GPU Introduction
JSC OpenACC Course 2018

29 October 2018 | Andreas Herten | Forschungszentrum Jülich

Member of the Helmholtz Association

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

Outline

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

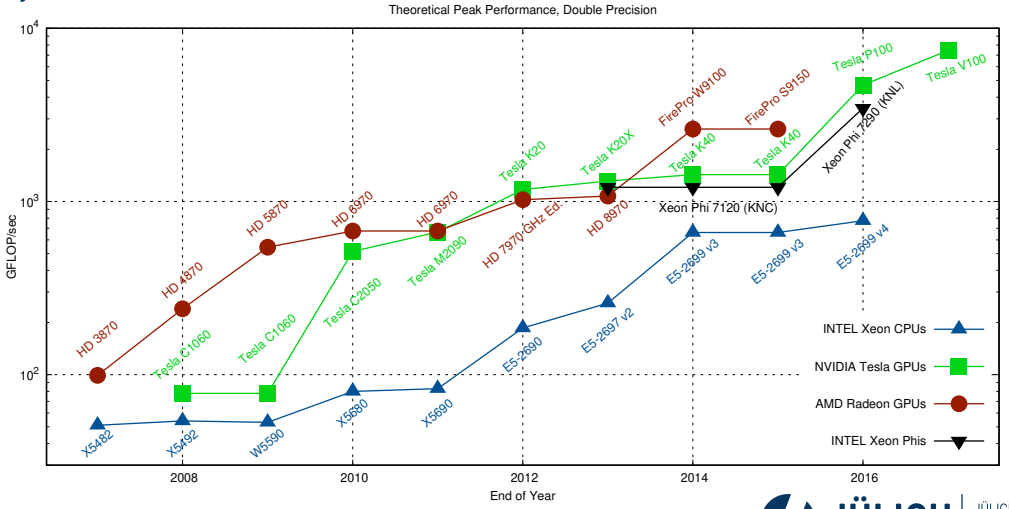
2007 CUDA

2009 OpenCL

2018 Top 500: 20 % with GPUs (#1, #3) [4], Green 500: 7 of top 10 with GPUs [5]

Status Quo Across Architectures

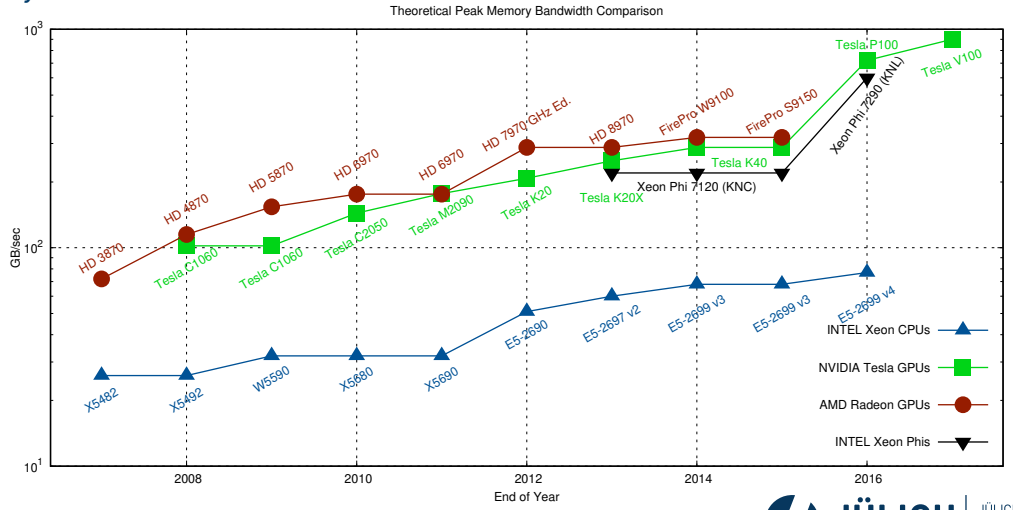
Memory Bandwidth



Graphic: Rupp [6]

Status Quo Across Architectures

Memory Bandwidth



Graphic: Rupp [6]

JURECA – Jülich's Multi-Purpose Supercomputer

- 1872 nodes with Intel Xeon E5 CPUs (2×12 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 (2×24 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

GEMM

N-Body

Location of Code:
`1-Introduction-…/Tasks/getting-started/`

See `Instructions.md` for hints.

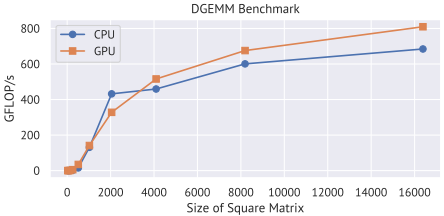
Mandelbrot

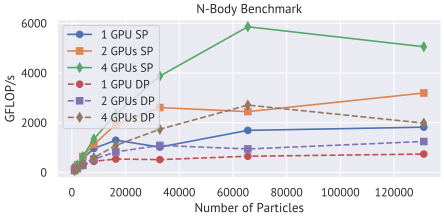
Dot Product

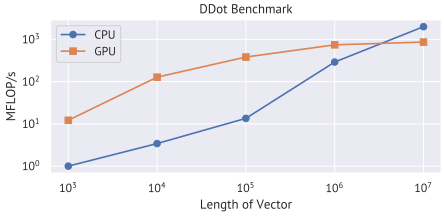
Getting GPU-Acquainted

Some Applications

TASK







JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

The GPU Platform

CPU vs. GPU

A matter of specialties

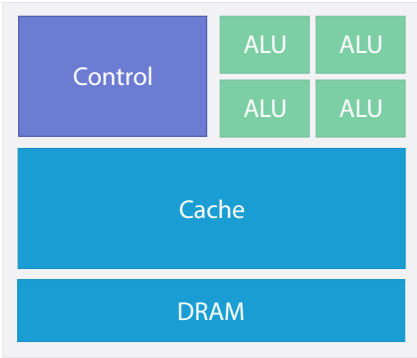
Transporting one

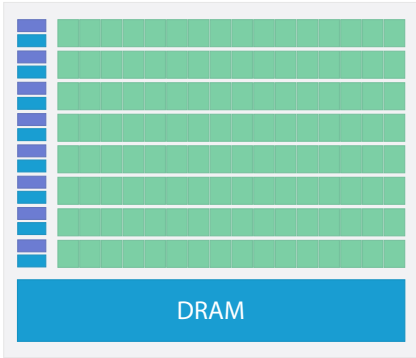
Transporting many

Graphics: Lee [7] and Shearings Holidays [8]

CPU vs. GPU

Chip





GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

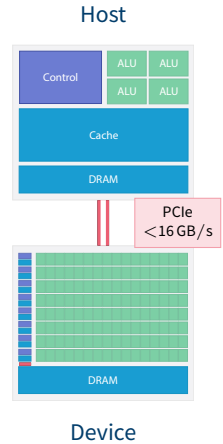
Memory

Memory

GPU memory ain't no CPU memory

Unified Virtual Addressing

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



Memory

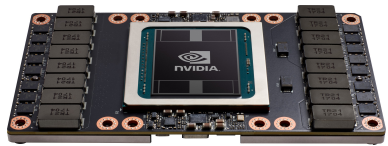
GPU memory ain't no CPU memory

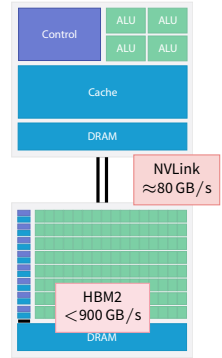
Unified Memory

- GPU: accelerator / extension card

→ Separate device from CPU
Separate memory, but UVA and UM

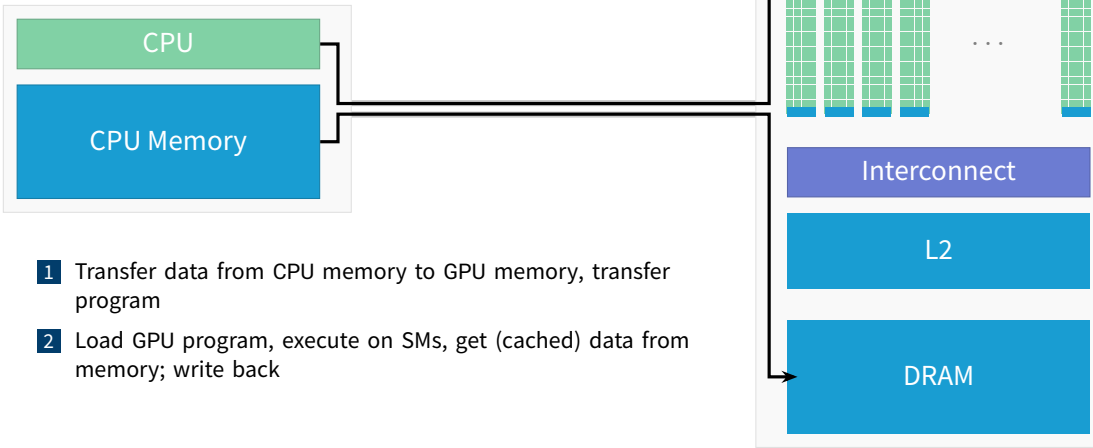
- Memory transfers need special consideration!
  Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
  Now: Done automatically (performance…?)
- P100: 16 GB RAM, 720 GB/s; V100: 16 (32) GB RAM, 900 GB/s





Processing Flow

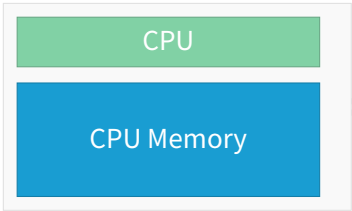
CPU → GPU → CPU

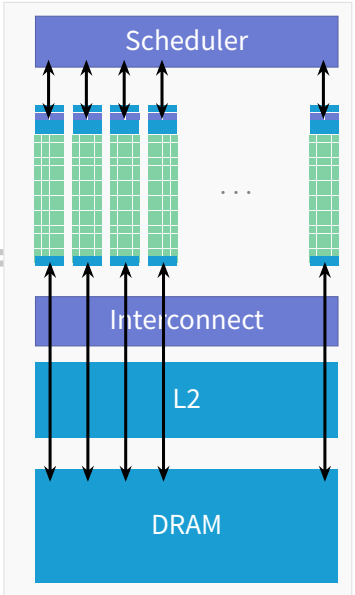


1. Transfer data from CPU memory to GPU memory, transfer program
2. Load GPU program, execute on SMs, get (cached) data from memory; write back

Processing Flow

CPU → GPU → CPU

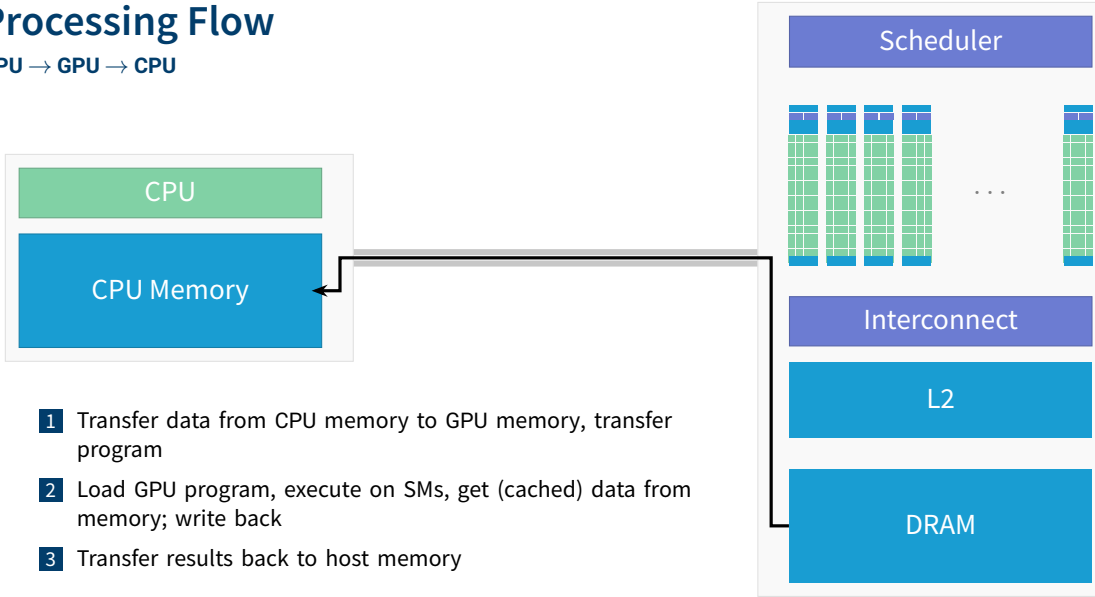




1. Transfer data from CPU memory to GPU memory, transfer program
2. Load GPU program, execute on SMs, get (cached) data from memory; write back

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

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

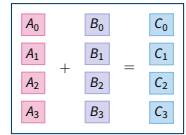
Memory

SIMT

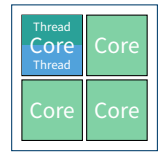
SIMT = SIMD ⊕ SMT

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

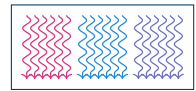
Vector



SMT

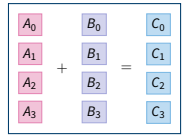


SIMT

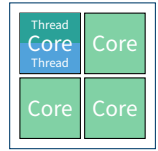


SIMT

Vector

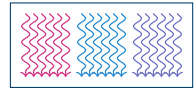


SMT



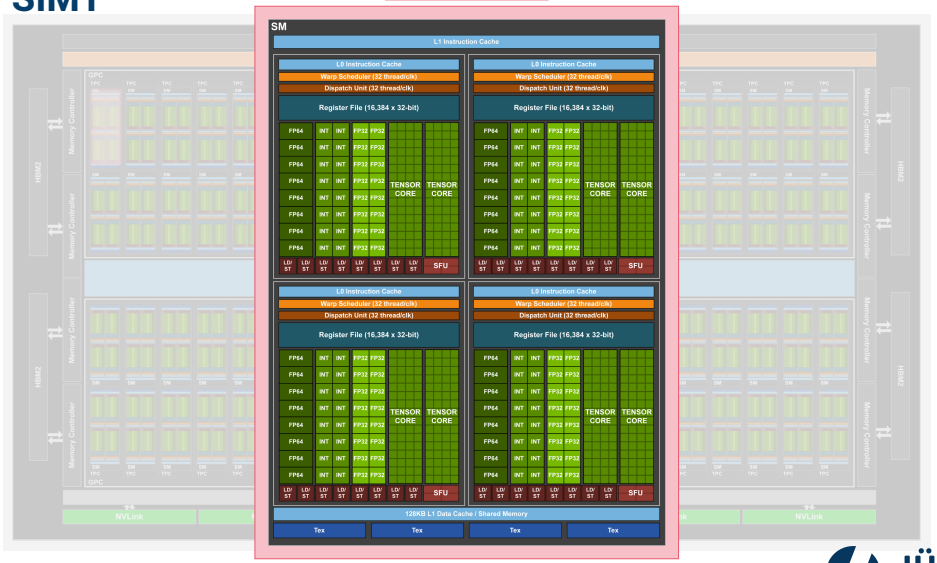
Graphics: Nvidia Corporation [9]

SIMT

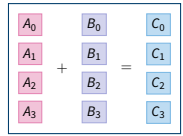


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

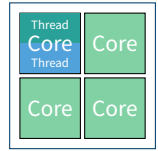
SIMT



Vector

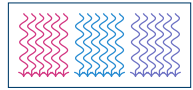


SMT



Graphics: Nvidia Corporation [9]

SIMT



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

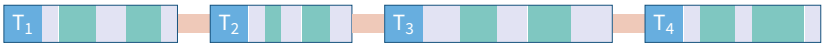
Low Latency vs. High Throughput

Maybe GPU's ultimate feature

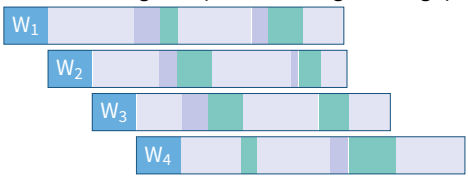
CPU Minimizes latency within each thread

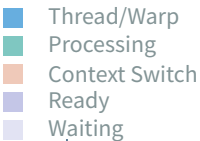
GPU Hides latency with computations from other thread warps

CPU Core: Low Latency



GPU Streaming Multiprocessor: High Throughput



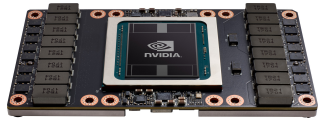


CPU vs. GPU

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

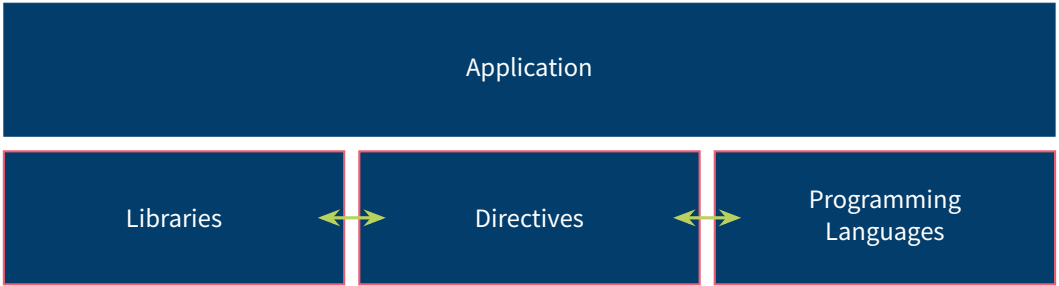


Optimized for **high throughput**

- \+ 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

Libraries

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

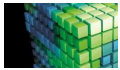
Use applications & libraries

Wizard: Breazell [10]

Libraries

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

Use applications & libraries


cuBLAS

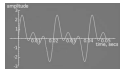

cuSPARSE

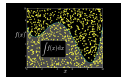
cuDNN

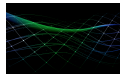
{} ArrayFire

Thrust

Numba


cuFFT


cuRAND


CUDA Math

theano

Wizard: Breazell [10]

Summary of Acceleration Possibilities

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Parallelism

Libraries are not enough?

You need to write your own GPU code?

Primer on Parallel Scaling

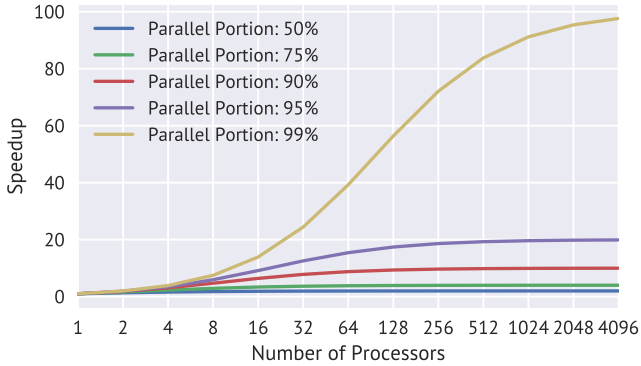
Amdahl's Law

Possible maximum speedup for N parallel processors

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

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

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

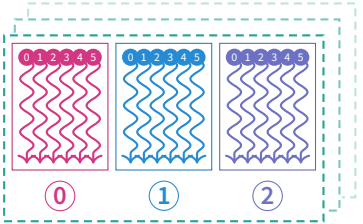
Call kernel 2 blocks, each 5 threads

Wait for kernel to finish

CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:
  - Thread → Block
  - Block → Grid
  - Threads & blocks in 3D



- Execution entity: **threads**
  - Lightweight → fast switchting!
  - 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

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

Appendix

- Glossary
- References

Glossary I

API A programmatic interface to software by well-defined functions. Short for application programming interface. 43

ATI Canada-based GPUs manufacturing company; bought by AMD in 2006. 3

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 3, 36, 37, 38, 40, 43

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 42

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. 6

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 7

NVIDIA US technology company creating GPUs. 3, 6, 7, 42, 43

Glossary II

OpenACC Directive-based programming, primarily for many-core machines. 1, 36, 38

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 3, 36

OpenGL The *Open Graphics Library*, an API for rendering graphics across different hardware architectures. 3

OpenMP Directive-based programming, primarily for multi-threaded machines. 36

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 37

Tesla The GPU product line for general purpose computing computing of NVIDIA. 6, 7

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 36

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 4, 5).

[7] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (page 11).

[8] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (page 11).

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 24, 25).

[10] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 30, 31).

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. URL: http://dx.doi.org/10.1145/311535.311567 (page 3).

[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 (page 3).

[4] Jack Dongarra et al. *TOP500*. Nov. 2016. URL: https://www.top500.org/lists/2016/11/ (page 3).

References: Literature II

[5] Jack Dongarra et al. *Green500*. Nov. 2016. URL: https://www.top500.org/green500/lists/2016/11/ (page 3).