GPU Introduction

JSC OpenACC Course 2017

Andreas Herten, Forschungszentrum Jülich, 16 October 2017

Outline

Member of the Helmholtz Association

History of GPUs

A short but parallel story

1999 Graphics computation pipeline implemented in dedicated *graphics hardware*
Computations using OpenGL graphics library [1]
»GPU« coined by NVIDIA [2]

History of GPUs

A short but parallel story

1999 Graphics computation pipeline implemented in dedicated *graphics hardware*
Computations using OpenGL graphics library [1]
»GPU« coined by NVIDIA [2]

2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

History of GPUs

A short but parallel story

1999 Graphics computation pipeline implemented in dedicated *graphics hardware*
Computations using OpenGL graphics library [1]
»GPU« coined by NVIDIA [2]

2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

2007 CUDA

History of GPUs

A short but parallel story

1999 Graphics computation pipeline implemented in dedicated *graphics hardware*
Computations using OpenGL graphics library [1]
»GPU« coined by NVIDIA [2]

2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

2007 CUDA

2009 OpenCL

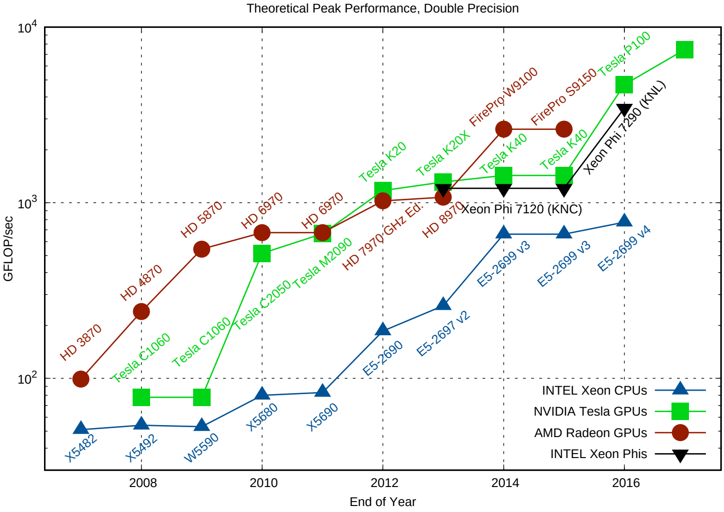
History of GPUs

A short but parallel story

1999	Graphics computation pipeline implemented in dedicated *graphics hardware* Computations using OpenGL graphics library [1] »GPU« coined by NVIDIA [2]
2001	NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
2007	CUDA
2009	OpenCL
2017	Top 500: 15 % with GPUs [3], Green 500: 9 of 10 of top 10 with GPUs [4]

Status Quo Across Architectures

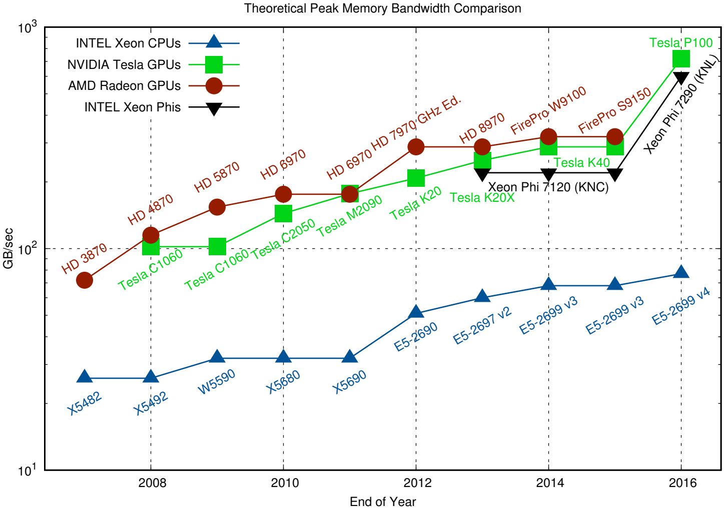
Performance



Graphic: Rupp [5]

Status Quo Across Architectures

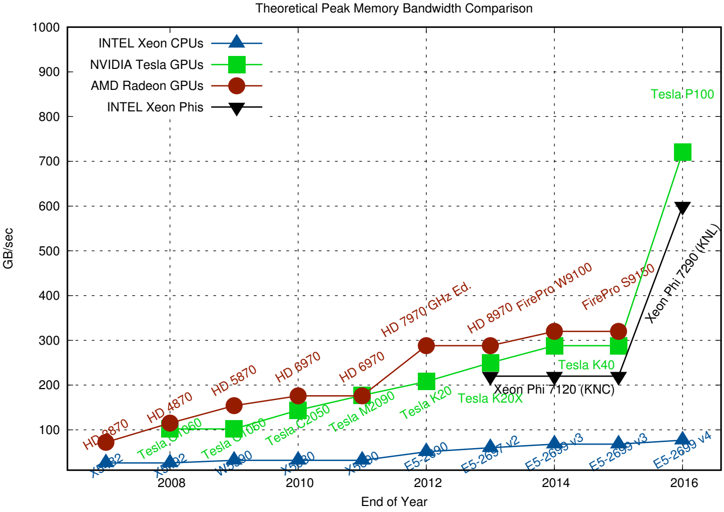
Memory Bandwidth



Graphic: Rupp [5]

Status Quo Across Architectures

Memory Bandwidth



Graphic: Rupp [5]


JÜLICH
JURECA

JURON – A Human Brain Project *Prototype*

- 18 nodes with IBM POWER8*NVL* CPUs (2×10 cores)
- Per Node: 4 NVIDIA Tesla P100 cards, connected via NVLink
- GPU: 0.38 PFLOP/s peak performance
- Dedicated visualization nodes

Getting GPU-Acquainted

Some Applications

TASK

Location of Code:

`Introduction-G…/Tasks/getting_started/`

See `Instructions.md` for hints.

Getting GPU-Acquainted

Some Applications

TASK

Dot Product

GEMM

Location of Code:

`Introduction-G…/Tasks/getting_started/`

See `Instructions.md` for hints.

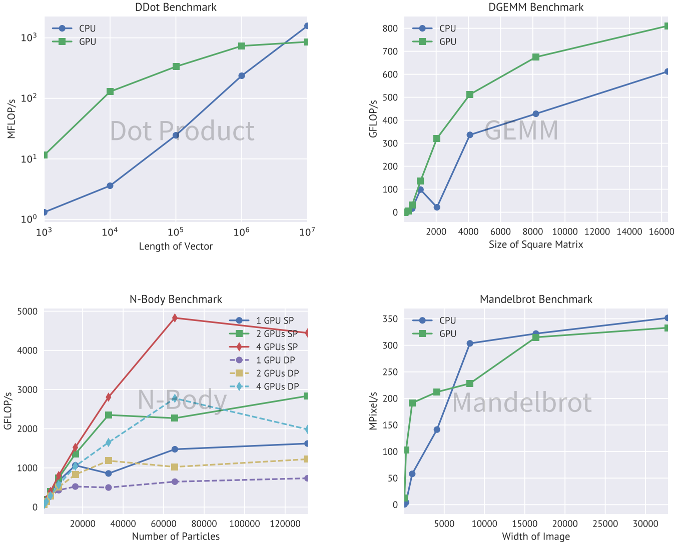
N-Body

Mandelbrot

Getting GPU-Acquainted

Some Applications

TASK



The GPU Platform

CPU vs. GPU

A matter of specialties

Graphics: Lee [6] and Shearings Holidays [7]

CPU vs. GPU

A matter of specialties

Transporting one

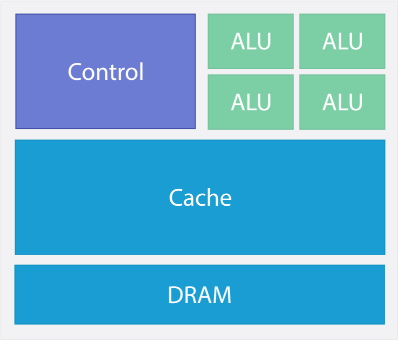
Transporting many

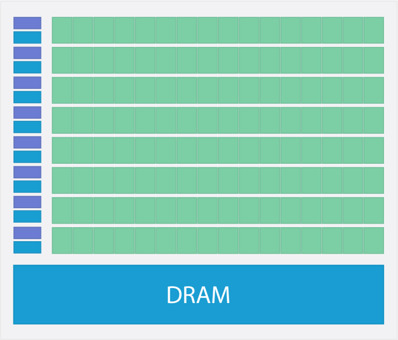
Graphics: Lee [6] and Shearings Holidays [7]

Member of the Helmholtz Association

CPU vs. GPU

Chip





Aim: Hide Latency

Everything else follows

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

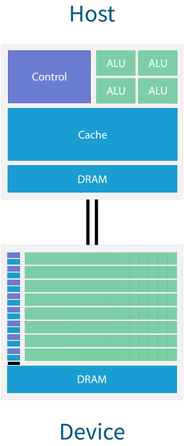
Memory

Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU



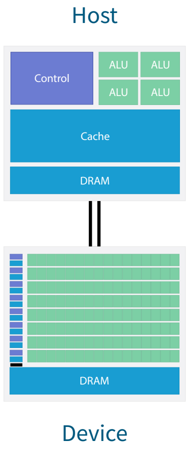
Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA



Memory

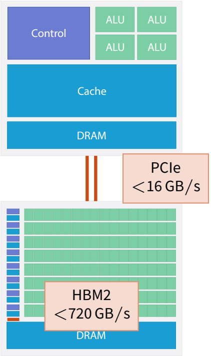
GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA

Host



Device

Memory

GPU memory ain't no CPU memory

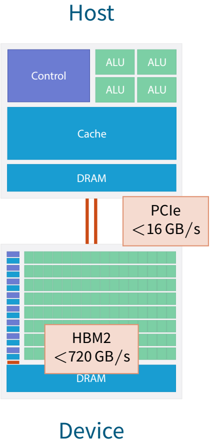
- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA

- Memory transfers need special consideration!

  Do as little as possible!



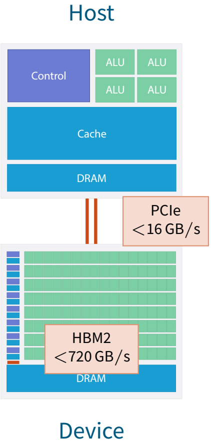
Memory

GPU memory ain't no CPU memory

- 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: Can be done automatically



Memory

GPU memory ain't no CPU memory

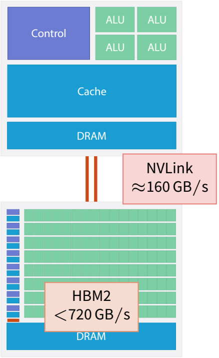
- GPU: accelerator / extension card

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

- Memory transfers need special consideration!
Do as little as possible!

- Formerly: Explicitly copy data to/from GPU
Now: Can be done automatically

Host



Device

Memory

GPU memory ain't no CPU memory

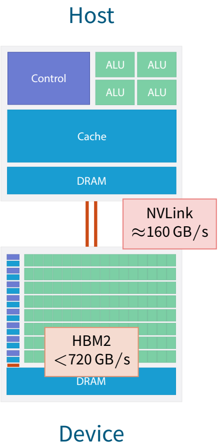
- GPU: accelerator / extension card

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

- Memory transfers need special consideration!
Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
Now: Can be done automatically
- Example values

P100

16 GB RAM, 720 GB/s



Memory

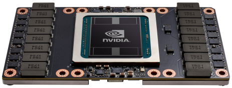
GPU memory ain't no CPU memory

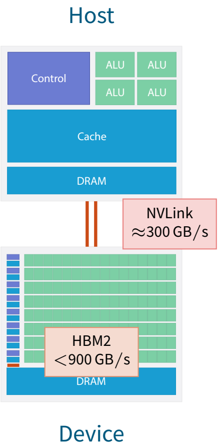
- GPU: accelerator / extension card

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

- Memory transfers need special consideration!
Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
Now: Can be done automatically
- Example values

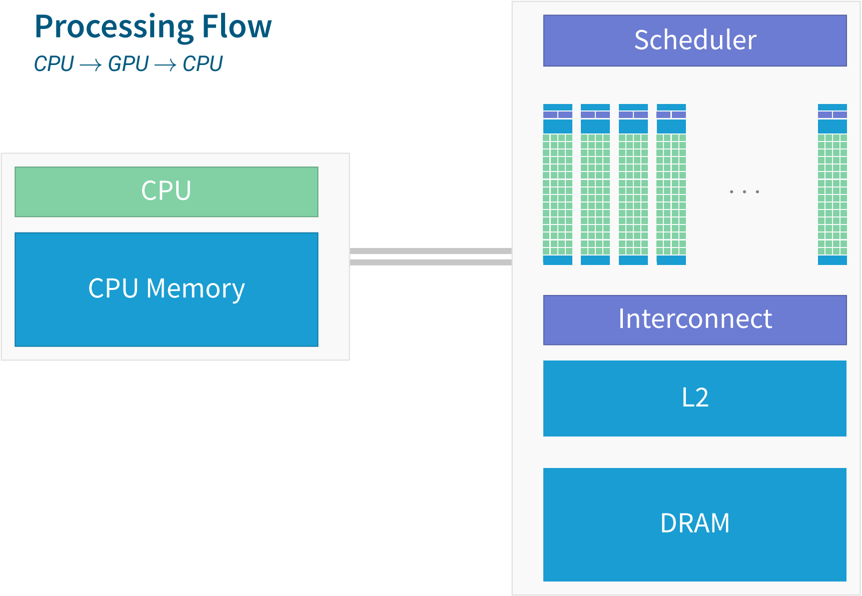
P100	**V100**
16 GB RAM, 720 GB/s	16 GB RAM, 900 GB/s





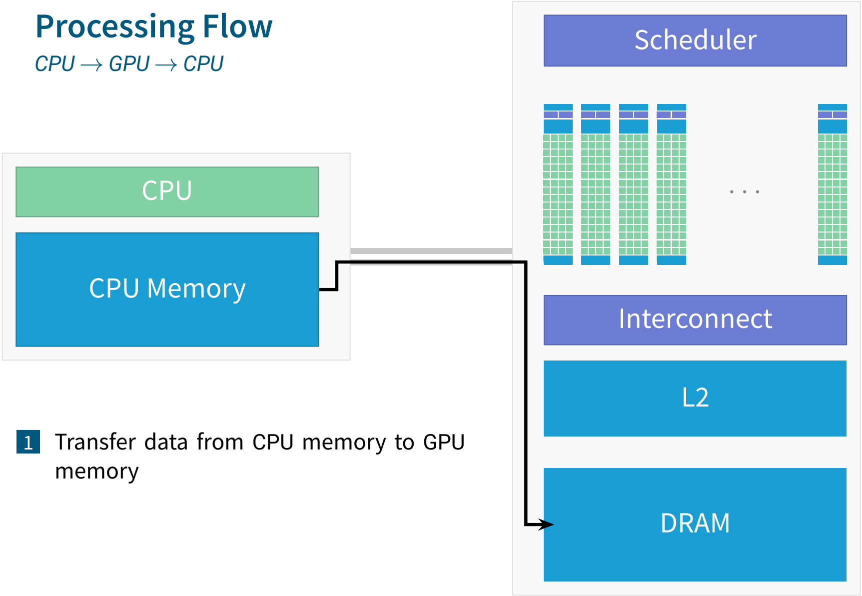
Processing Flow

CPU → GPU → CPU



Processing Flow

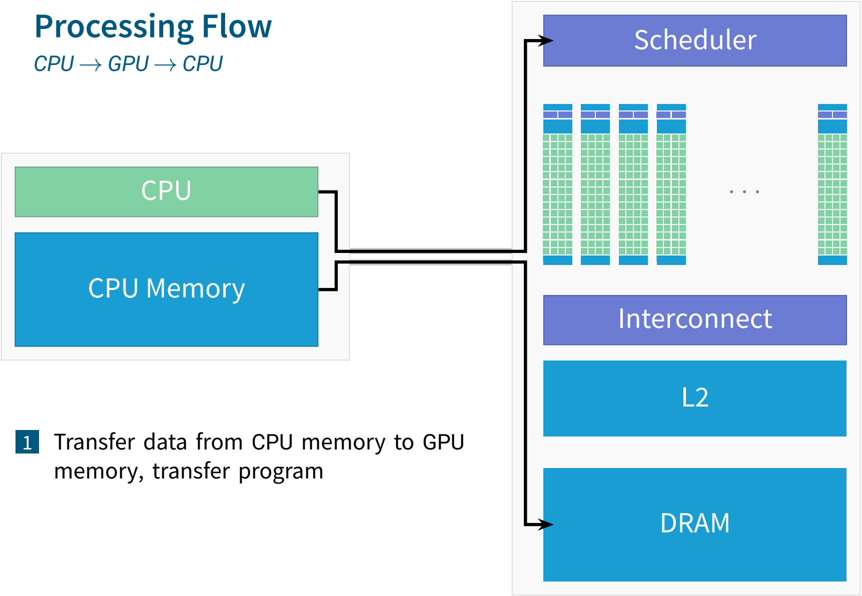
CPU → GPU → CPU



1 Transfer data from CPU memory to GPU memory

Processing Flow

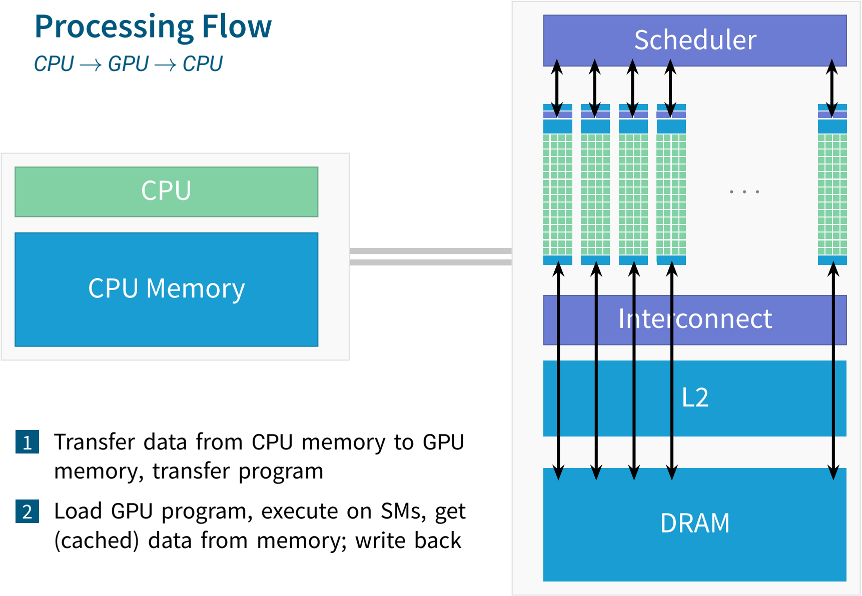
CPU → GPU → CPU



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

Processing Flow

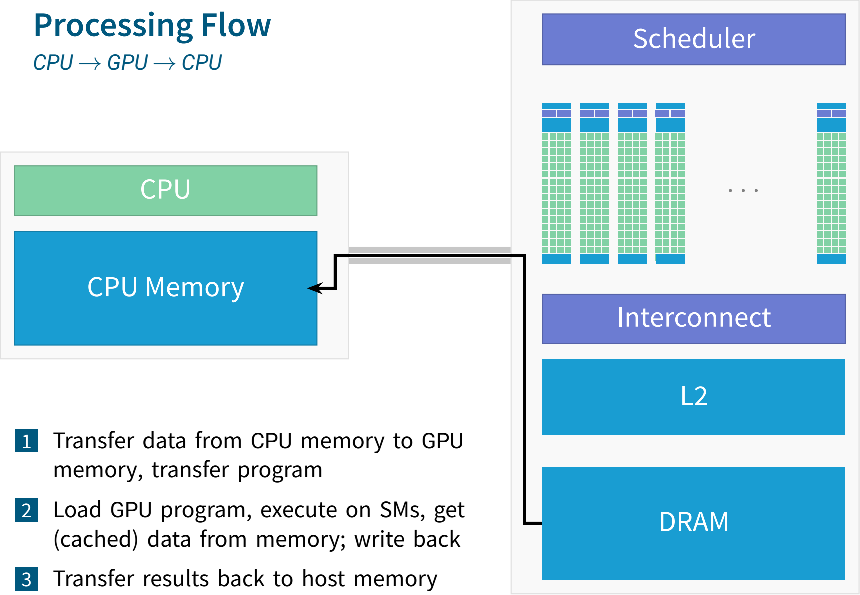
CPU → GPU → CPU



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

Processing Flow

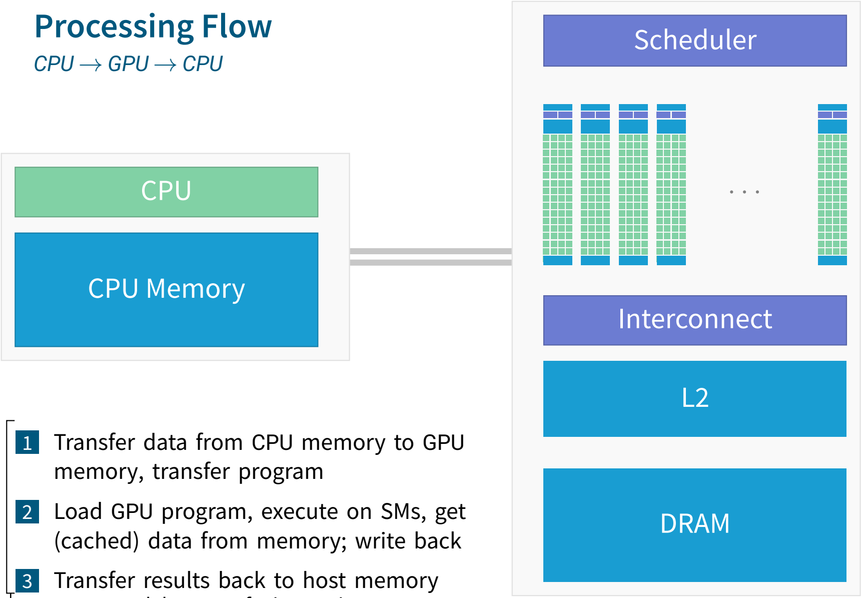
CPU → GPU → CPU



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

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

- **UVA**: Manual data transfer invocations
- **UM**: Driver automatically transfers data

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

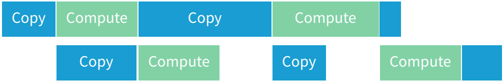
Async

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

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

SIMT

Of threads and warps

JÜLICH
FORSCHUNGSZENTRUM

Scalar

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

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

SIMT

Of threads and warps

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

Vector

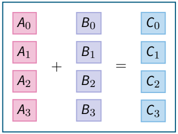
$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} + \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

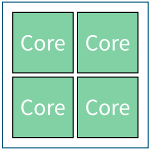
SIMT

Of threads and warps

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

Vector





SIMT

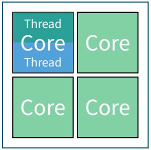
Of threads and warps

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

Vector

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

SMT

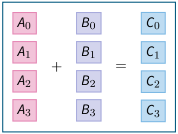


SIMT

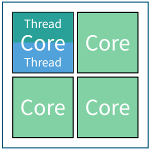
Of threads and warps

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

Vector



SMT

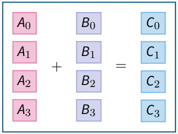


SIMT

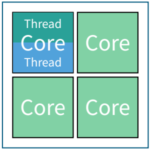
Of threads and warps

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

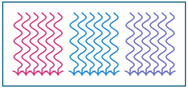
Vector



SMT



SIMT

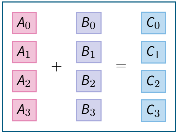


SIMT

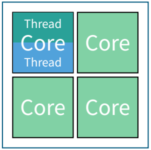
Of threads and warps

- 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 

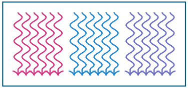
Vector



SMT

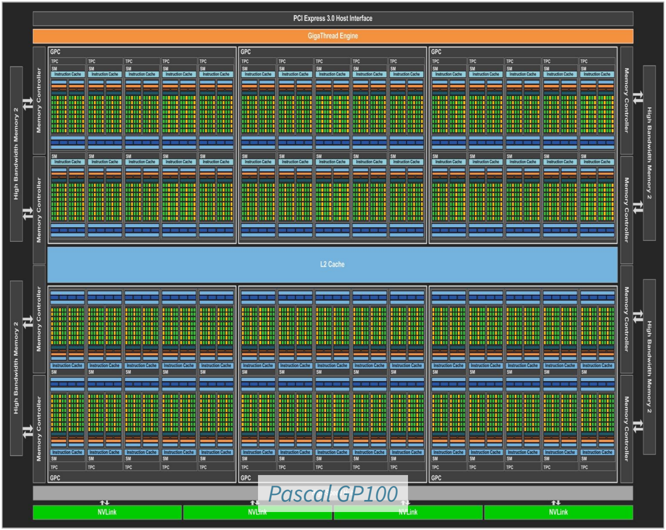


SIMT

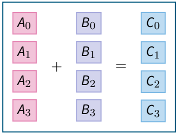


SIMT

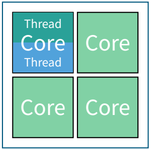
Of threads and warps



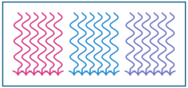
Vector



SMT



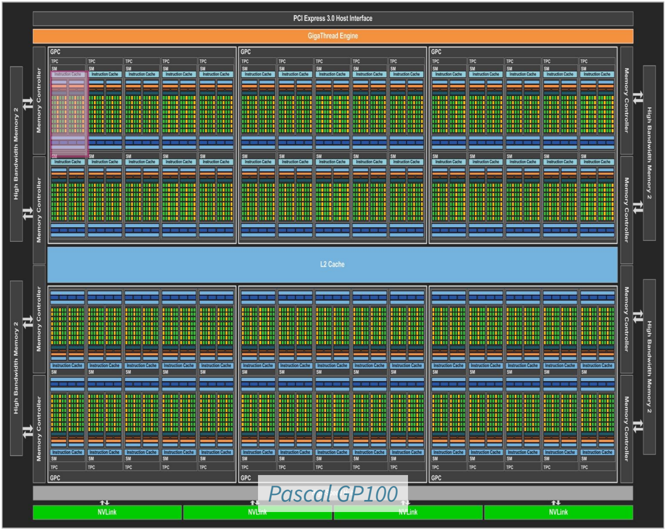
SIMT



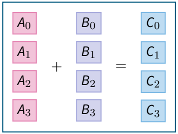
Graphics: Nvidia Corporation [8]

SIMT

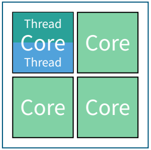
Of threads and warps



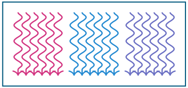
Vector



SMT



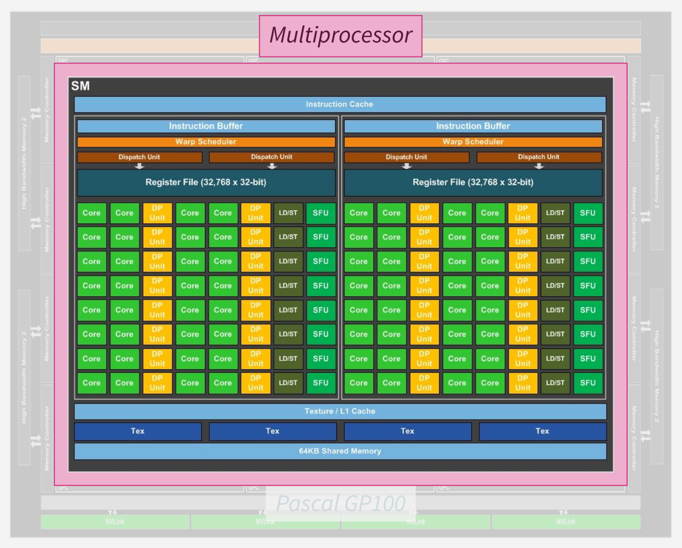
SIMT



Graphics: Nvidia Corporation [8]

SIMT

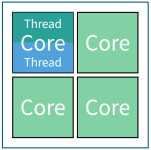
Of threads and warps



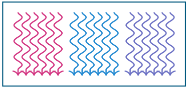
Vector

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

SMT



SIMT



Graphics: Nvidia Corporation [8]

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

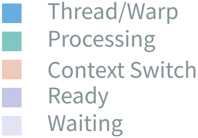
Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency

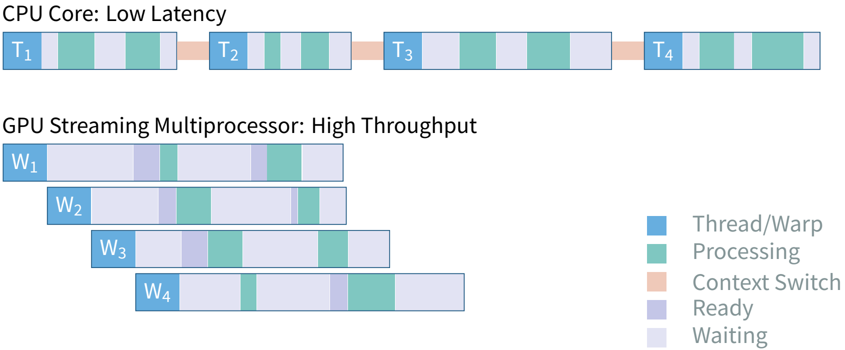


Low Latency vs. High Throughput

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**

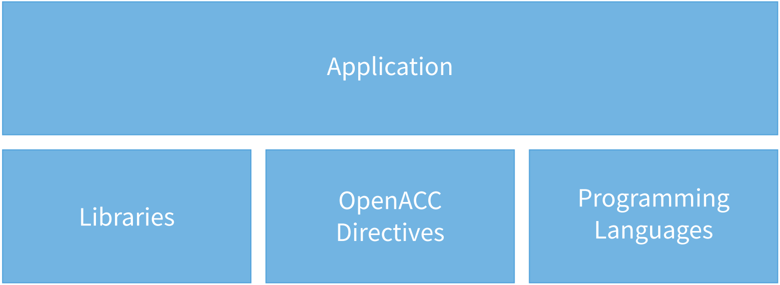
- \+ Large main memory
- \+ Fast clock rate
- \+ Large caches
- \+ Branch prediction
- \+ Powerful ALU
- − Relatively low memory bandwidth
- − Cache misses costly
- − Low performance per watt

Optimized for **high throughput**

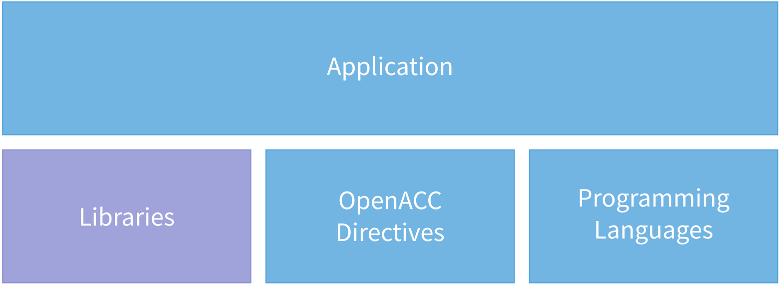
- \+ High bandwidth main memory
- \+ Latency tolerant (parallelism)
- \+ More compute resources
- \+ High performance per watt
- − Limited memory capacity
- − Low per-thread performance
- − Extension card

Programming GPUs

Summary of Acceleration Possibilities



Summary of Acceleration Possibilities



Libraries

The truth is out there!

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

Libraries

The truth is out there!

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

Use applications & libraries!

Libraries

The truth is out there!

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

Use applications & libraries!

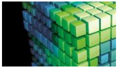
Wizard: Breazell [9]

Libraries

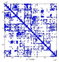
The truth is out there!

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

Use applications & libraries!



cuBLAS



cuSPARSE

cuDNN

Wizard: Breazell [9]

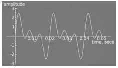
OpenCV

ArrayFire

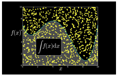
Thrust

Numba

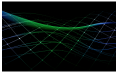
theano



cuFFT



cuRAND



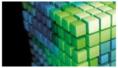
CUDA Math

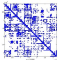
Libraries

The truth is out there!

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

Use applications & libraries!

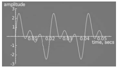

cuBLAS


cuSPARSE

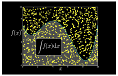
cuDNN

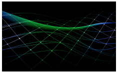
Wizard: Breazell [9]

OpenCV



cuFFT


cuRAND


CUDA Math

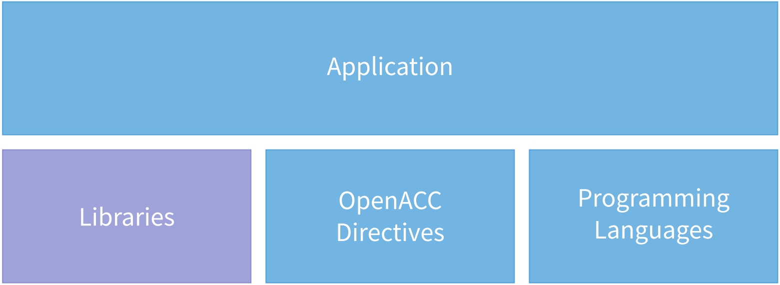
Thrust

ArrayFire

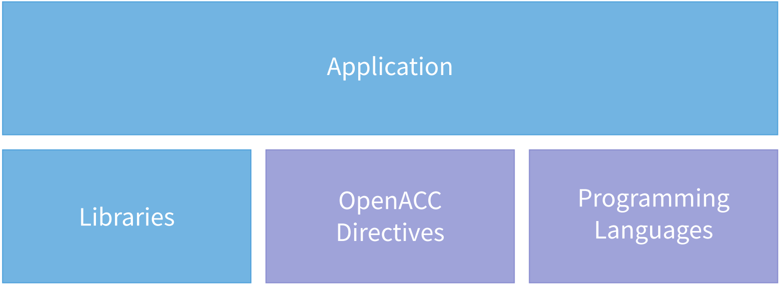
Numba

theano

Summary of Acceleration Possibilities



Summary of Acceleration Possibilities



Parallelism

Libraries are not enough?

You need to write your own GPU code?

Primer on Parallel Scaling

Amdahl's Law

Possible maximum speedup for N parallel processors

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

Primer on Parallel Scaling

Amdahl's Law

Possible maximum speedup for N parallel processors

Total Time	$t = t_{\mathbf{s}\text{erial}} + t_{\mathbf{p}\text{arallel}}$
N Processors	$t(N) = t_\text{s} + t_\text{p}/N$

Primer on Parallel Scaling

Amdahl's Law

Possible maximum speedup for *N* parallel processors

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

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

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

Efficiency: $\varepsilon = {}^{s}/_{N}$

Primer on Parallel Scaling

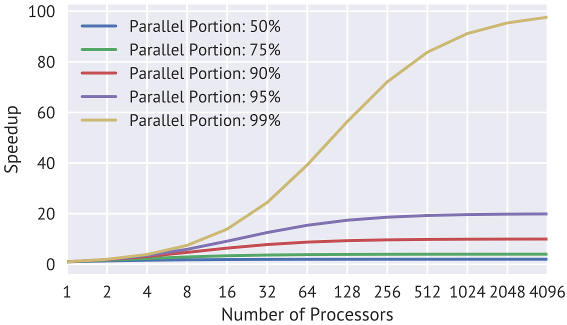
Amdahl's Law

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 $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p / N}$ Efficiency: $\varepsilon = {}^{s}/_{N}$



Parallelism

Parallel programming is not easy!

Things to consider:

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

Possibilities

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

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

Possibilities

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

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

Possibilities

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

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

CUDA SAXPY

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

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

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

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

cudaDeviceSynchronize();
```

CUDA SAXPY

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

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

Member of the Helmholtz Association

CUDA SAXPY

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

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

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

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

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

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

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

JÜLICH FORSCHUNGSZENTRUM

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

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

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

cudaDeviceSynchronize();
```

Specify kernel

ID variables

Guard against too many threads

Allocate GPU-capable memory

Call kernel 2 blocks, each 5 threads

CUDA SAXPY

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

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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 SAXPY

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

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

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

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

cudaDeviceSynchronize();
```

Specify kernel

ID variables

Guard against too many threads

Allocate GPU-capable memory

Call kernel 2 blocks, each 5 threads

Wait for kernel to finish

CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:
  - Thread

- Methods to exploit parallelism:

  - Threads

CUDA Threading Model

Warp the kernel, it's a thread!

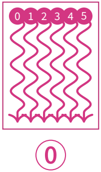
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$

Member of the Helmholtz Association

CUDA Threading Model

Warp the kernel, it's a thread!

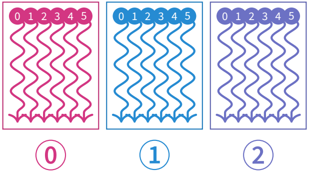
- Methods to exploit parallelism:
  - Threads → Block
  - Block



CUDA Threading Model

Warp the kernel, it's a thread!

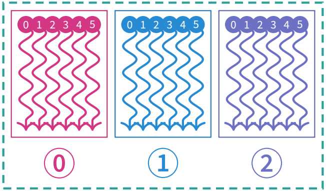
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks



CUDA Threading Model

Warp the kernel, it's a thread!

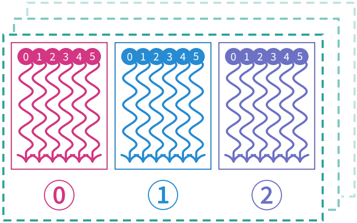
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid



CUDA Threading Model

Warp the kernel, it's a thread!

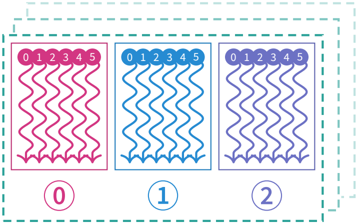
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



CUDA Threading Model

Warp the kernel, it's a thread!

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



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

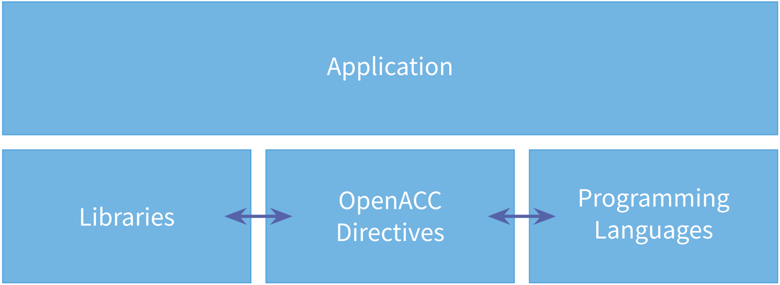
Application

Libraries

OpenACC Directives

Programming Languages

Summary of Acceleration Possibilities



Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Summary of Acceleration Possibilities

JÜLICH
FORSCHUNGSZENTRUM

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Conclusions

JÜLICH FORSCHUNGSZENTRUM

- 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 $\rightarrow$ **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

Member of the Helmholtz Association

Glossary I

JÜLICH FORSCHUNGSZENTRUM

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

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 3, 4, 5, 6, 7, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 95, 96, 97, 99

NVIDIA US technology company creating GPUs. 3, 4, 5, 6, 7, 12, 99

Glossary II

JÜLICH FORSCHUNGSZENTRUM

NVLink NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with 80 GB/s. PCI-Express: 16 GB/s. 12, 99

OpenACC Directive-based programming, primarily for many-core machines. 1, 72, 73, 74, 83, 84, 85, 86, 87, 88, 89, 90, 91

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, 72, 73, 74

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

OpenMP Directive-based programming, primarily for multi-threaded machines. 72, 73, 74

Glossary III

JÜLICH FORSCHUNGSZENTRUM

P100 A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast *HBM2* memory. 12

Pascal GPU architecture from NVIDIA (announced 2016). 99

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 75, 76, 77, 78, 79, 80, 81, 82

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/`. 72, 73, 74

References: Images, Graphics I

[5] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (pages 8–10).

[6] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (pages 17, 18).

[7] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (pages 17, 18).

References: Images, Graphics II

JÜLICH FORSCHUNGSZENTRUM

[8] Nvidia Corporation. *Pictures: Pascal Blockdiagram, Pascal Multiprocessor*. Pascal Architecture Whitepaper. URL: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` (pages 49–51).

[9] Wes Breazell. *Picture: Wizard*. URL: `https://thenounproject.com/wes13/collection/its-a-wizards-world/` (pages 59–63).

Member of the Helmholtz Association

References: Literature I

JÜLICH FORSCHUNGSZENTRUM

[1] Kenneth E. Hoff III et al. “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. ISBN: 0-201-48560-5. DOI: `10.1145/311535.311567`. URL: `http://dx.doi.org/10.1145/311535.311567` (pages 3–7).

[2] Chris McClanahan. “History and Evolution of GPU Architecture”. In: *A Survey Paper* (2010). URL: `http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf` (pages 3–7).

[3] Jack Dongarra et al. *TOP500*. Nov. 2016. URL: `https://www.top500.org/lists/2016/11/` (pages 3–7).

References: Literature II

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

[10] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.