



# GPU Introduction

## JSC OpenACC Course 2022

25 October 2022 | Andreas Herten | Forschungszentrum Jülich

# Outline

## Introduction

- GPU History

- GPU History

- Architecture Comparison

- Jülich Systems

  - JUWELS Cluster

  - JUWELS Booster

  - JURECA DC

- App Showcase

## Platform

- 3 Core Features

  - Memory

  - Asynchronicity

  - SIMT

  - Generation Comparison

- High Throughput

- Summary

- Programming GPUs

  - Libraries

  - GPU programming models

  - CUDA C/C++

  - Parallel Model

- Conclusions

# 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]
- 2022 : **Leonardo** (250 PFLOP/s\*, Italy), **NVIDIA** GPUs; **LUMI** (552 PFLOP/s, Finland), **AMD** GPUs  
: **Frontier** ( $R_{\max} = 1.102$  EFLOP/s, ORNL), **AMD** GPUs

\*: Effective FLOP/s, not theoretical peak

# History of GPUs

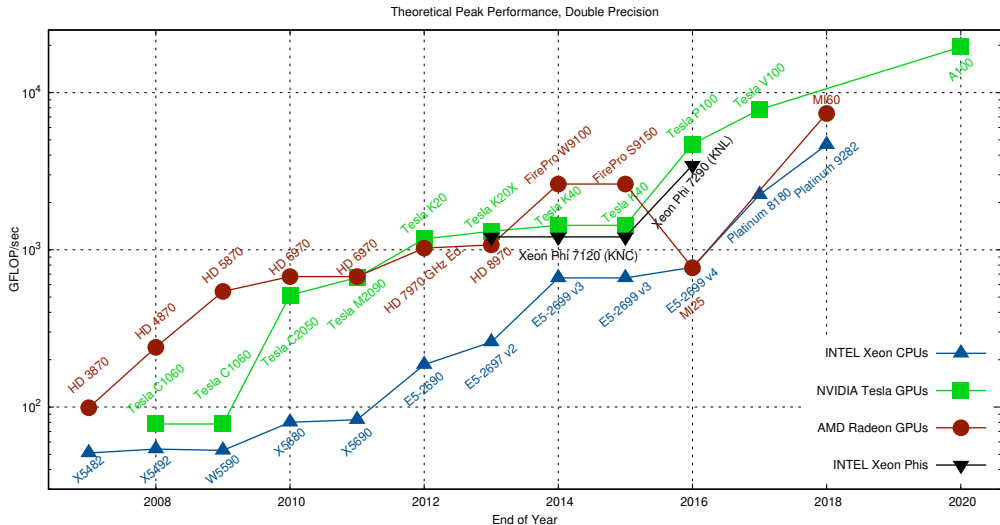
## A short but unparalleled story

- 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]
- 2022 : **Leonardo** (250 PFLOP/s\*, Italy), **NVIDIA** GPUs; **LUMI** (552 PFLOP/s, Finland), **AMD** GPUs  
: **Frontier** ( $R_{\max} = 1.102$  EFLOP/s, ORNL), **AMD** GPUs
- Future* : JUPITER ( $\approx 1$  EFLOP/s, **JSC**)  
: **Aurora** ( $\approx 2$  EFLOP/s, Argonne), **Intel** GPUs; **El Capitan** ( $\approx 2$  EFLOP/s, LLNL), **AMD** GPUs

\*: Effective FLOP/s, not theoretical peak

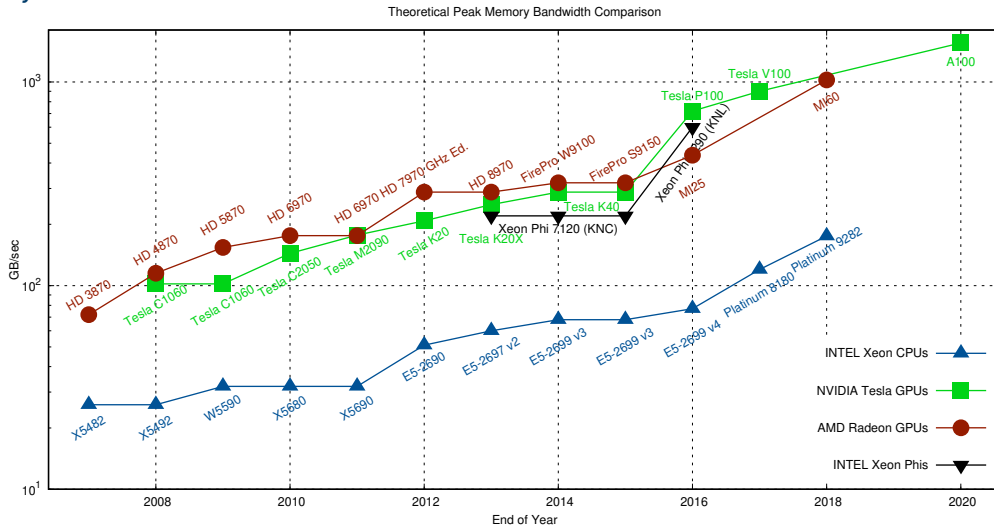
# Status Quo Across Architectures

## Performance



# Status Quo Across Architectures

## Memory Bandwidth





## 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 ( $2 \times 24$  cores)
- Each with 4 NVIDIA A100 Ampere GPUs (each:  $\text{FP64TC: } 19.5$  TFLOP/s, 40 GB memory  
 $\text{FP64: } 9.7$  TFLOP/s)
- InfiniBand DragonFly+ HDR-200 network;  $4 \times 200$  Gbit/s per node



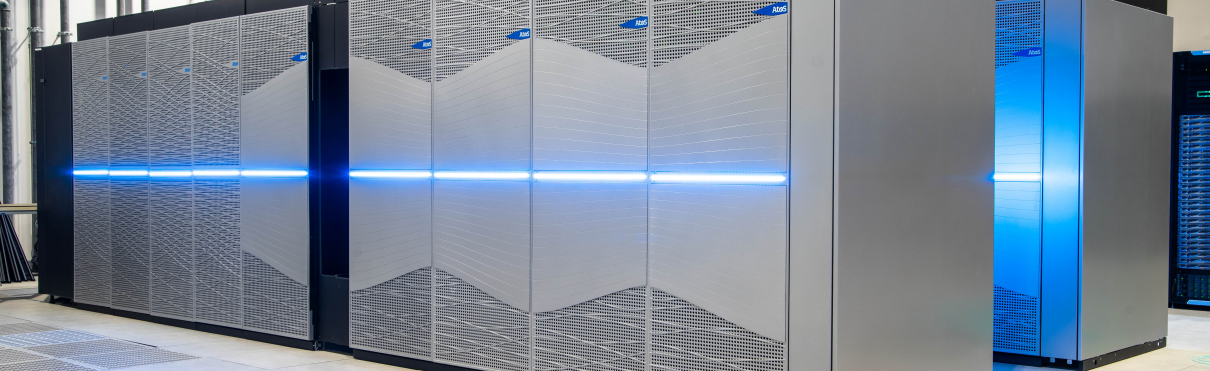
### Top500 List Nov 2021:

- #1 Europe
- #8 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:  $\text{FP64TC: } 19.5$  TFLOP/s, 40 GB memory)  
 $\text{FP64: } 9.7$
- InfiniBand DragonFly+ HDR-200 network;  $4 \times 200$  Gbit/s per node



## JURECA DC – Multi-Purpose

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

# 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!*

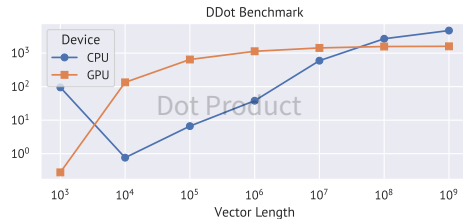
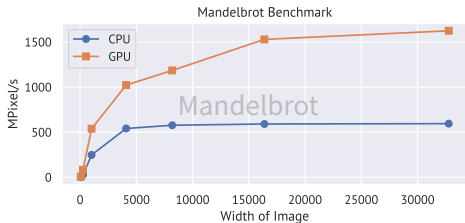
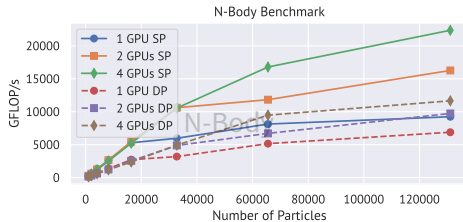
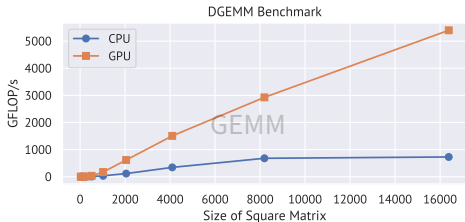
Mandelbrot

Dot Product

# Getting GPU-Acquainted

## Some Applications

TASK



# Platform

# CPU vs. GPU

A matter of specialties





# CPU vs. GPU

A matter of specialties



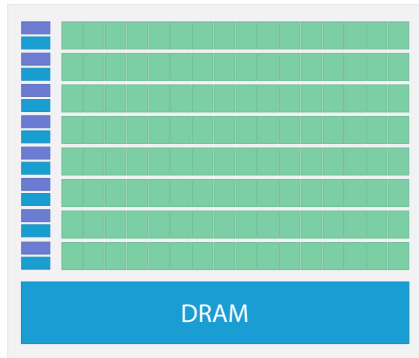
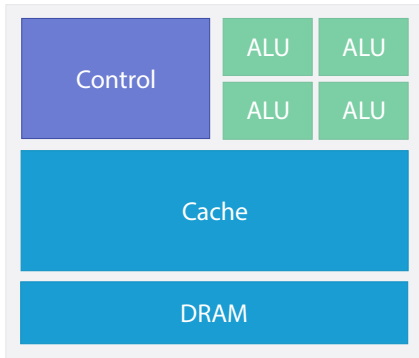
Transporting one



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

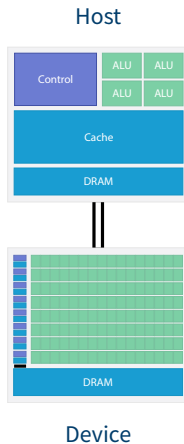
Asynchronicity

Memory

# Memory

GPU memory ain't no CPU memory

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

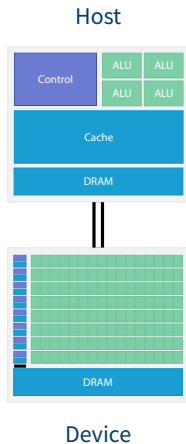


# Memory

GPU memory ain't no CPU memory

Unified Virtual Addressing

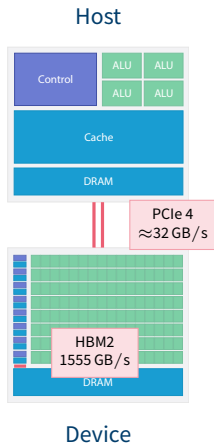
- GPU: accelerator / extension card
- Separate device from CPU
- Separate memory, but UVA**



# Memory

GPU memory ain't no CPU memory

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

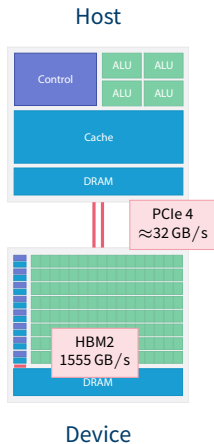




# Memory

## 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!*



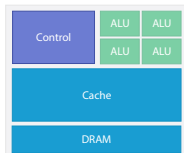
# Memory

## GPU memory ain't no CPU memory

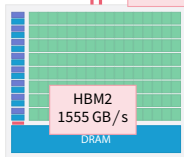
Unified Memory

- GPU: accelerator / extension card
- Separate device from CPU
- **Separate memory, but UVA and UM**
- Memory transfers need special consideration!  
*Do as little as possible!*
- Choice: automatic transfers (convenience) or manual transfers (control)

Host



PCIe 4  
≈32 GB/s

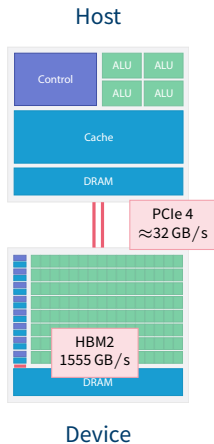


Device

# 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!*
- 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)

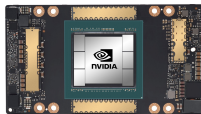
### V100

32 GB RAM, 900 GB/s

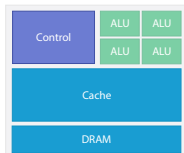


### A100

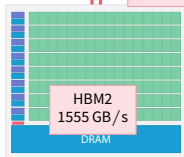
40 GB RAM, 1555 GB/s



### Host



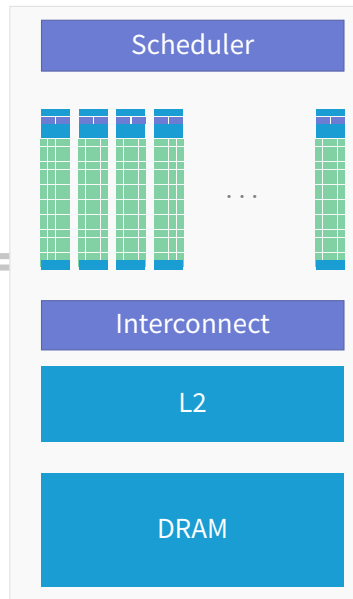
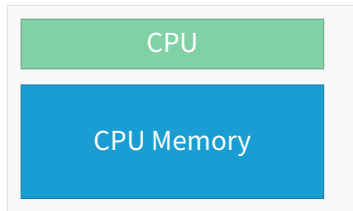
PCIe 4  
≈32 GB/s



### Device

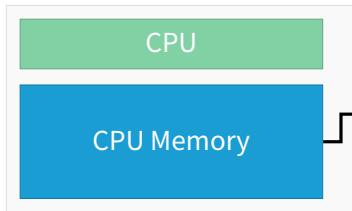
# Processing Flow

CPU → GPU → CPU

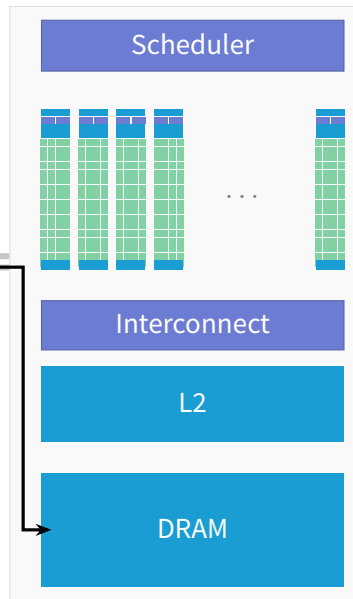


# Processing Flow

CPU → GPU → CPU

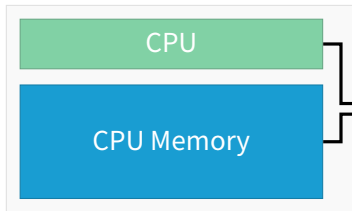


- 1 Transfer data from CPU memory to GPU memory

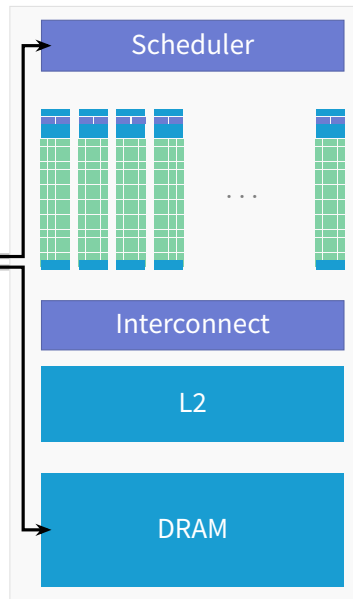


# Processing Flow

CPU → GPU → CPU

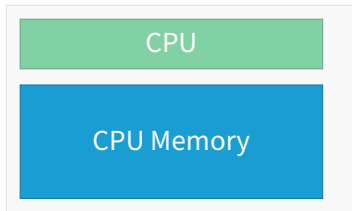


- 1 Transfer data from CPU memory to GPU memory, transfer program

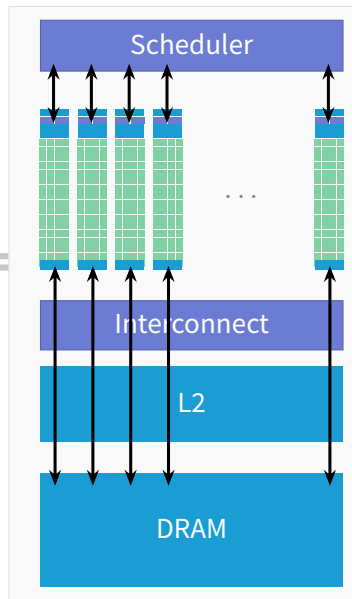


# Processing Flow

CPU → GPU → CPU



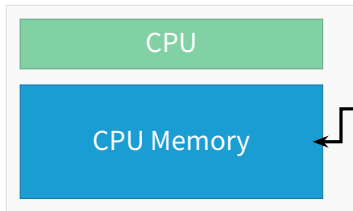
- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back



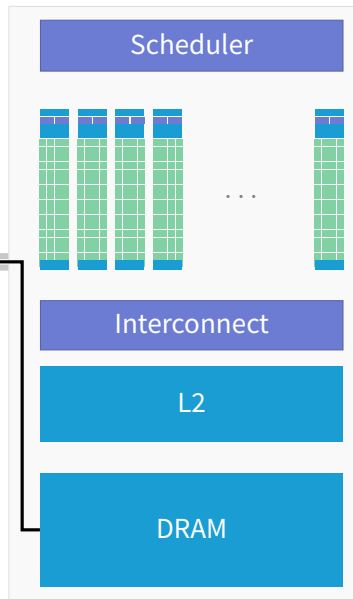


# Processing Flow

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*

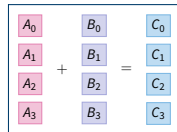
$A_0$	+	$B_0$	=	$C_0$
$A_1$	+	$B_1$	=	$C_1$
$A_2$	+	$B_2$	=	$C_2$
$A_3$	+	$B_3$	=	$C_3$

# SIMT

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

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

*Vector*



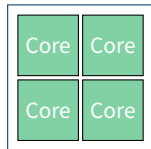
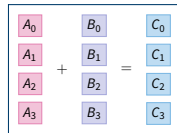


# SIMT

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

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

*Vector*

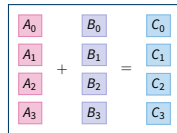


# SIMT

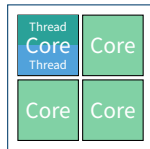
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

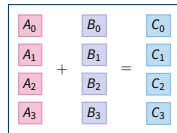


# SIMT

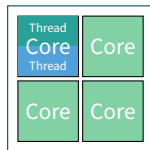
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

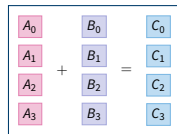


# SIMT

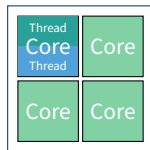
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

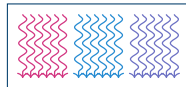
Vector



SMT




SIMT

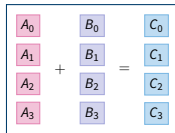


# SIMT

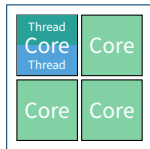
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

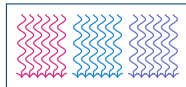
Vector



SMT

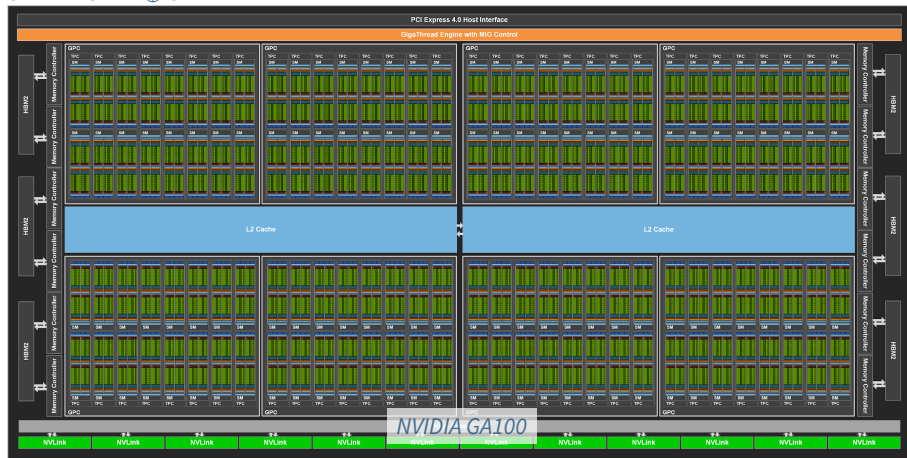


SIMT



# SIMT

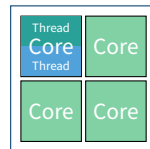
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



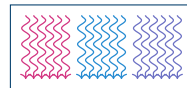
Vector

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

SMT



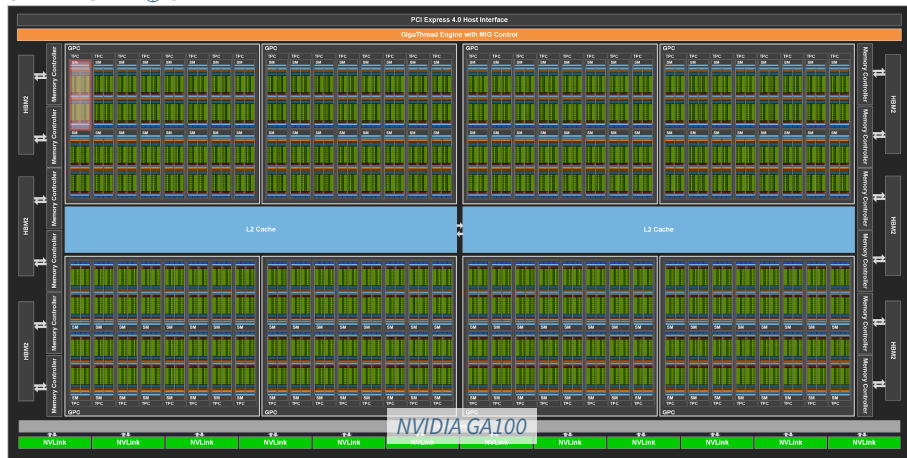
SIMT



Graphics: img:amperepictures

# SIMT

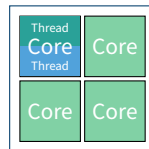
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



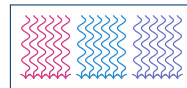
Vector

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

SMT



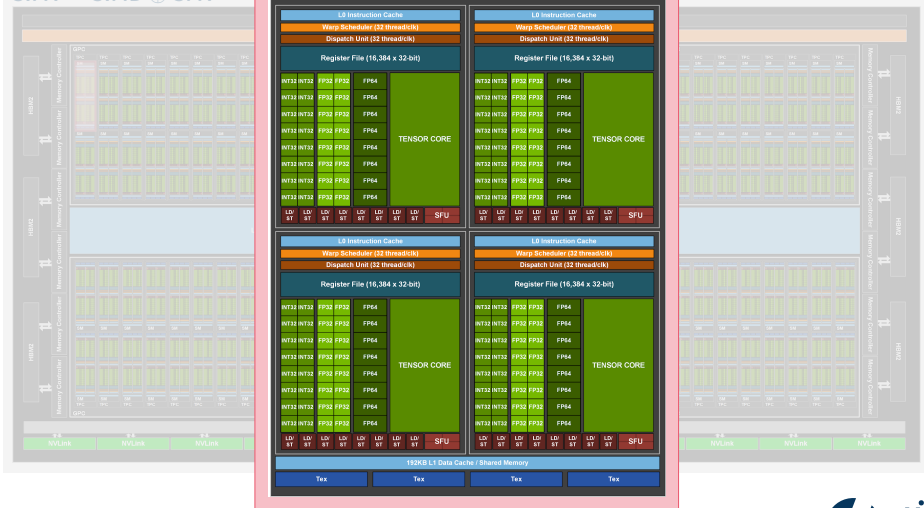
SIMT



Graphics: img:amperepictures

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

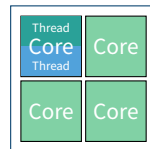
## Multiprocessor



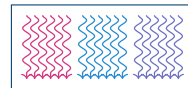
## Vector

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

*SMT*



SIMT



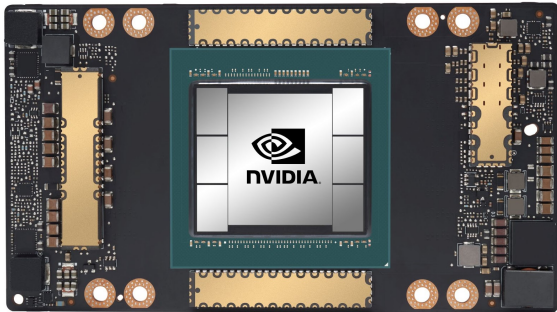
Graphics: [img:amperepictures](#)



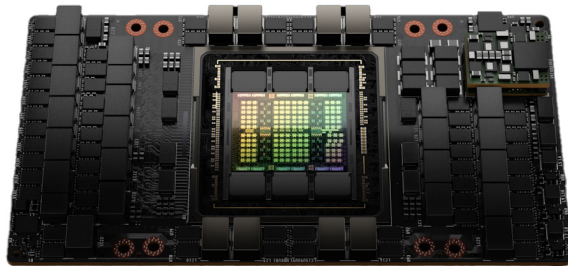
# A100 vs H100

Comparison of current vs. next generation

A100



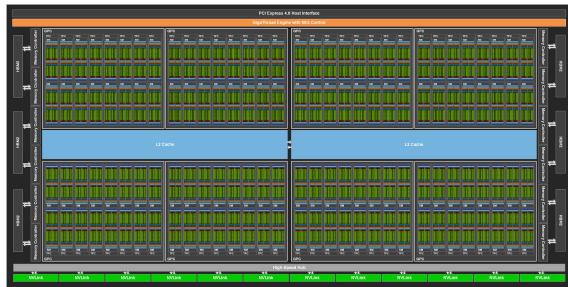
H100



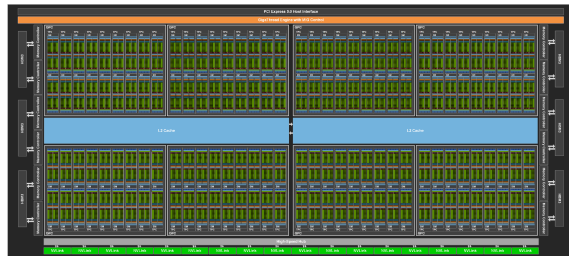
# A100 vs H100

Comparison of current vs. next generation

## A100



## H100



# A100 vs H100

## Comparison of current vs. next generation

### A100



### H100



# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

**CPU** Minimizes latency within each thread

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

# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

**CPU** Minimizes latency within each thread

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

CPU Core: Low Latency



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

# Low Latency vs. High Throughput

Maybe GPU's ultimate feature

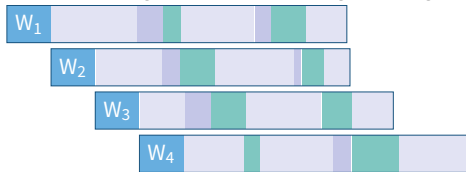
**CPU** Minimizes latency within each thread

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

CPU Core: Low Latency



GPU Streaming Multiprocessor: High Throughput



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

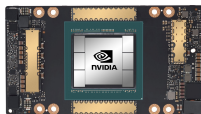
# CPU vs. GPU

Let's summarize this!



Optimized for **low latency**

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



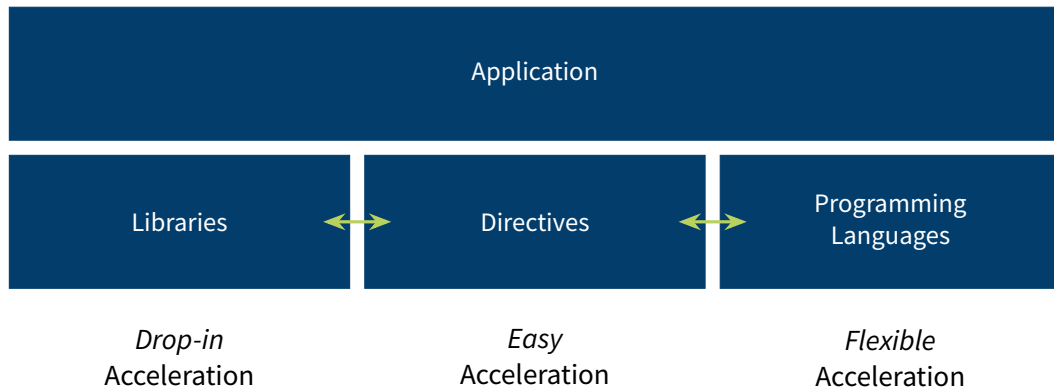
Optimized for **high throughput**

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

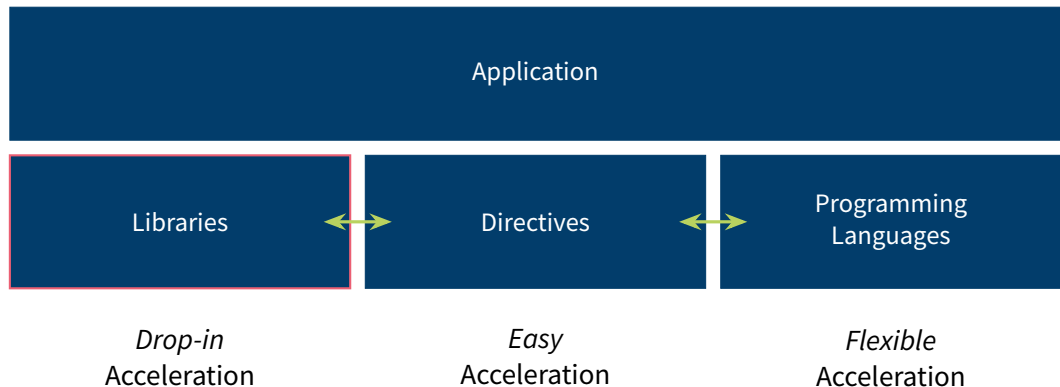
# Programming GPUs



# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



# Libraries

Programming GPUs is easy: Just don't!

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



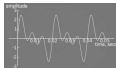
cuBLAS



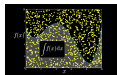
cuSPARSE



cuDNN



cuFFT



cuRAND



CUDA Math



{A} ARRAYFIRE

Numba

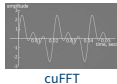


CuPy

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*

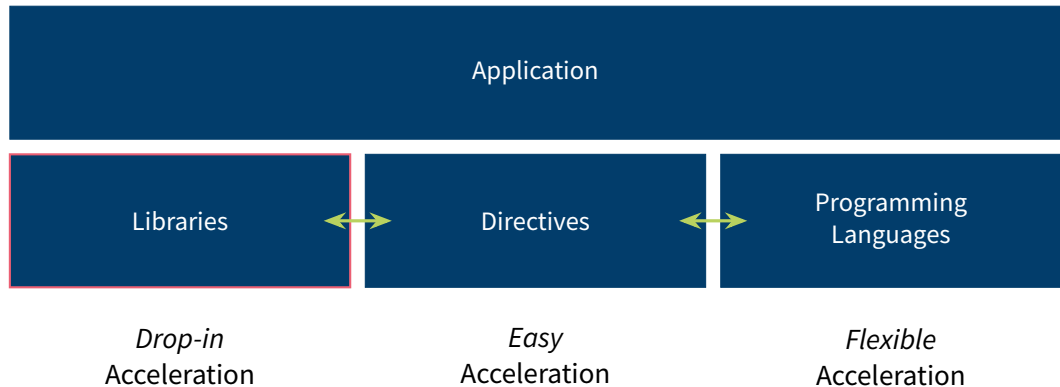


Numba



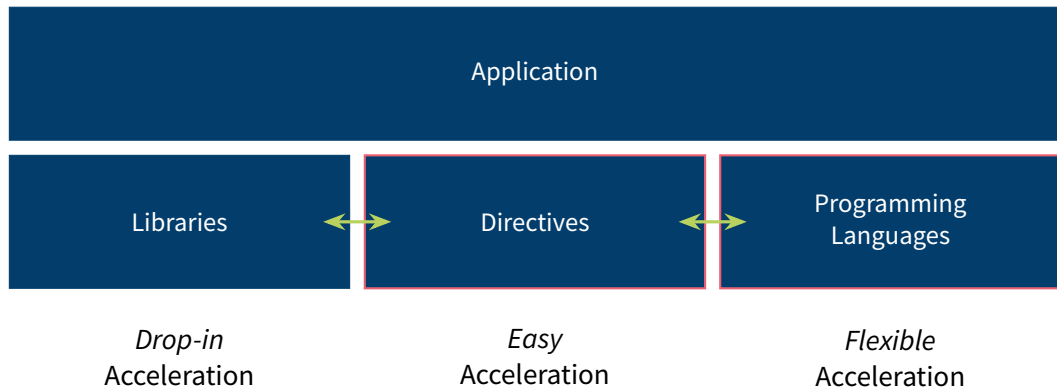
CuPy

# Summary of Acceleration Possibilities





# Summary of Acceleration Possibilities



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

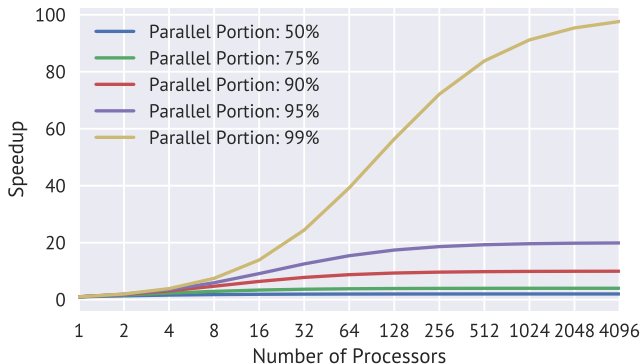
## Amdahl's Law

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, 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];  
}
```

Specify kernel

ID variables

Guard against  
too many threads

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

```
// fill x, y
```

```
cudaMallocManaged(&x, n * sizeof(float));
```

```
cudaMallocManaged(&y, n * sizeof(float));
```

Allocate GPU-capable  
memory

Call kernel  
2 blocks, each 5 threads

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

Wait for  
kernel to finish

```
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



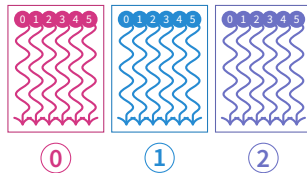
# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks



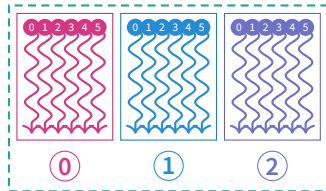
# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid



# CUDA's Parallel Model

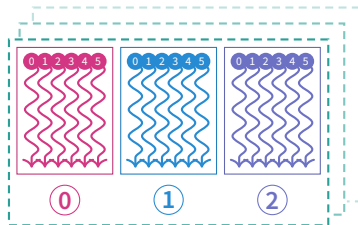
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



# CUDA's Parallel Model

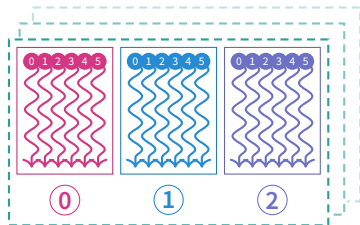
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



- Execution entity: **threads**

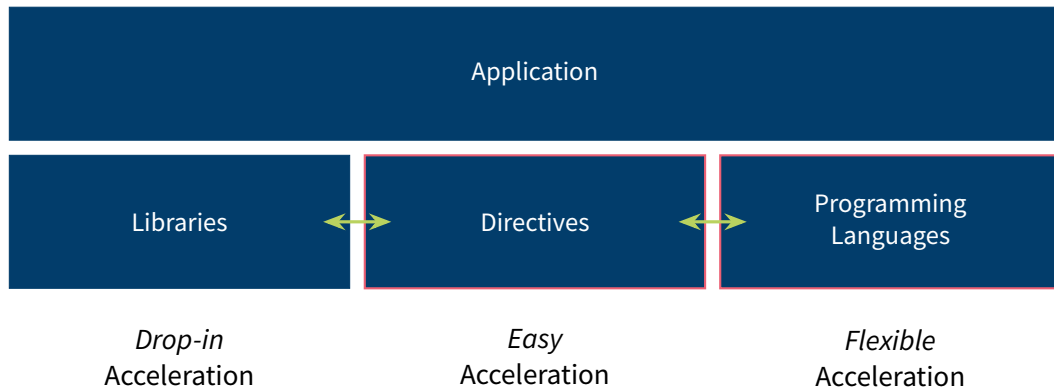
- Lightweight → fast switching!

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

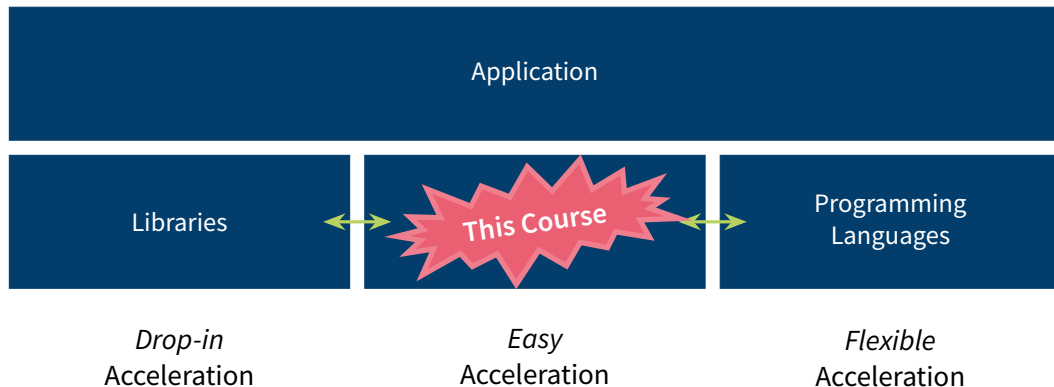
- **OpenACC** takes care of threads and blocks for you!

- Block configuration is just an optimization!

# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



# Conclusions



# Conclusions

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

# Conclusions

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

# Conclusions

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

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

# Appendix

# 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. 99

**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, 76, 77, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 93, 94, 95, 99

**JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 99

# 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](#), [50](#), [51](#), [52](#), [98](#), [100](#)

**OpenACC** Directive-based programming, primarily for many-core machines. [1](#), [76](#), [81](#), [82](#), [83](#), [84](#), [85](#), [86](#), [87](#), [88](#), [89](#)

**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](#), [76](#)

**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. [76](#)

# Glossary III

**SAXPY** Single-precision  $A \times X + Y$ . A simple code example of scaling a vector and adding an offset. 78, 79, 80

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

**Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 76

**CPU** Central Processing Unit. 12, 15, 20, 21, 22, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 78, 98, 99

**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, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 56, 57, 58, 60, 63, 64, 65, 66, 67, 70, 77, 80, 93, 94, 95, 98, 99, 100



# Glossary IV

**SIMD** Single Instruction, Multiple Data. [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#), [52](#)

**SIMT** Single Instruction, Multiple Threads. [23](#), [24](#), [25](#), [38](#), [39](#), [41](#), [42](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#), [52](#)

**SM** Streaming Multiprocessor. [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#), [52](#)

**SMT** Simultaneous Multithreading. [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [51](#), [52](#)

# 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 63–67).

# 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).