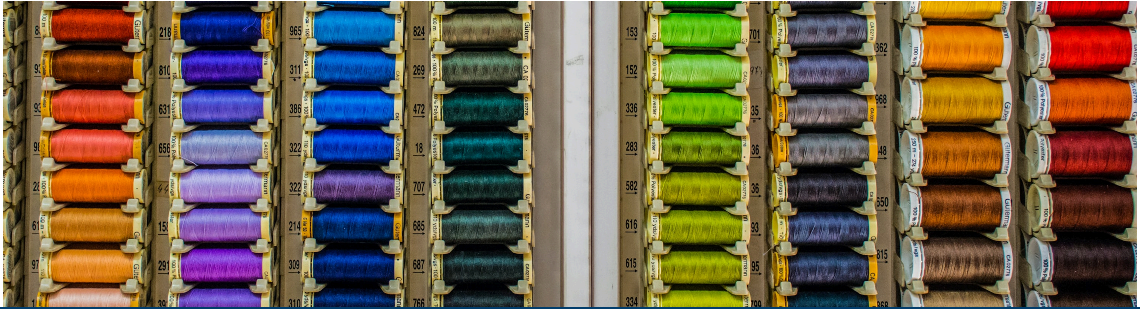


# GPU Introduction

## JSC OpenACC Course 2023

24 October 2023 | Andreas Herten | Forschungszentrum Jülich

# Outline

# History of GPUs

## A short but unparalleled 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 unparalleled 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 unparalleled 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 unparalleled 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 unparalleled 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

2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]

# History of GPUs

## A short but unparalleled 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
- 2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]
- 2023 🇪🇺: Leonardo (238 PFLOP/s*, Italy), NVIDIA GPUs; LUMI (309 PFLOP/s*, Finland), AMD GPUs
  🇺🇸: Frontier (1.102 EFLOP/s*, ORNL), AMD GPUs

*: *Effective* FLOP/s, *not theoretical peak (HPL* $R_{max}$*)*

# History of GPUs

## A short but unparalleled 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

2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]

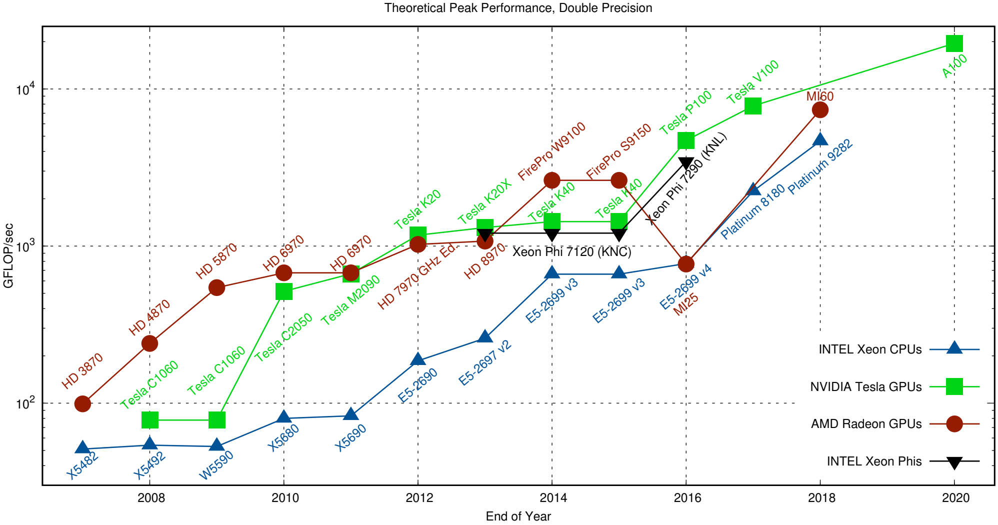
2023 : Leonardo (238 PFLOP/s*, Italy), NVIDIA GPUs; LUMI (309 PFLOP/s*, Finland), AMD GPUs
: Frontier (1.102 EFLOP/s*, ORNL), AMD GPUs

*Soon* : JUPITER (≈ 1 EFLOP/s, NVIDIA GPUs, JSC)
: Aurora (≈ 2 EFLOP/s, Argonne), Intel GPUs; El Capitan (≈ 2 EFLOP/s, LLNL), AMD GPUs

*: *Effective* FLOP/s, *not theoretical peak (HPL $R_{max}$)*

# Status Quo Across Architectures

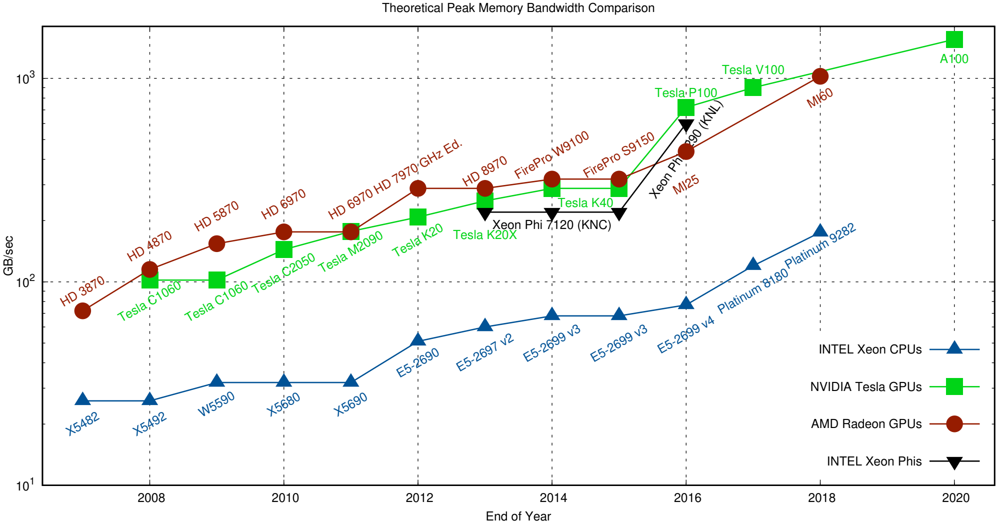
## Performance




Graphic: Rupp [6]

# Status Quo Across Architectures

## Memory Bandwidth




Graphic: Rupp [6]


## JUWELS Cluster – Jülich's Scalable System

- 2500 nodes with Intel Xeon CPUs (2 $\times$ 24 cores)
- 46 $+$ 10 nodes with 4 NVIDIA Tesla V100 cards (16 GB memory)
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance (Top500: #86)


## JUWELS Booster – Scaling Higher!

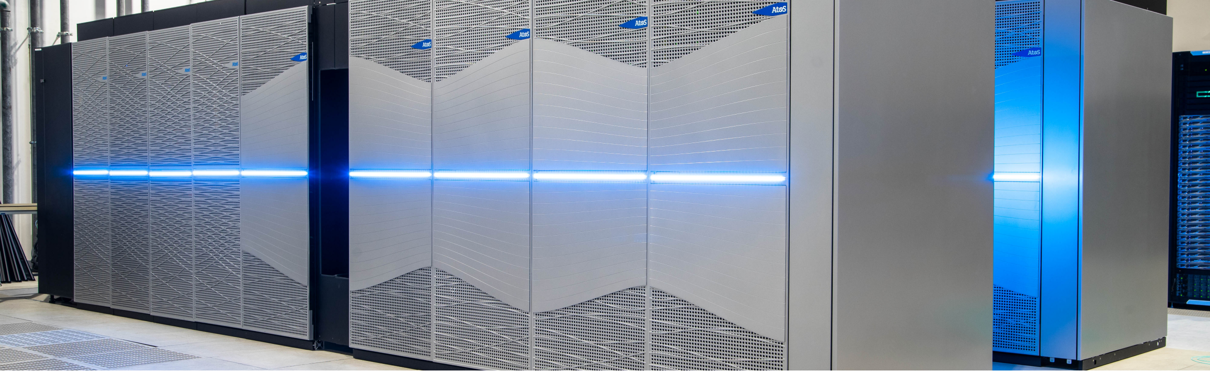
- 936 nodes with AMD EPYC Rome CPUs (2 × 24 cores)
- Each with 4 NVIDIA A100 Ampere GPUs (each: *FP64TC:* 19.5 / *FP64:* 9.7 TFLOP/s, 40 GB memory)
- InfiniBand DragonFly+ HDR-200 network; 4 × 200 Gbit/s per node


**Top500 List Nov 2020:**

- #1 Europe
- #7 World
- #4* Top/Green500

## JUWELS Booster – Scaling Higher!

- 936 nodes with AMD EPYC Rome CPUs ($2 \times 24$ cores)
- Each with 4 NVIDIA A100 Ampere GPUs (each: *FP64TC:* 19.5 / *FP64:* 9.7 TFLOP/s, 40 GB memory)
- InfiniBand DragonFly+ HDR-200 network; $4 \times 200$ Gbit/s per node




## JURECA DC – Multi-Purpose

- 768 nodes with AMD EPYC Rome CPUs (2 × 64 cores)
- 192 nodes with 4 NVIDIA A100 Ampere GPUs
- InfiniBand DragonFly+ HDR-100 network


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

# Getting GPU-Acquainted

## Some Applications

TASK

Location of Code:

`1-Introduction-GPU-Programming/Tasks/getting-started`

See `Instructions.iypnb` for hints.

*Make sure to have sourced the course environment!*

# Getting GPU-Acquainted

## Some Applications

TASK

GEMM

N-Body

Location of Code:

`1-Introduction-GPU-Programming/Tasks/getting-started`

See `Instructions.iypnb` for hints.

*Make sure to have sourced the course environment!*

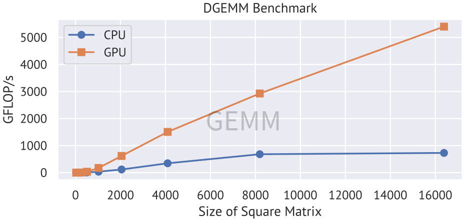
Mandelbrot

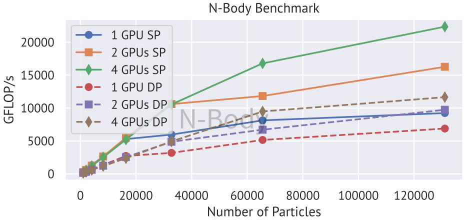
Dot Product

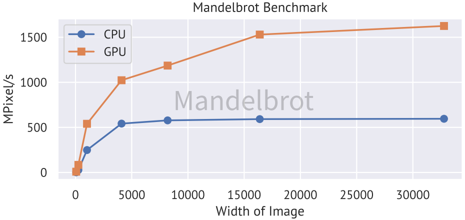
# Getting GPU-Acquainted

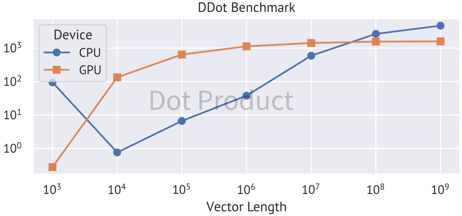
TASK

## Some Applications



# Platform

# CPU vs. GPU

## A matter of specialties



Graphics: Lee [8] and Shearings Holidays [9]

# CPU vs. GPU

## A matter of specialties


Transporting one


Transporting many

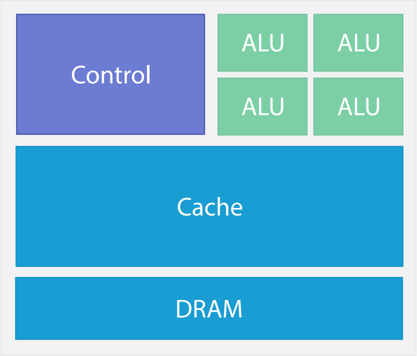
Graphics: Lee [8] and Shearings Holidays [9]

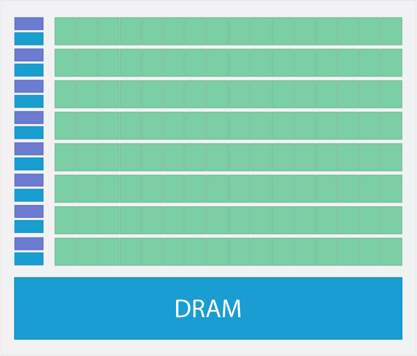


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

# 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

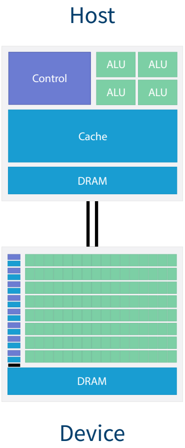
**Memory**

# Memory

## GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU



# Memory

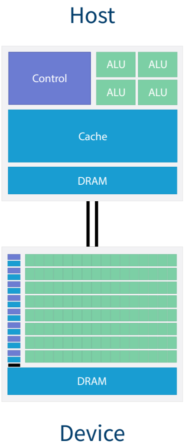
## GPU memory ain't no CPU memory

*Unified Virtual Addressing*

- GPU: accelerator / extension card

→ Separate device from CPU

**Separate memory, but UVA**






JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

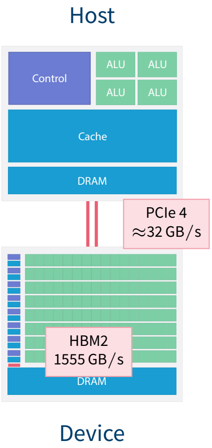
# Memory

## 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

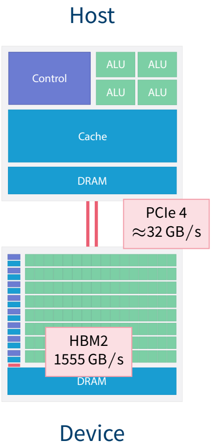
- GPU: accelerator / extension card

→ 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

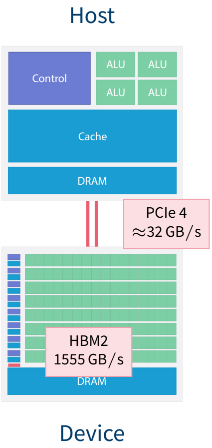
→ Separate device from CPU

**Separate memory, but UVA and UM**

- Memory transfers need special consideration!

  *Do as little as possible!*

- Choice: automatic transfers (convenience) or manual transfers (control)



# Memory

## GPU memory ain’t no CPU memory

- GPU: accelerator / extension card

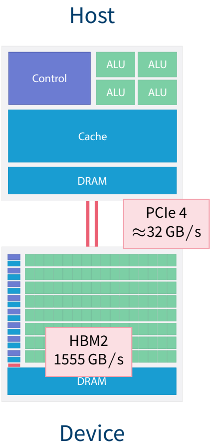
→ Separate device from CPU

**Separate memory, but UVA and UM**

- Memory transfers need special consideration!

  *Do as little as possible!*

- Choice: automatic transfers (convenience) or manual transfers (control)



# Memory

## GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU

**Separate memory, but UVA and UM**

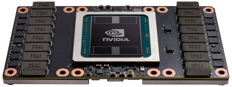
- Memory transfers need special consideration!

  *Do as little as possible!*

- Choice: automatic transfers (convenience) or manual transfers (control)

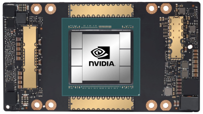
**V100**

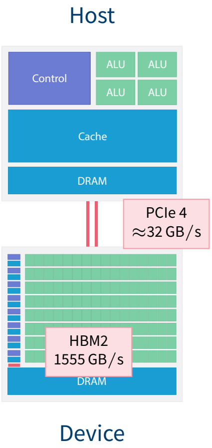
32 GB RAM, 900 GB/s



**A100**

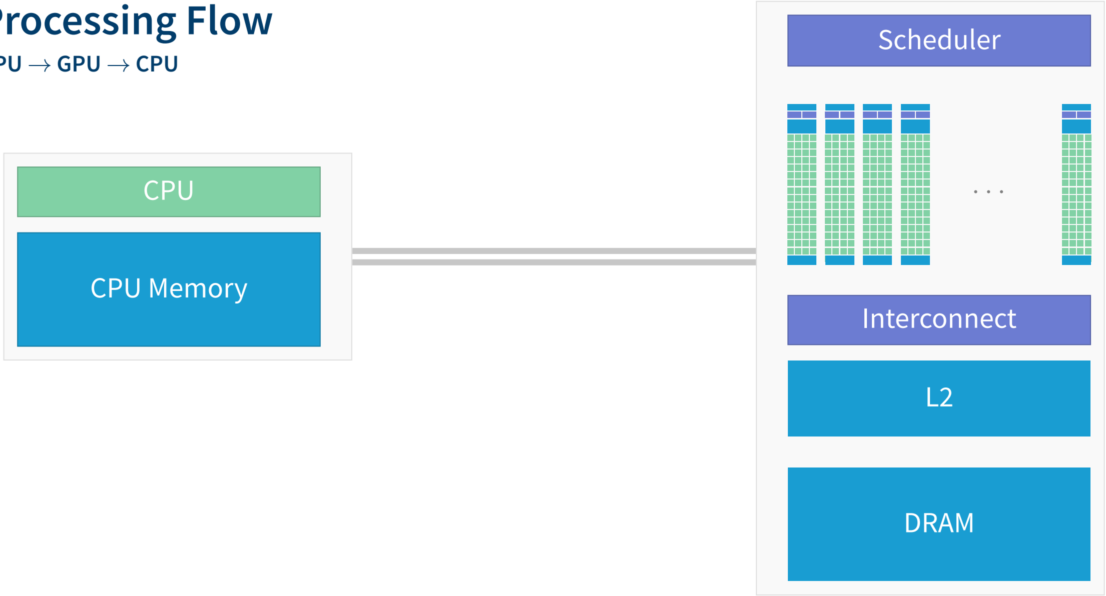
40 GB RAM, 1555 GB/s





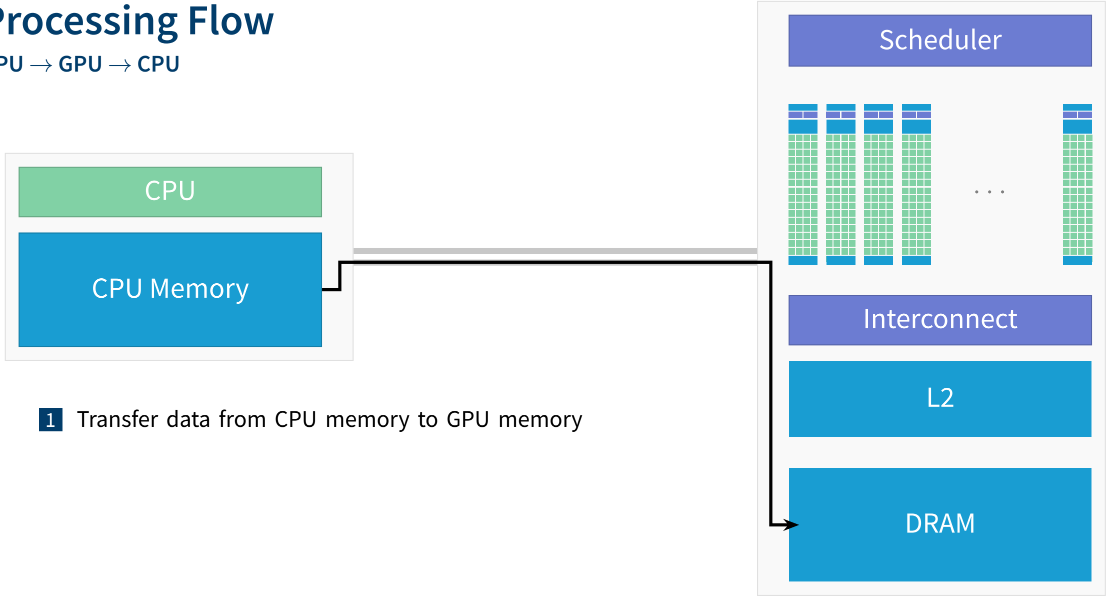
# Processing Flow

**CPU → GPU → CPU**



# Processing Flow

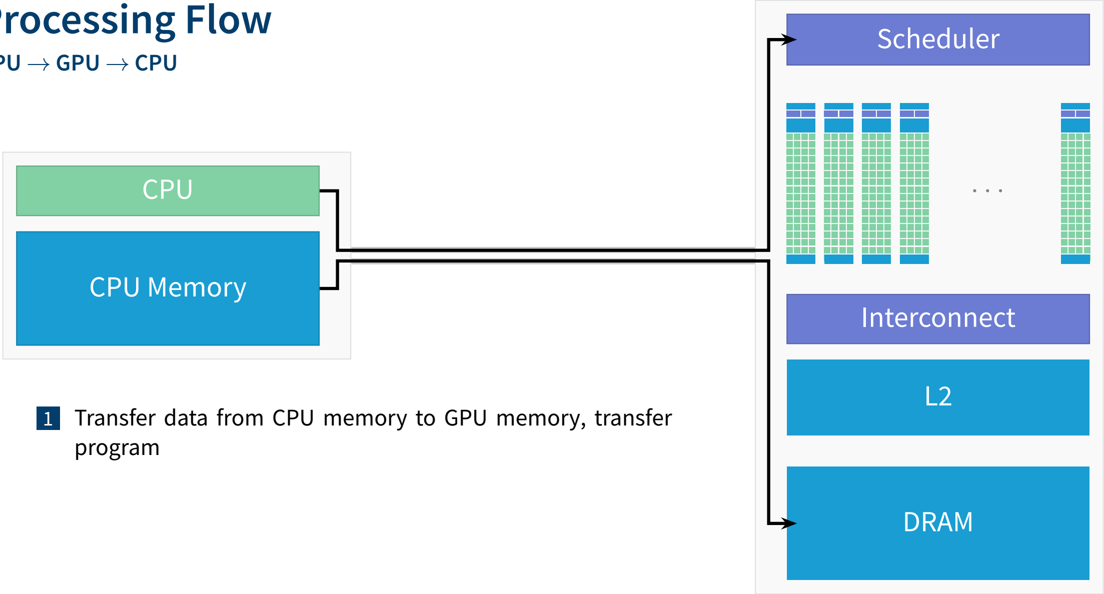
## CPU → GPU → CPU




1. Transfer data from CPU memory to GPU memory

# Processing Flow

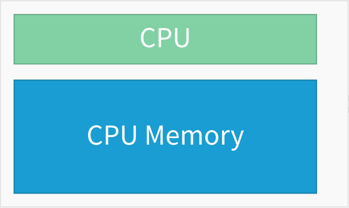
## CPU $\rightarrow$ GPU $\rightarrow$ CPU

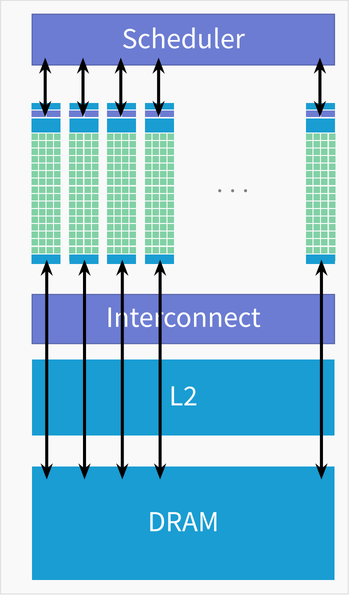



1. Transfer data from CPU memory to GPU memory, transfer program

# Processing Flow

## CPU → GPU → CPU

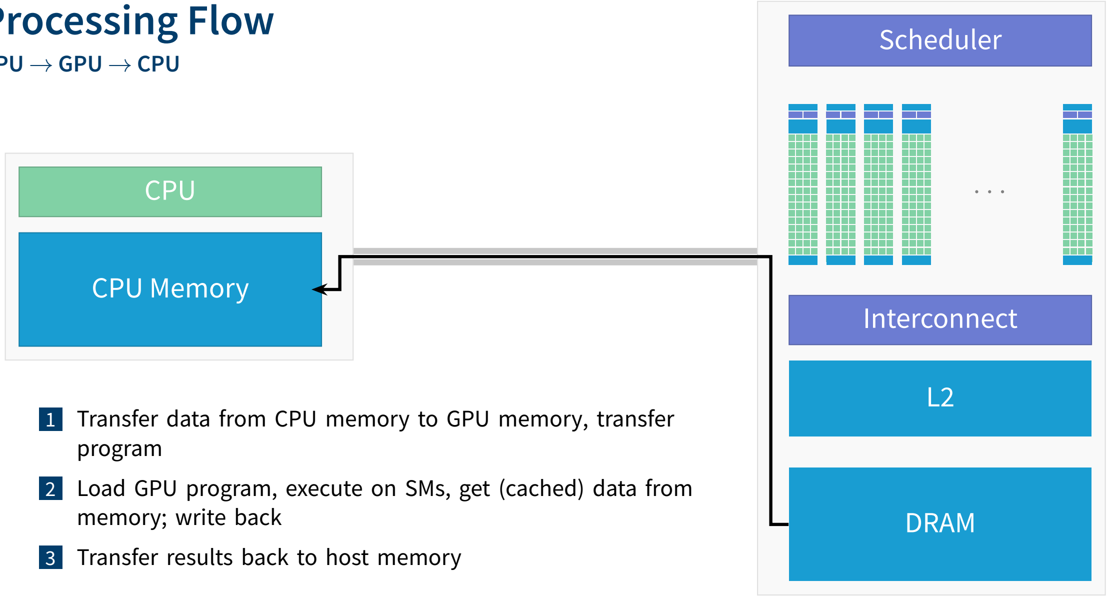



1. Transfer data from CPU memory to GPU memory, transfer program
2. Load GPU program, execute on SMs, get (cached) data from memory; write back

# 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**

# Flynn’s Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements

# Flynn's Taxonomy

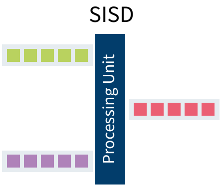
- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix}\text{Single}\\\text{Multiple}\end{pmatrix} \otimes \begin{pmatrix}\text{Instruction}\\\text{Data}\end{pmatrix}$

# Flynn’s Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix}\textbf{S}\text{ingle}\\ \textbf{M}\text{ultiple}\end{pmatrix} \otimes \begin{pmatrix}\textbf{I}\text{nstruction}\\ \textbf{D}\text{ata}\end{pmatrix}$

  SISD Single Instruction, Single Data

# Flynn’s Taxonomy



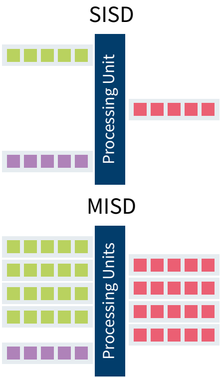

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix}\textbf{Single}\\ \textbf{Multiple}\end{pmatrix} \otimes \begin{pmatrix}\text{Instruction}\\ \text{Data}\end{pmatrix}$

  SISD Single Instruction, Single Data

# Flynn's Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix}\text{Single}\\ \text{Multiple}\end{pmatrix} \otimes \begin{pmatrix}\text{Instruction}\\ \text{Data}\end{pmatrix}$

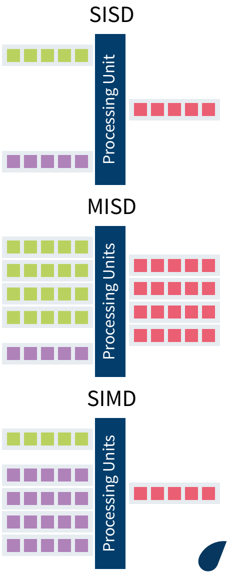
  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data



# Flynn's Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix} \text{Single} \\ \text{Multiple} \end{pmatrix} \otimes \begin{pmatrix} \text{Instruction} \\ \text{Data} \end{pmatrix}$

  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data
  SIMD Single Instruction, Multiple Data



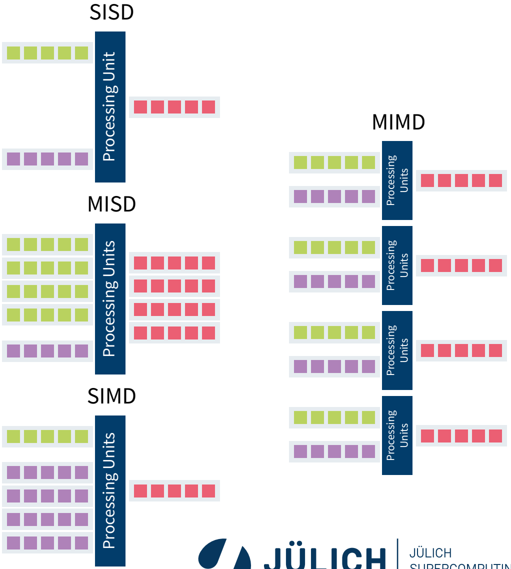


JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

# Flynn's Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix}\text{Single}\\\text{Multiple}\end{pmatrix} \otimes \begin{pmatrix}\text{Instruction}\\\text{Data}\end{pmatrix}$

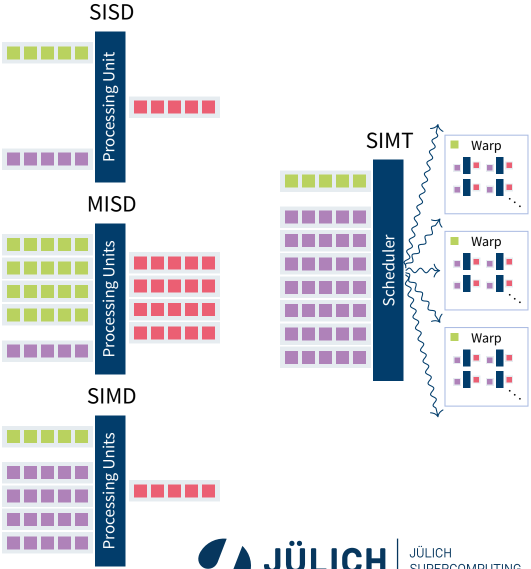
  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data
  SIMD Single Instruction, Multiple Data
  MIMD Multiple Instructions, Multiple Data



# Flynn’s Taxonomy

- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix} \text{Single} \\ \text{Multiple} \end{pmatrix} \otimes \begin{pmatrix} \text{Instruction} \\ \text{Data} \end{pmatrix}$

  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data
  SIMD Single Instruction, Multiple Data
  MIMD Multiple Instructions, Multiple Data
  SIMT Single Instruction, Multiple *Threads*



# SIMT

## SIMT $=$ SIMD $\oplus$ SMT

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

*Scalar*

$A_0 + B_0 = C_0$

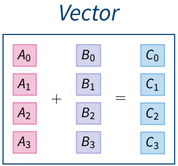
$A_1 + B_1 = C_1$

$A_2 + B_2 = C_2$

$A_3 + B_3 = C_3$

# SIMT

## SIMT $=$ SIMD $\oplus$ SMT

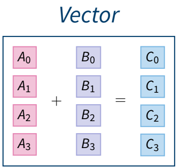


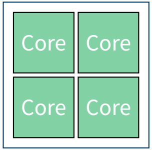

- CPU:
  - Single Instruction, Multiple Data (SIMD)

# SIMT

## SIMT $=$ SIMD $\oplus$ SMT

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



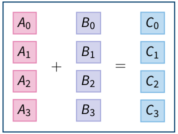



# SIMT

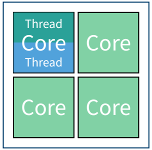
## SIMT $=$ SIMD $\oplus$ SMT

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

*Vector*




*SMT*

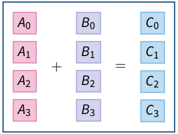


# SIMT

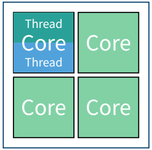
## SIMT = SIMD ⊕ SMT

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

*Vector*




*SMT*

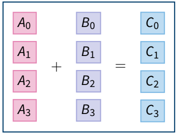


# SIMT

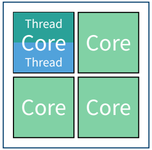
## SIMT $=$ SIMD $\oplus$ SMT

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

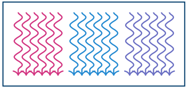
*Vector*




*SMT*




*SIMT*

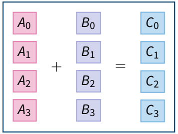


# SIMT

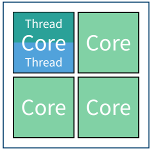
## SIMT $=$ SIMD $\oplus$ SMT

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

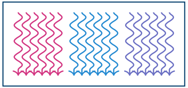

*Vector*




*SMT*

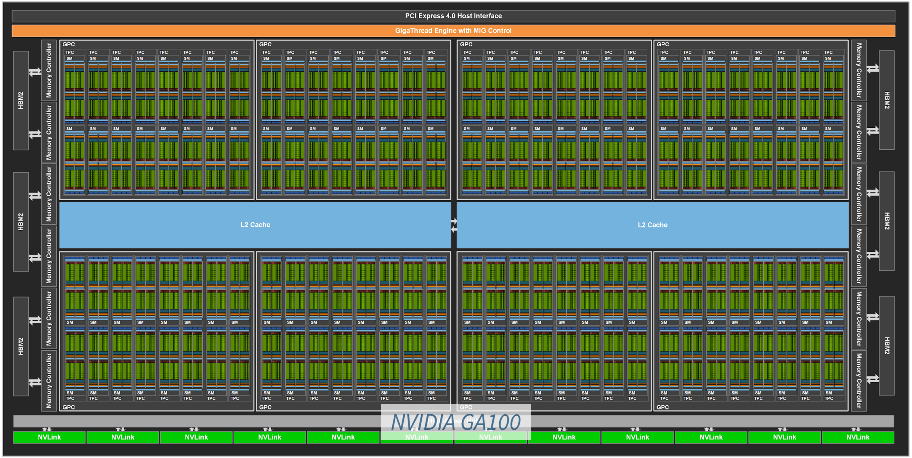



*SIMT*

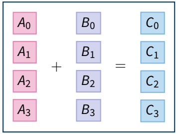


# SIMT

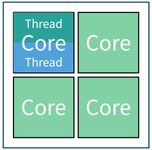
## SIMT = SIMD ⊕ SMT



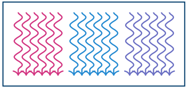

*Vector*




*SMT*



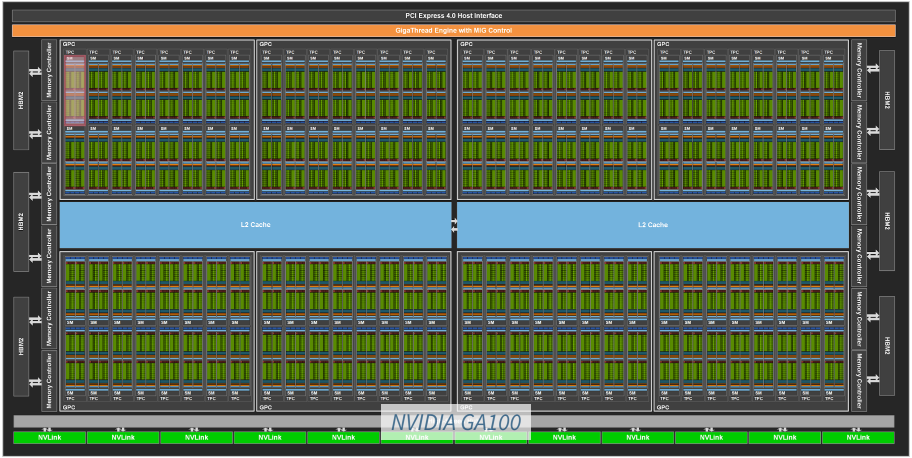

*SIMT*

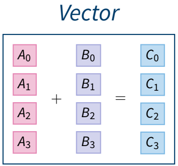


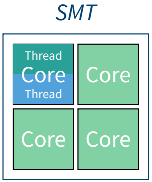
Graphics: img:amperepictures

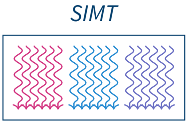
# SIMT

## SIMT $=$ SIMD $\oplus$ SMT



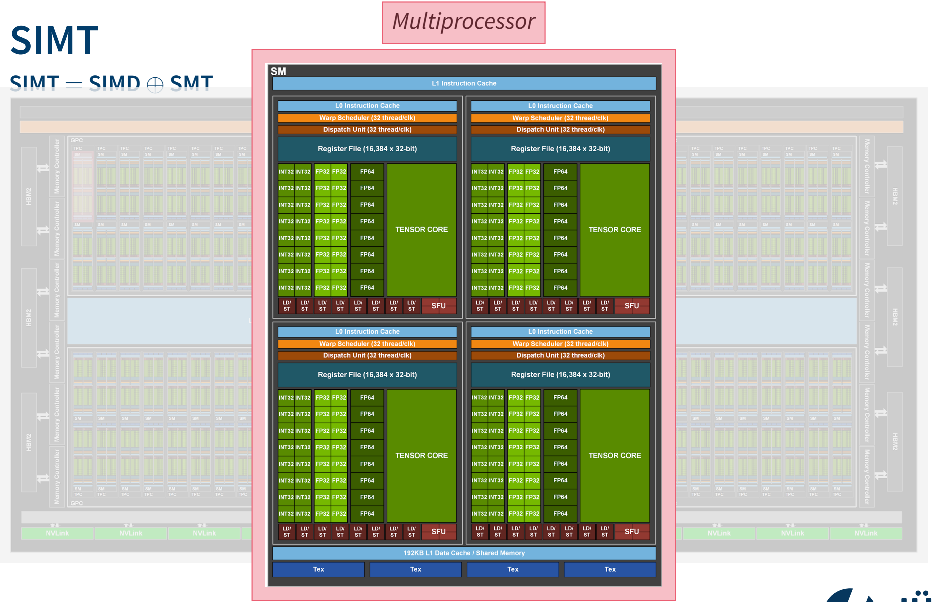



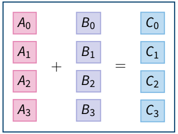

Graphics: img:amperepictures

# SIMT

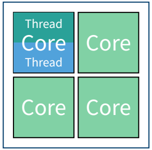
## SIMT = SIMD ⊕ SMT



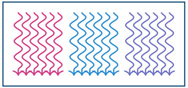

*Vector*




*SMT*




*SIMT*



Graphics: img:amperepictures

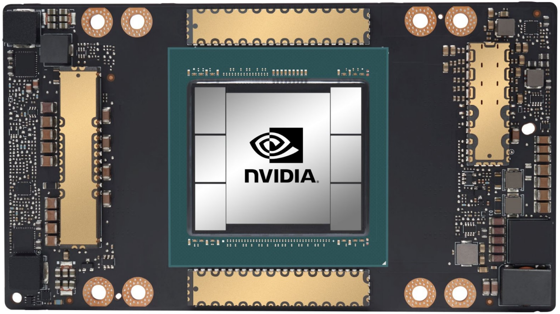


JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

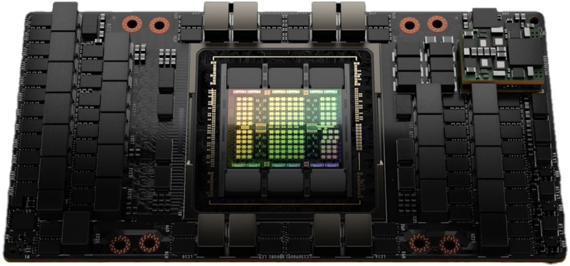
# A100 vs H100

## Comparison of current vs. next generation

A100




H100



# A100 vs H100

## Comparison of current vs. next generation

### A100


### H100




JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

# A100 vs H100

## Comparison of current vs. next generation

### A100


### H100




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

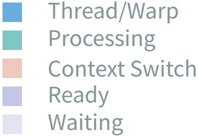
# Low Latency vs. High Throughput

## Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency

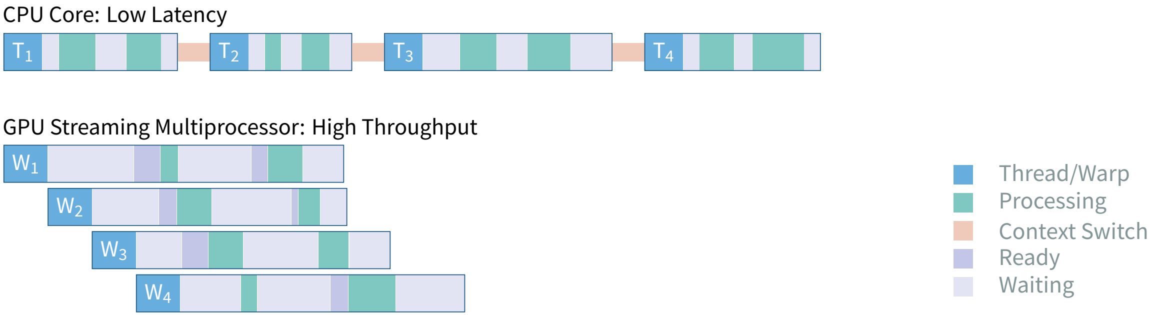



# Low Latency vs. High Throughput

## Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

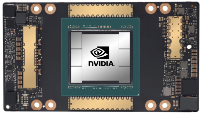


# CPU vs. GPU

## 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

Application

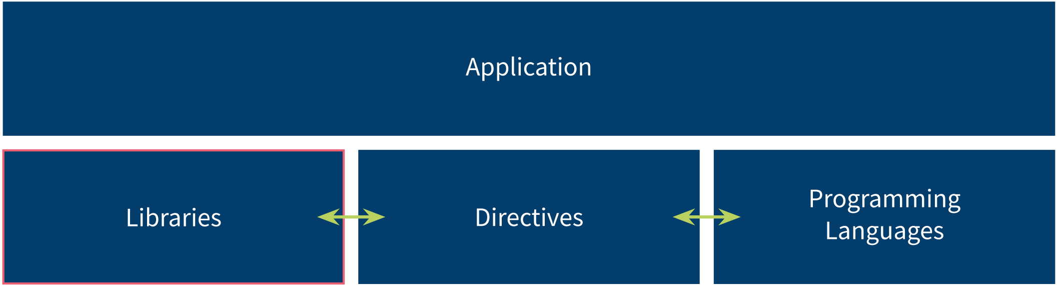
Libraries ↔ Directives ↔ Programming Languages

*Drop-in* Acceleration

*Easy* Acceleration

*Flexible* Acceleration

# Summary of Acceleration Possibilities




*Drop-in* Acceleration

*Easy* Acceleration

*Flexible* Acceleration

# 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!

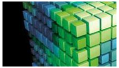
***Use applications & libraries***

Wizard: Breazell [10]

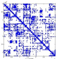
# Libraries

Programming GPUs is easy: Just don't!

***Use applications & libraries***



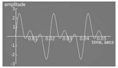
cuBLAS



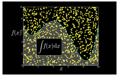
cuSPARSE

cuDNN

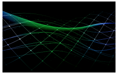
OpenCV




cuFFT



cuRAND



CUDA Math

Thrust

ArrayFire

Numba

CuPy

Wizard: Breazell [10]



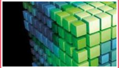
JÜLICH Forschungszentrum

JÜLICH SUPERCOMPUTING CENTRE

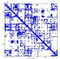
# Libraries

Programming GPUs is easy: Just don't!

***Use applications & libraries***



cuBLAS



cuSPARSE

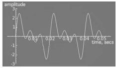
cuDNN

OpenCV

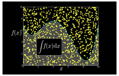
Thrust

ArrayFire

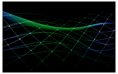
Numba



cuFFT



cuRAND



CUDA Math

CuPy

Wizard: Breazell [10]

# Summary of Acceleration Possibilities

Application

Libraries

Directives

Programming Languages

*Drop-in* Acceleration

*Easy* Acceleration

*Flexible* Acceleration

# Summary of Acceleration Possibilities

Application

Libraries ↔ Directives ↔ Programming Languages

*Drop-in* Acceleration

*Easy* Acceleration

*Flexible* Acceleration

# Parallelism

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 $t = t_{\textbf{s}\text{erial}} + t_{\textbf{p}\text{arallel}}$

# Primer on Parallel Scaling

## Amdahl's Law

Possible maximum speedup for $N$ parallel processors

Total Time $t = t_{\textbf{s}\text{erial}} + t_{\textbf{p}\text{arallel}}$

$N$ Processors $t(N) = t_\text{s} + t_\text{p}/N$



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

# Primer on Parallel Scaling

## Amdahl's Law

Possible maximum speedup for $N$ parallel processors

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

$N$ 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

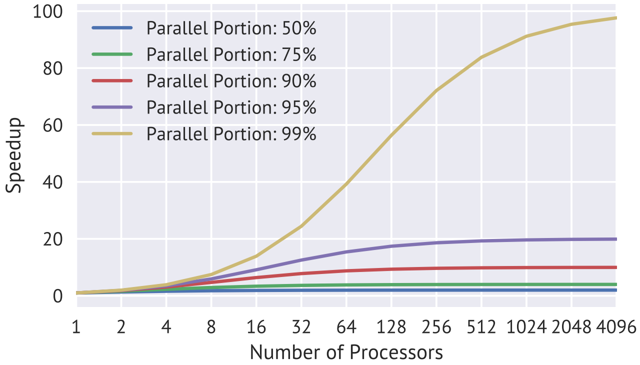
## Amdahl's Law

Possible maximum speedup for $N$ parallel processors

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

$N$ 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, OpenMP
- Thrust
- Kokkos, RAJA, ALPAKA, SYCL, DPC++, pSTL
- PyCUDA, Cupy, Numba

Other alternatives

- CUDA Fortran
- HIP
- OpenCL

# Programming GPUs

## CUDA C/C++

# 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);
```

# 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 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();
```

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’s Parallel Model

## In software: Threads, Blocks

- Methods to exploit parallelism:

# CUDA’s Parallel Model

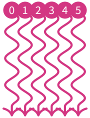
## In software: Threads, Blocks

- Methods to exploit parallelism:
  - Thread

# CUDA’s Parallel Model

## In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads



# CUDA's Parallel Model

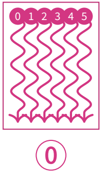
## In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$

# CUDA's Parallel Model

## In software: Threads, Blocks

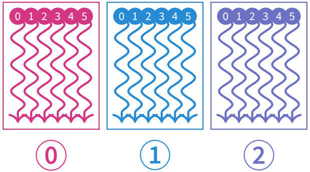
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$
  - $\boxed{\text{Block}}$



# CUDA's Parallel Model

## In software: Threads, Blocks

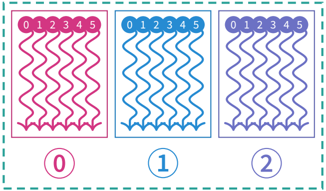
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$
  - $\boxed{\text{Blocks}}$



# CUDA's Parallel Model

## In software: Threads, Blocks

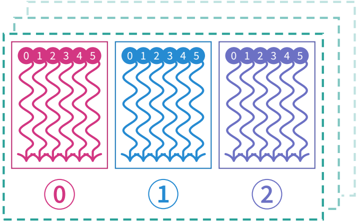
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$
  - $\boxed{\text{Blocks}}$ → Grid



# CUDA’s Parallel Model

## In software: Threads, Blocks

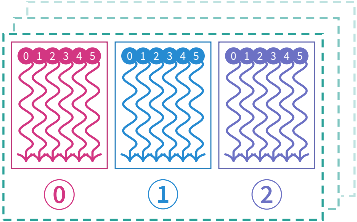
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



# CUDA’s Parallel Model

## In software: Threads, Blocks

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




- 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

Application

| Libraries | Directives | Programming Languages |
|---|---|---|
| *Drop-in* Acceleration | *Easy* Acceleration | *Flexible* Acceleration |

# Summary of Acceleration Possibilities

Application

Libraries

This Course

Programming Languages

*Drop-in* Acceleration

*Easy* Acceleration

*Flexible* Acceleration

# Conclusions

# Conclusions

- GPUs achieve performance by specialized hardware $\rightarrow$ **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 $\rightarrow$ **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*

# Appendix

Appendix

Glossary

References

# Glossary I

**AMD** Manufacturer of CPUs and GPUs. 3, 4, 5, 6, 7, 8, 9

**Ampere** GPU architecture from NVIDIA (announced 2019). 13, 14, 15

**API** A programmatic interface to software by well-defined functions. Short for application programming interface. 107

**ATI** Canada-based GPUs manufacturing company; bought by AMD in 2006. 3, 4, 5, 6, 7, 8, 9

**CUDA** Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 3, 4, 5, 6, 7, 8, 9, 84, 85, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 101, 102, 103, 107

**JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 107

# Glossary II

**JURECA** A multi-purpose supercomputer at JSC. 15

**JUWELS** Jülich's new supercomputer, the successor of JUQUEEN. 12, 13, 14

**NVIDIA** US technology company creating GPUs. 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 58, 59, 60, 106, 108

**OpenACC** Directive-based programming, primarily for many-core machines. 1, 84, 89, 90, 91, 92, 93, 94, 95, 96, 97

**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, 8, 9, 84

**OpenGL** The *Open Graphics Library*, an API for rendering graphics across different hardware architectures. 3, 4, 5, 6, 7, 8, 9

**OpenMP** Directive-based programming, primarily for multi-threaded machines. 84

# Glossary III

**SAXPY** Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 86, 87, 88

**Tesla** The GPU product line for general purpose computing computing of NVIDIA. 12

**Thrust** A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 84

**CPU** Central Processing Unit. 12, 15, 20, 21, 22, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 86, 106, 107

**GPU** Graphics Processing Unit. 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 41, 42, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 64, 65, 66, 68, 71, 72, 73, 74, 75, 78, 85, 88, 101, 102, 103, 106, 107, 108

# Glossary IV

SIMD Single Instruction, Multiple Data. 51, 52, 53, 54, 55, 56, 57, 58, 59, 60

SIMT Single Instruction, Multiple Threads. 23, 24, 25, 38, 39, 41, 42, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60

SM Streaming Multiprocessor. 51, 52, 53, 54, 55, 56, 57, 58, 59, 60

SMT Simultaneous Multithreading. 51, 52, 53, 54, 55, 56, 57, 58, 59, 60

# 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 3–9).

[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 3–9).

[4] Jack Dongarra et al. *TOP500*. June 2019. URL: https://www.top500.org/lists/2019/06/ (pages 3–9).

# References II


[5] Jack Dongarra et al. *Green500*. June 2019. URL: https://www.top500.org/green500/lists/2019/06/ (pages 3–9).

[6] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (pages 10, 11).

[10] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 71–75).

# References: Images, Graphics I


[1] Héctor J. Rivas. *Color Reels*. Freely available at Unsplash. URL: https://unsplash.com/photos/87hFrPk3V-s.

[7] Forschungszentrum Jülich GmbH (Ralf-Uwe Limbach). *JUWELS Booster*.

[8] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (pages 20, 21).

[9] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (pages 20, 21).