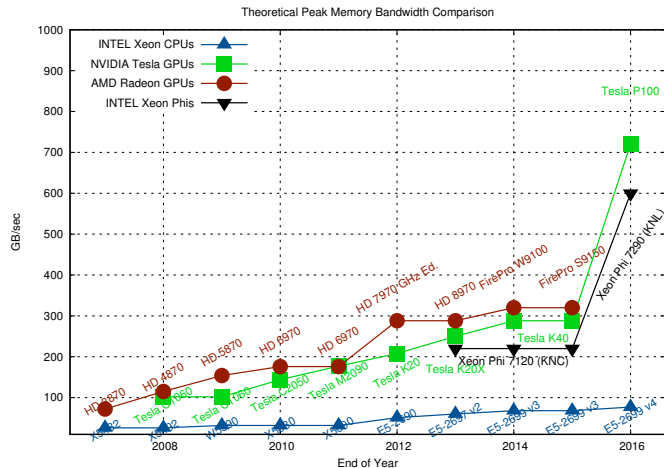
## Memory Bandwidth

Theoretical Peak Memory Bandwidth Comparison



Graphic: Rupp [6]

## Memory Bandwidth



Graphic: Rupp [6]



## 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



## JURON – A Human Brain Project *Prototype*

- 18 nodes with IBM POWER8NVL CPUs ( $2 \times 10$  cores)
- Per Node: 4 NVIDIA Tesla P100 cards, connected via NVLink
- GPU: 0.38 PFLOP/s peak performance





## JURON – A Human Brain Project *Prototype*

- 18 nodes with IBM POWER8NVL CPUs ( $2 \times 10$  cores)
- Per Node: 4 NVIDIA Tesla P100 cards, connected via NVLink
- GPU: 0.38 PFLOP/s peak performance



## JUWELS – Jülich's New Large System *just went online*

- 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-Basics/exercises/tasks/getting_started/`

See `Instructions.rst` for hints.

# Getting GPU-Acquainted

## Some Applications

TASK

GEMM

N-Body

Location of Code:

`1-Basics/exercises/tasks/getting_started/`

See `Instructions.rst` for hints.

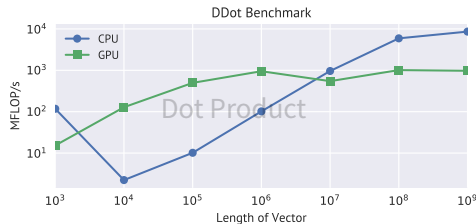
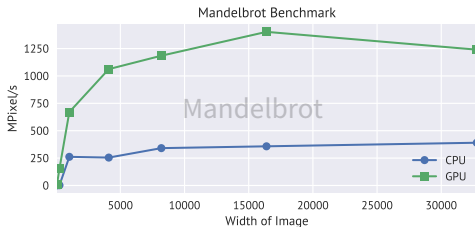
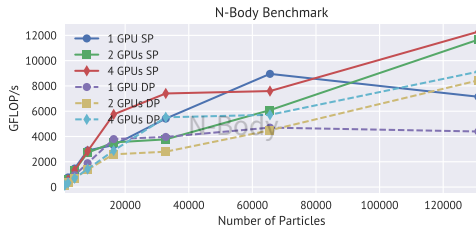
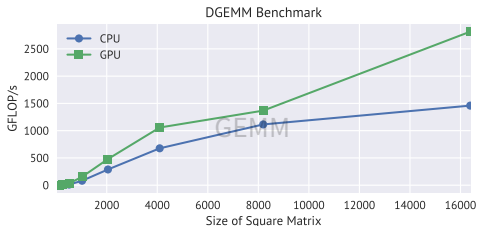
Mandelbrot

Dot Product

# Getting GPU-Acquainted

## Some Applications

TASK



# The GPU Platform

# CPU vs. GPU

A matter of specialties



# CPU vs. GPU

A matter of specialties



Transporting one

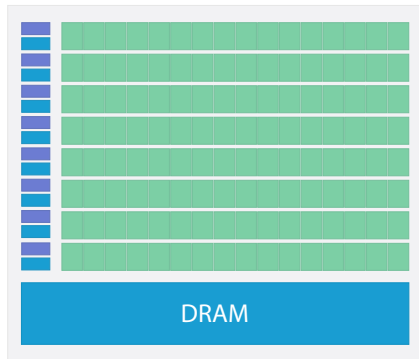
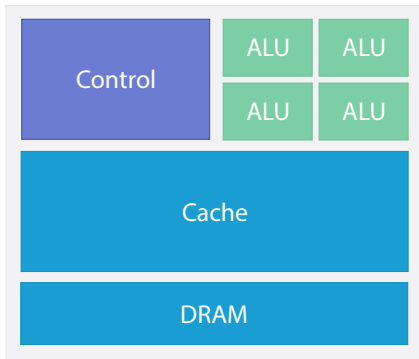


Transporting many



# 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

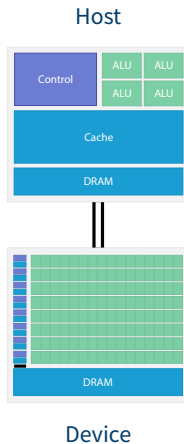
Asynchronicity

Memory

# Memory

GPU memory ain't no CPU memory

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

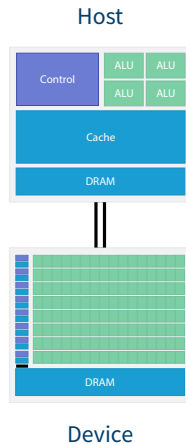


# Memory

## GPU memory ain't no CPU memory

Unified Virtual Addressing

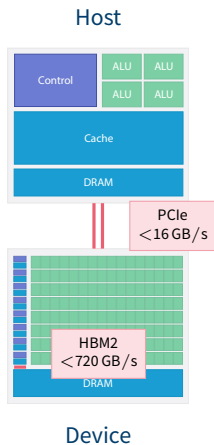
- GPU: accelerator / extension card
  - Separate device from CPU
- Separate memory, but UVA**



# Memory

## GPU memory ain't no CPU memory

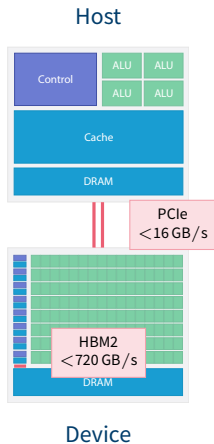
- GPU: accelerator / extension card
- Separate device from CPU  
**Separate memory, but UVA**



# 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!*





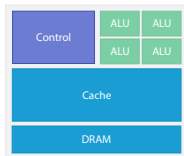
# Memory

## GPU memory ain't no CPU memory

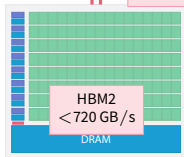
Unified Memory

- 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...?)

Host



PCIe  
< 16 GB/s

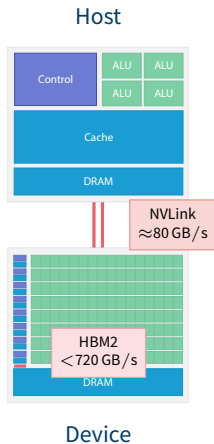


Device

# 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...?)



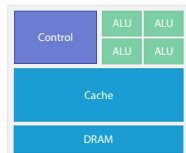
# Memory

## GPU memory ain't no CPU memory

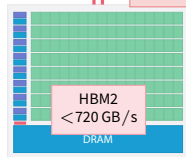
- GPU: accelerator / extension card
- Separate device from CPU
  - **Separate memory, but UVA and UM**
- Memory transfers need special consideration!  
*Do as little as possible!*
- Formerly: Explicitly copy data to/from GPU  
Now: Done automatically (performance...?)
- P100: 16 GB RAM, 720 GB/s; V100: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



Host



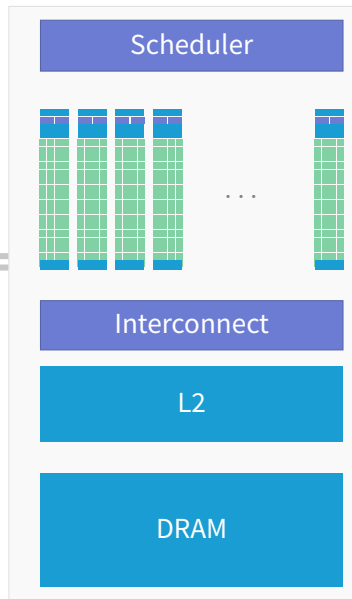
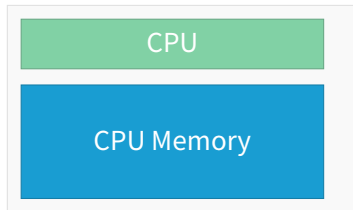
NVLink  
≈ 80 GB/s



Device

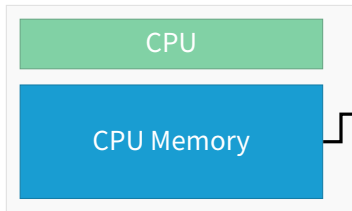
# Processing Flow

CPU → GPU → CPU

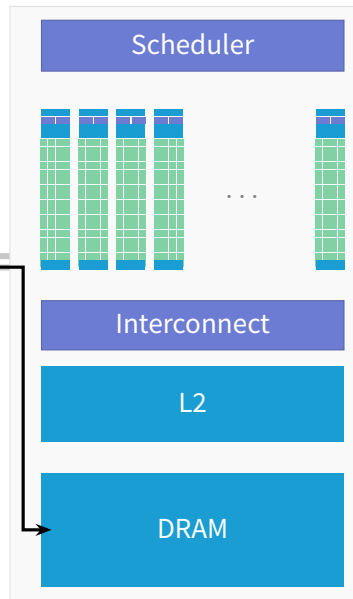


# Processing Flow

CPU → GPU → CPU

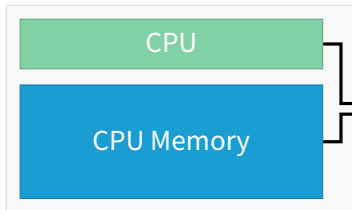


- 1 Transfer data from CPU memory to GPU memory

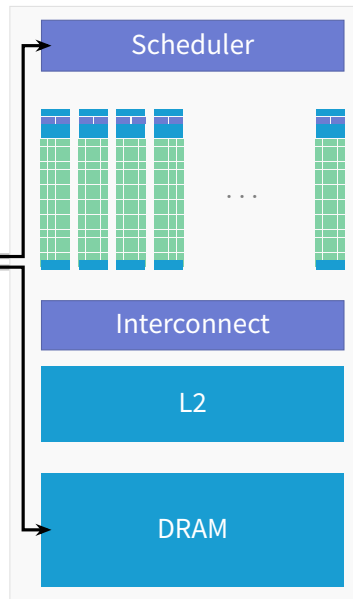


# Processing Flow

CPU → GPU → CPU

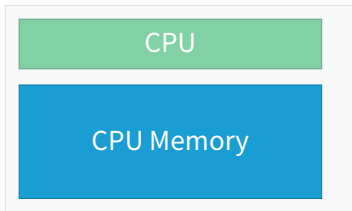


- 1 Transfer data from CPU memory to GPU memory, transfer program

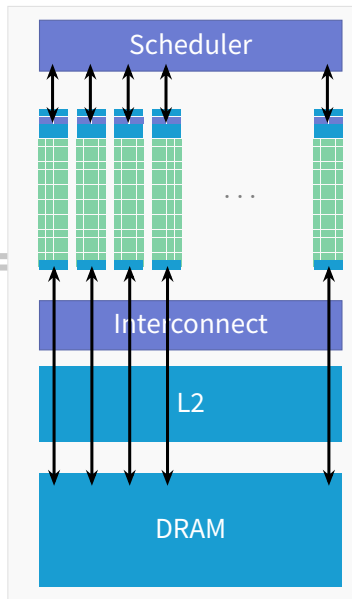


# Processing Flow

CPU → GPU → CPU

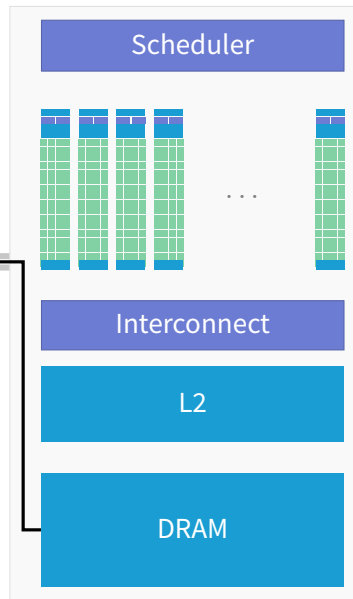
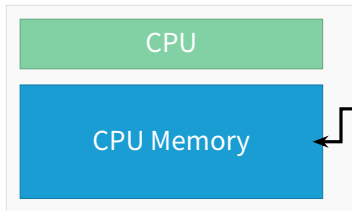


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back



# 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

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

*Scalar*

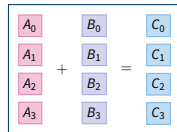
$A_0$	+	$B_0$	=	$C_0$
$A_1$	+	$B_1$	=	$C_1$
$A_2$	+	$B_2$	=	$C_2$
$A_3$	+	$B_3$	=	$C_3$

# SIMT

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

*Vector*

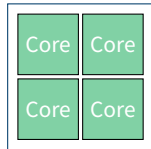
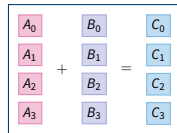


# SIMT

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

*Vector*



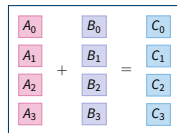


# SIMT

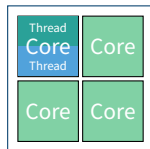
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

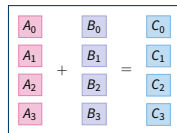


# SIMT

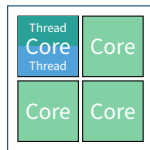
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

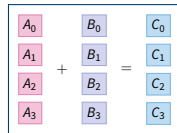


# SIMT

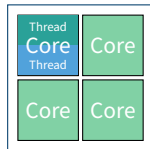
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

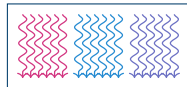
Vector



SMT




SIMT

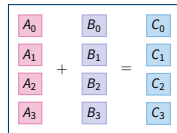


# SIMT

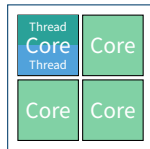
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

- 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 

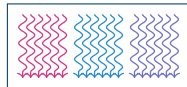
Vector



SMT



SIMT



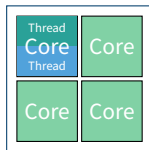
# SIMT



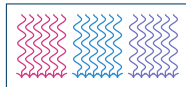
## Vector

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

## SMT



## SIMT



Graphics: Nvidia Corporation [9]

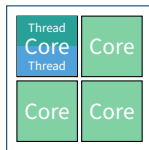
# SIMT



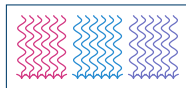
## Vector

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

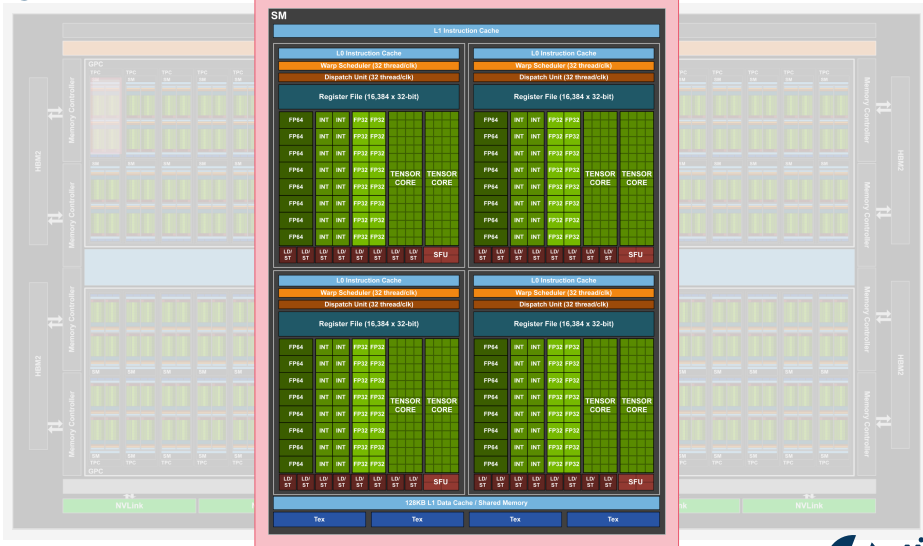
## SMT



## SIMT



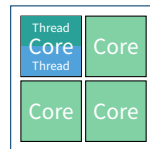
Graphics: Nvidia Corporation [9]



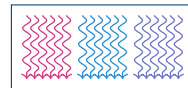
## Vector

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

*SMT*



SIMT



Graphics: Nvidia Corporation [9]

# 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



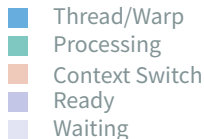
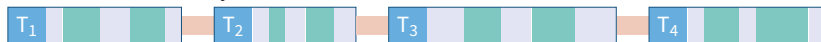
# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

**CPU** Minimizes latency within each thread

**GPU** Hides latency with computations from other thread warps

CPU Core: Low Latency



# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

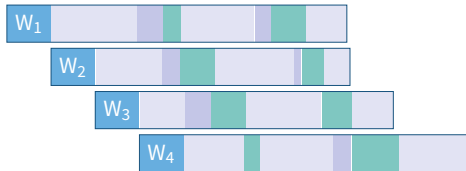
**CPU** Minimizes latency within each thread

**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

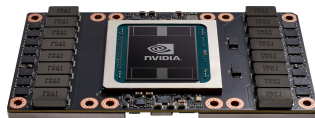
# CPU vs. GPU

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

# Preface: CPU

A simple CPU program!

SAXPY:  $\vec{y} = a\vec{x} + \vec{y}$ , with single precision

Part of LAPACK BLAS Level 1

```
void saxpy(int n, float a, float * x, float * y) {  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}
```

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
saxpy(n, a, x, y);
```



# 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!*



cuBLAS



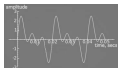
cuSPARSE



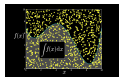
cuDNN



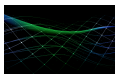
Numba



cuFFT



cuRAND



CUDA Math

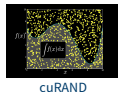
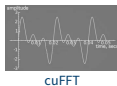


theano

# Libraries

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

*Use applications & libraries!*



Numba



theano



- GPU-parallel BLAS (all 152 routines)
- Single, double, complex data types
- Constant competition with Intel's MKL
- Multi-GPU support

→ <https://developer.nvidia.com/cublas>  
<http://docs.nvidia.com/cuda/cublas>

# cuBLAS

## Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);

cublasSaxpy(n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```



# cuBLAS

## Code example

```
int a = 42;  int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

# cuBLAS

## Code example

```
int a = 42;  int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

# cuBLAS

## Code example

```
int a = 42;  int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

Copy data to GPU

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

# cuBLAS

## Code example

```
int a = 42; int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

Copy data to GPU

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

Call BLAS routine

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```



# cuBLAS

## Code example

```
int a = 42; int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

Copy data to GPU

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

Call BLAS routine

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

Copy result to host

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

# cuBLAS

## Code example

```
int a = 42; int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;  
cudaMallocManaged(&d_x, n * sizeof(x[0]));  
cudaMallocManaged(&d_y, n * sizeof(y[0]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

Copy data to GPU

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

Call BLAS routine

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

Copy result to host

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

Finalize

## Implement a matrix-matrix multiplication

- Location of code: 1-Basics/exercercises/tasks/cublas/
- Look at `Instructions.rst` for instructions
  - 1 Implement call to double-precision GEMM of cuBLAS
  - 2 Build with make (CUDA needs to be loaded!)
  - 3 Run with `make run`  
Or `srun ./dgemm_um N, where N=100, 200, ...`
- Check [cuBLAS documentation](#) for details on `cublasDgemm()`

## JUWELS Getting Started

```
module load CUDA/9.1.85
salloc --partition=gpus --gres=mem192,gpu:4 -n 1
srun hostname
srun --pty --forward-x bash -i
```

# Programming GPUs

## About CUDA Alternatives

Libraries are not enough?

You think you want 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

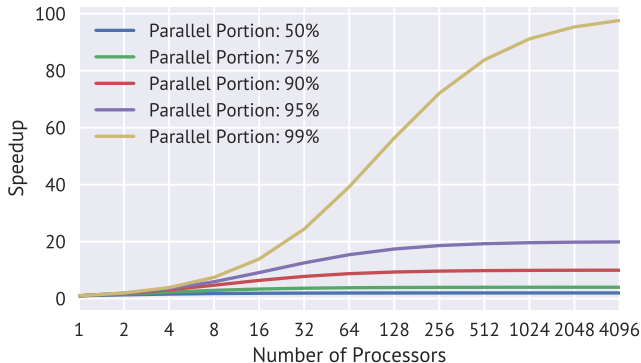
## Amdahl's Law

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**?

# Alternatives

## The twilight

There are alternatives to CUDA C, which **can** ease the *pain*...

- OpenACC
- Thrust
- PyCUDA

Other alternatives (for completeness)

- CUDA Fortran
- OpenMP
- OpenCL

# Alternatives

## The twilight

There are alternatives to CUDA C, which **can** ease the *pain*...

- **OpenACC**
- **Thrust**
- PyCUDA

Other alternatives (for completeness)

- CUDA Fortran
- OpenMP
- OpenCL

# Programming GPUs

## Directives

# GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

```
#pragma acc loop
```

```
for (int i = 0; i < 1; i++) {};
```

# GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

```
#pragma acc loop  
for (int i = 0; i < 1; i++) {};
```

- Also: Generalized API functions

```
acc_copy();
```

- Compiler interprets directives, creates according instructions

# GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

```
#pragma acc loop  
for (int i = 0; i < 1; i++) {};
```

- Also: Generalized API functions

```
acc_copy();
```

- Compiler interprets directives, creates according instructions

## Pro

- Portability
  - Other compiler? No problem! To it, it's a serial program
  - Different target architectures from same code
- Easy to program

## Con

- Only few compilers
- Not all the raw power available
- Harder to debug
- Easy to program wrong





# GPU Programming with Directives

The power of... two.

**OpenMP** Standard for multithread programming on CPU, GPU since 4.0, better since 4.5

```
#pragma omp target map(tofrom:y), map(to:x)
#pragma omp teams num_teams(10) num_threads(10)
#pragma omp distribute
for ( ) {
    #pragma omp parallel for
    for ( ) {
        // ...
    }
}
```

**OpenACC** Similar to OpenMP, but more specifically for GPUs  
For C/C++ and Fortran

# OpenACC

## Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma acc kernels  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```



# OpenACC

## Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma acc parallel loop copy(y) copyin(x)  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma acc parallel loop copy(y) copyin(x)  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma acc parallel loop copy(y) copyin(x)  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}
```

```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
saxpy_acc(n, a, x, y);
```

Tommorrow!

# Programming GPUs

## Thrust

# Thrust

Iterators! Iterators everywhere! 

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- Based on iterators
- Data-parallel primitives (`scan()`, `sort()`, `reduce()`, ...)
- Fully compatible with plain CUDA C (comes with **CUDA** Toolkit)

→ <http://thrust.github.io/>  
<http://docs.nvidia.com/cuda/thrust/>

# Thrust

## Code example

```
int a = 42;
int n = 10;
thrust::host_vector<float> x(n), y(n);
// fill x, y

thrust::device_vector d_x = x, d_y = y;

using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);

x = d_x;
```



# Thrust Task

TASK

Let's sort some randomness

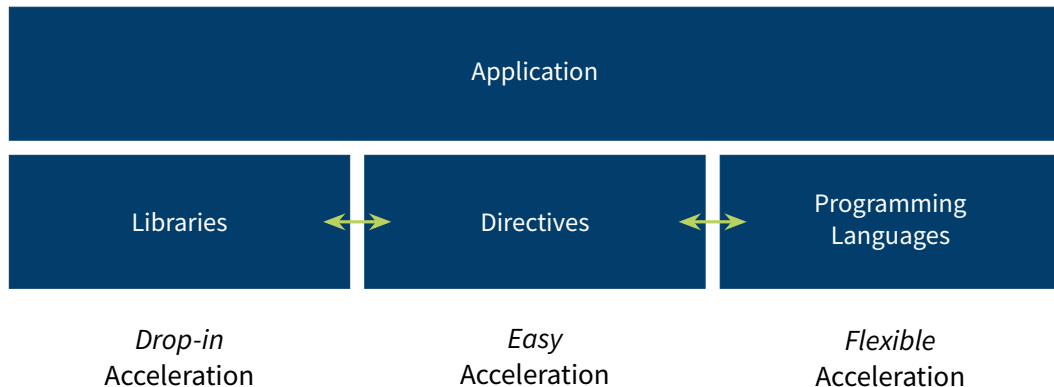
- Location of code: 1-Basics/excercises/tasks/thrust/
- Look at `Instructions.rst` for instructions

- 1 Sort random numbers with Thrust on CPU and GPU
- 2 Build with make (CUDA needs to be loaded!)
- 3 Run with make run

```
Orsrun -p gpus --gres=gpu:1 ./ThrustSort
```

- Check [Thrust documentation](#) for details on `thrust::sort()`

# Summary of Acceleration Possibilities



# Programming GPUs

## CUDA C/C++

# CUDA SAXPY

With runtime-managed data transfers

```
__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's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Thread



# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads



# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block





# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Block

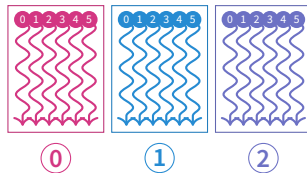


# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block
- Blocks



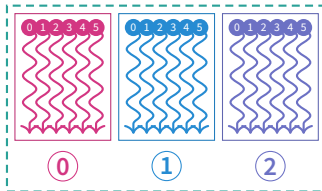
# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid



# CUDA's Parallel Model

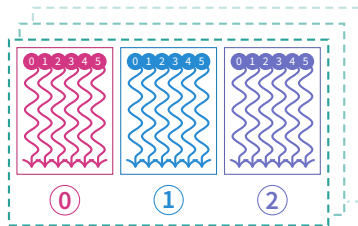
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



# CUDA's Parallel Model

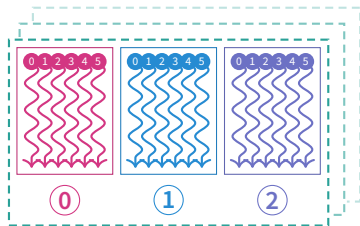
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



- Parallel function: **kernel**

- `__global__ kernel(int a, float * b) { }`

- Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...

- Execution entity: **threads**

- Lightweight → fast switching!

- 1000s threads execute simultaneously → order non-deterministic!

# CUDA API

?!?

→ Jan!

# Conclusions

## ...of Part 1

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- Thrust, OpenACC can give first entry point
- Full power with CUDA
- CUDA parallelizes for GPUs with many threads

# Conclusions

## ...of Part 1

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- Thrust, OpenACC can give first entry point
- Full power with CUDA
- CUDA parallelizes for GPUs with many threads

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





# APPENDIX

Appendix  
Glossary  
References

# Glossary I

**API** A programmatic interface to software by well-defined functions. Short for application programming interface. [82](#), [83](#), [84](#), [106](#), [113](#)

**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](#), [79](#), [80](#), [91](#), [95](#), [96](#), [97](#), [98](#), [99](#), [100](#), [101](#), [102](#), [103](#), [104](#), [105](#), [106](#), [107](#), [108](#), [112](#)

**JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. [111](#)

**JURECA** A multi-purpose supercomputer with 1800 nodes at JSC. [11](#)

## Glossary II

**JURON** One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. [12](#), [13](#)

**JUWELS** Jülich's new supercomputer, the successor of JUQUEEN. [14](#), [71](#)

**NVIDIA** US technology company creating GPUs. [3](#), [4](#), [5](#), [6](#), [7](#), [11](#), [12](#), [13](#), [14](#), [111](#), [112](#), [113](#), [114](#)

**NVLink** NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. [12](#), [13](#), [113](#)

**OpenACC** Directive-based programming, primarily for many-core machines. [79](#), [80](#), [85](#), [86](#), [87](#), [88](#), [89](#), [107](#), [108](#)

**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](#), [79](#), [80](#)

## Glossary III

- 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. [79](#), [80](#), [85](#)
- P100** A large **GPU** with the **Pascal** architecture from **NVIDIA**. It employs **NVLink** as its interconnect and has fast *HBM2* memory. [12](#), [13](#)
- Pascal** **GPU** architecture from **NVIDIA** (announced 2016). [113](#)
- POWER** **CPU** architecture from IBM, earlier: PowerPC. See also POWER8. [113](#)
- POWER8** Version 8 of IBM's **POWER**processor, available also under the OpenPOWER Foundation. [12](#), [13](#), [113](#)
- SAXPY** Single-precision  $A \times X + Y$ . A simple code example of scaling a vector and adding an offset. [57](#), [96](#)

## Glossary IV

**Tesla** The GPU product line for general purpose computing computing of NVIDIA. 11, 12, 13, 14

**Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 79, 80, 91, 93, 107, 108

**CPU** Central Processing Unit. 11, 12, 13, 14, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 57, 85, 93, 112, 113

**GPU** Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 56, 58, 59, 60, 61, 62, 63, 72, 73, 81, 82, 83, 84, 85, 90, 93, 95, 107, 108, 111, 112, 113, 114

**HBP** Human Brain Project. 112

# Glossary V

**SIMD** Single Instruction, Multiple Data. [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#)

**SIMT** Single Instruction, Multiple Threads. [22](#), [23](#), [24](#), [37](#), [38](#), [40](#), [41](#), [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#)

**SM** Streaming Multiprocessor. [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#)

**SMT** Simultaneous Multithreading. [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#)

# References 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. 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 II

- [5] Jack Dongarra et al. *Green500*. Nov. 2016. URL: <https://www.top500.org/green500/lists/2016/11/> (pages 3–7).
- [6] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/> (pages 8–10).
- [10] Wes Breazell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/> (pages 58–62).

# References: Images, Graphics I

- [1] Igor Ovsyannykov. *Yarn*. Freely available at Unsplash. URL: <https://unsplash.com/photos/hvILKk7SlH4>.
- [7] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (pages 19, 20).
- [8] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (pages 19, 20).
- [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 49–51).