CUDA Introduction

GPU Programming *Foundations* 2024

8 April 2024 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum

Outline

JÜLICH Forschungszentrum

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]

JÜLICH Forschungszentrum

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

2023 Top 500: 32 % with GPUs (9 of top 10) [4], Green 500: 48 of top 50 with GPUs [5]

JÜLICH Forschungszentrum

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

2023 Top 500: 32 % with GPUs (9 of top 10) [4], Green 500: 48 of top 50 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

2023 Top 500: 32 % with GPUs (9 of top 10) [4], Green 500: 48 of top 50 with GPUs [5]

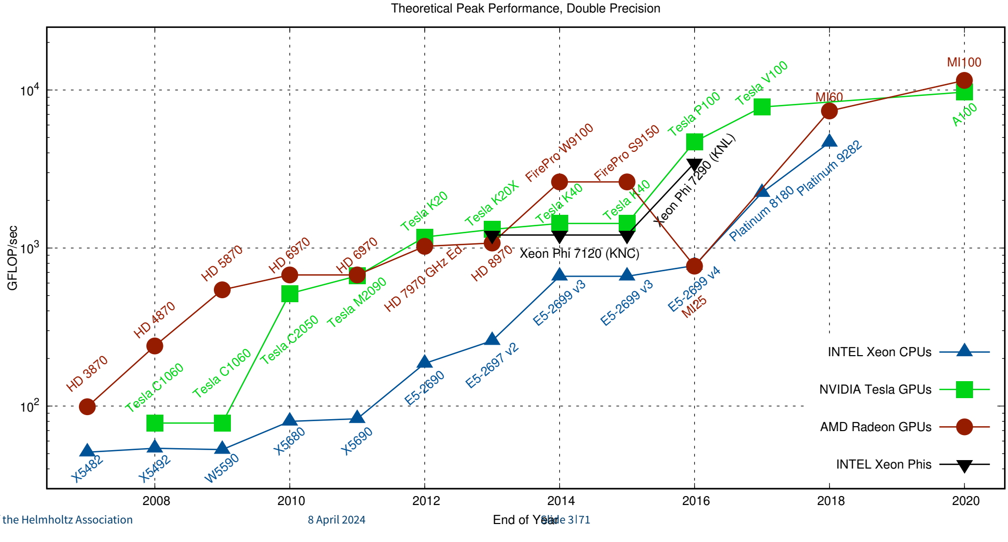
2023 🇪🇺: Leonardo (238 PFLOP/s*, Italy), NVIDIA GPUs; LUMI (309 PFLOP/s*, Finland), AMD GPUs
🇺🇸: Frontier (1.102 EFLOP/s*, ORNL), AMD GPUs

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

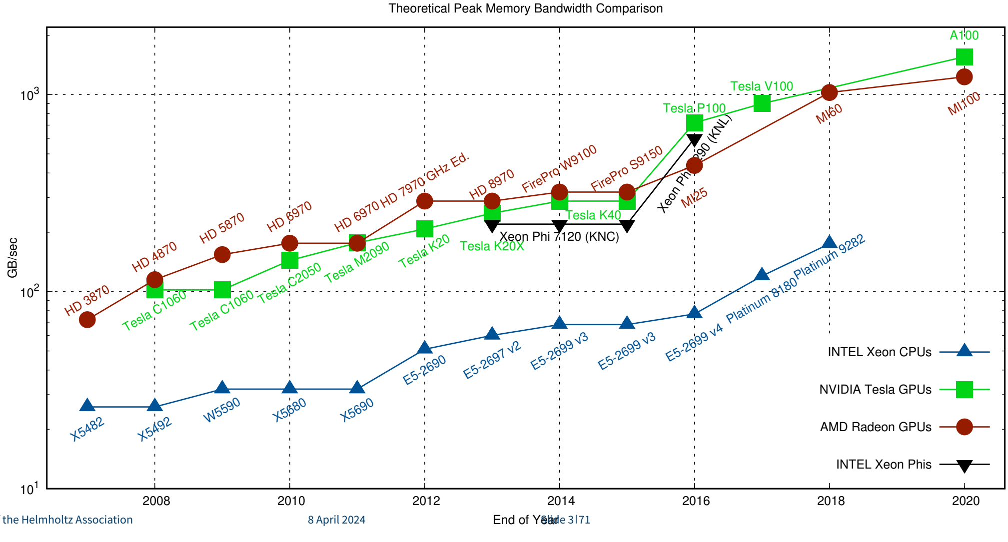
Performance



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 × 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

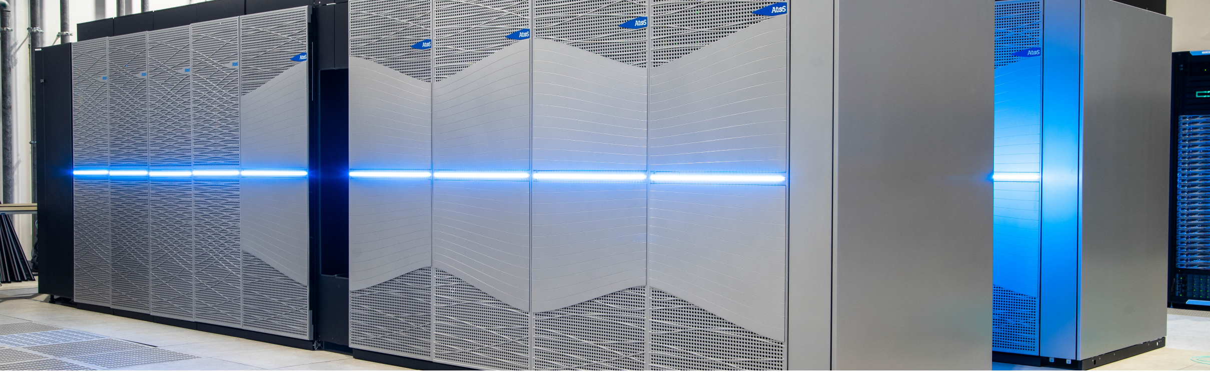
Top500 List Nov 2020:

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

JUWELS Booster – Scaling Higher!

- 936 nodes with AMD EPYC Rome CPUs (2 × 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

JÜLICH Forschungszentrum



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

JUPITER – Exascale

- First Exascale system in Europe
- Procured by EuroHPC JU, BMBF, MKW-NRW, hosted by JSC
- Currently in pre-installation
- 24 000 NVIDIA H100 GPUs (Grace-Hopper superchips)
- 1 EFLOP/s FP64 (HPL), 32 EFLOP/s FP8 (peak)

→ `jupiter.fz-juelich.de`

JÜLICH Forschungszentrum

Getting GPU-Acquainted

TASK

Some Applications

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

TASK

Some Applications

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!

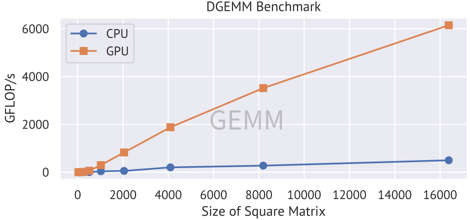
Mandelbrot

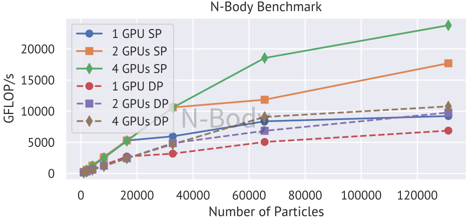
Dot Product

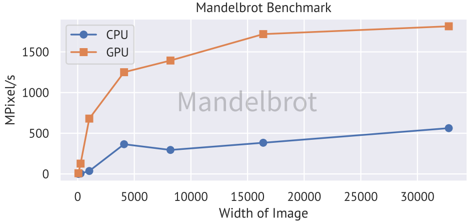
Getting GPU-Acquainted

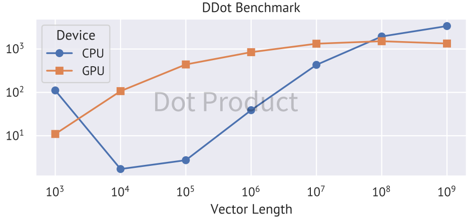
TASK

Some Applications









Platform

CPU vs. GPU

A matter of specialties

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

CPU vs. GPU

A matter of specialties

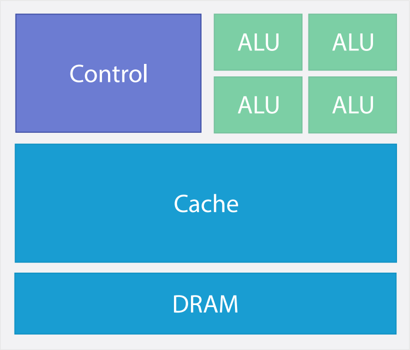
Transporting one

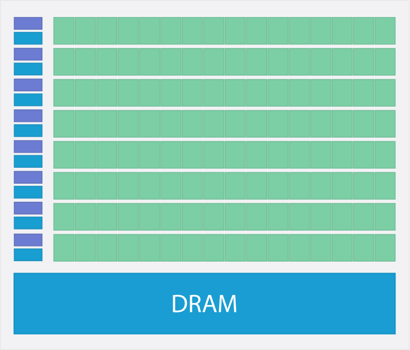
Transporting many

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

CPU vs. GPU

Chip





GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

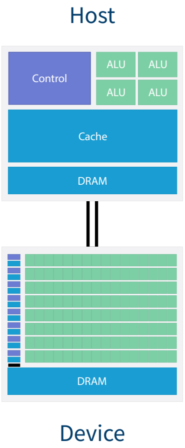
Memory

Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU



Memory

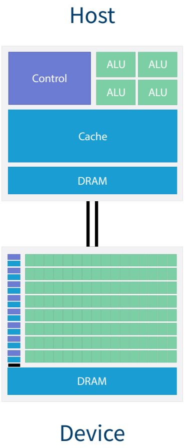
GPU memory ain't no CPU memory

Unified Virtual Addressing

- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA



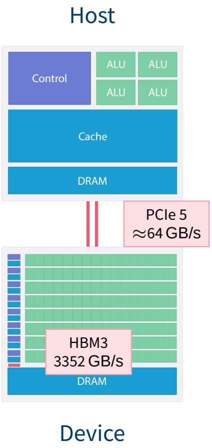
Memory

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

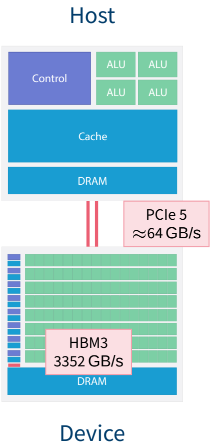
- GPU: accelerator / extension card

→ 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

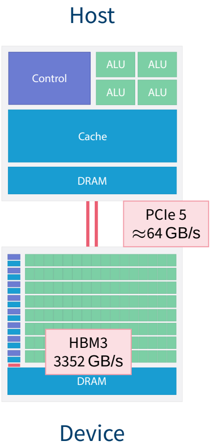
→ Separate device from CPU

Separate memory, but UVA and UM

- Memory transfers need special consideration!

Do as little as possible!

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



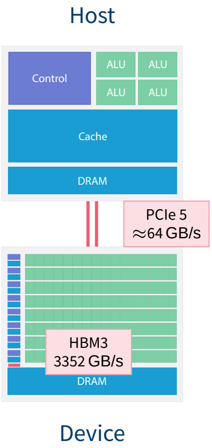
Memory

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

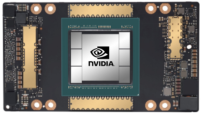
- Memory transfers need special consideration!

Do as little as possible!

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

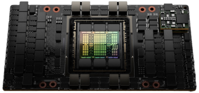
A100

40 GB RAM, 1555 GB/s

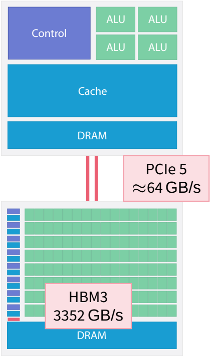


H100

80 GB RAM, 3352 GB/s



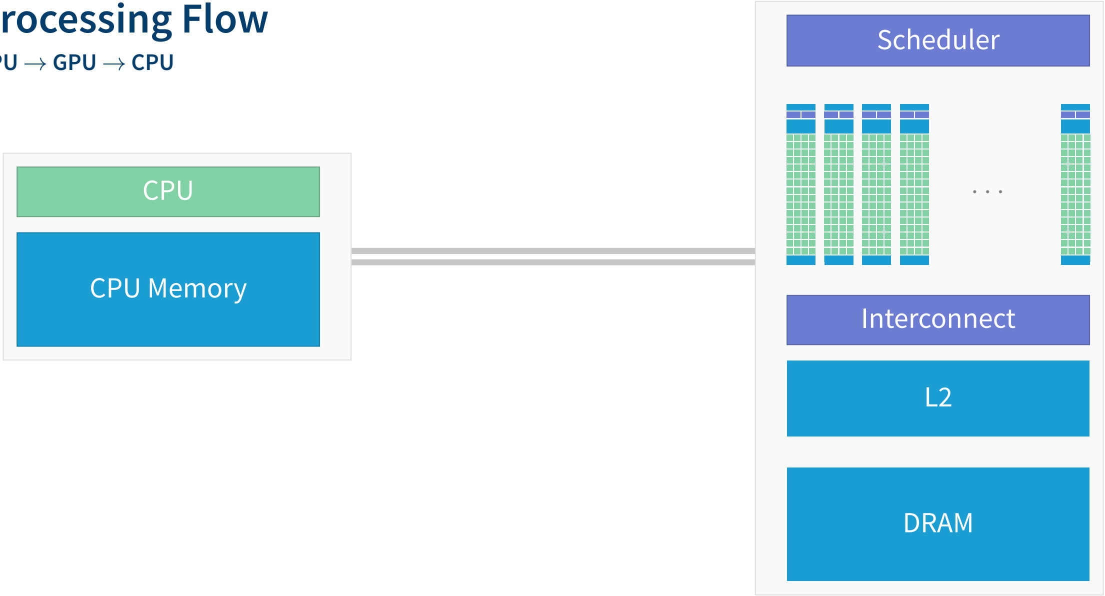
Host



Device

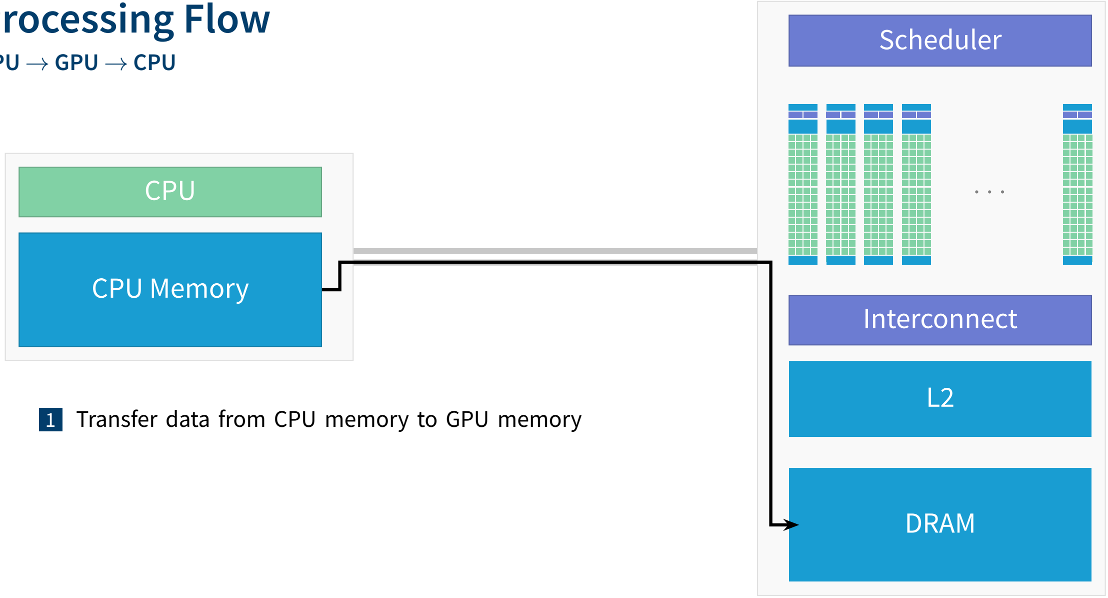
Processing Flow

CPU $\longrightarrow$ GPU $\longrightarrow$ CPU



Processing Flow

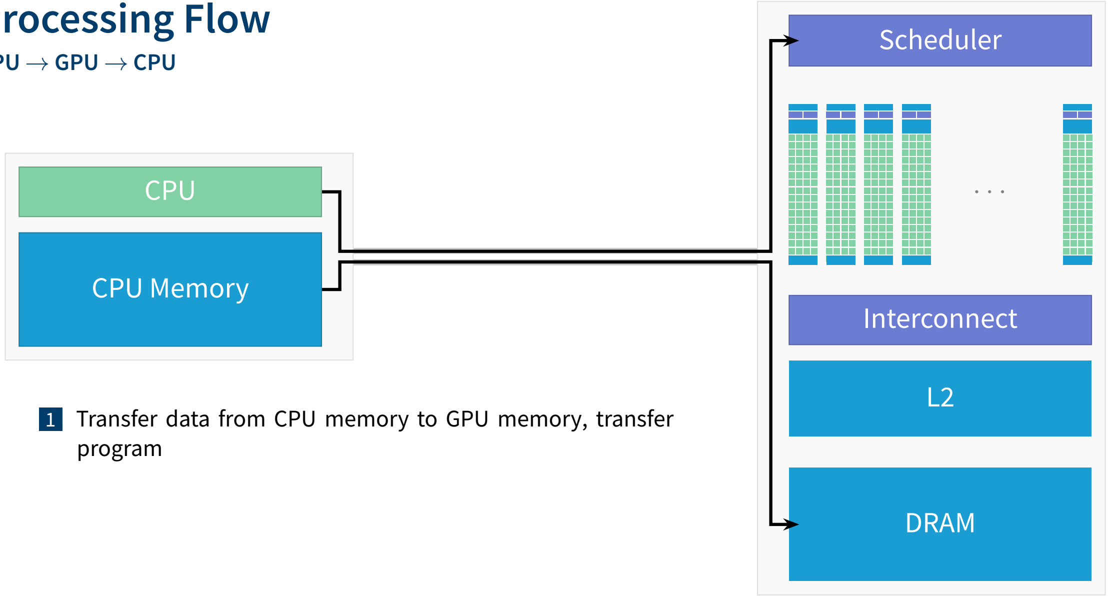
CPU → GPU → CPU



1 Transfer data from CPU memory to GPU memory

Processing Flow

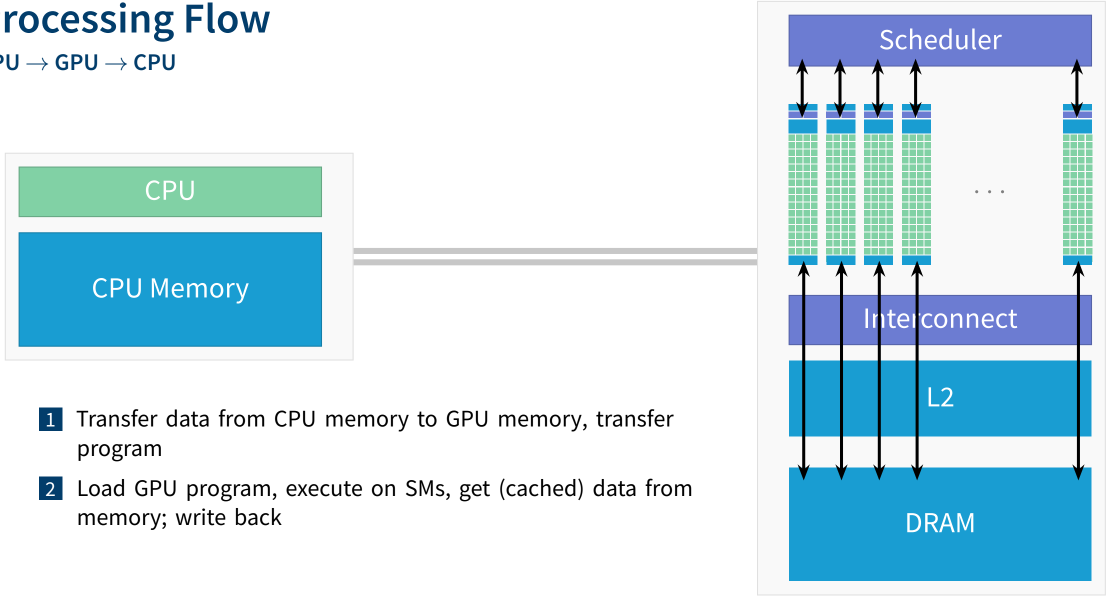
CPU → GPU → CPU



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

Processing Flow

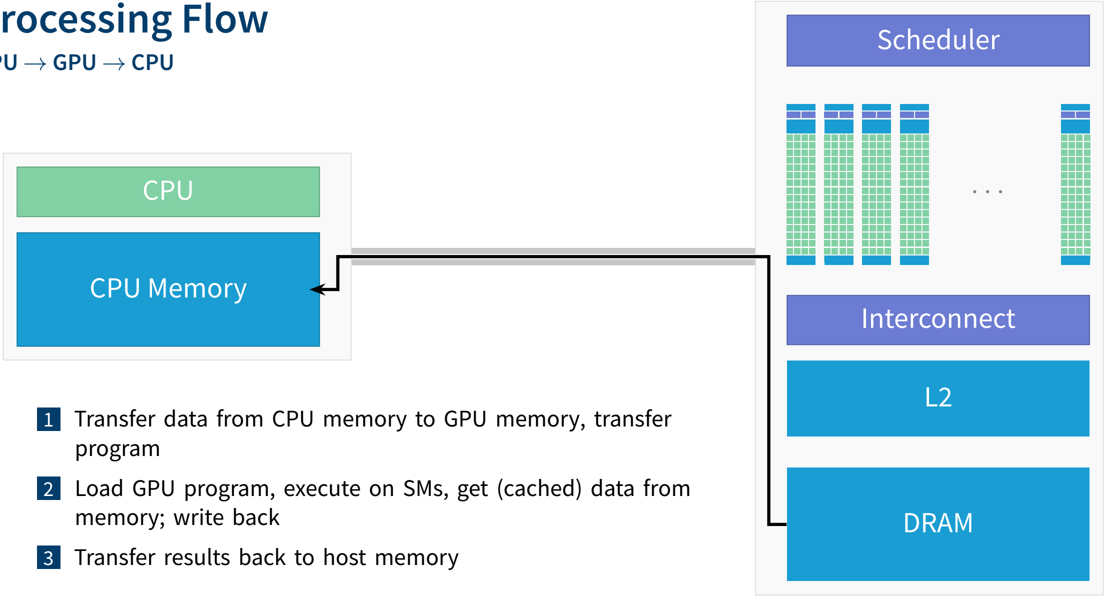
CPU → GPU → CPU



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

Flynn's Taxonomy

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

Flynn's Taxonomy

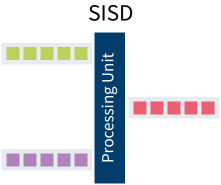
- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix} \textbf{S}\text{ingle} \\ \textbf{M}\text{ultiple} \end{pmatrix} \otimes \begin{pmatrix} \textbf{I}\text{nstruction} \\ \textbf{D}\text{ata} \end{pmatrix}$

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



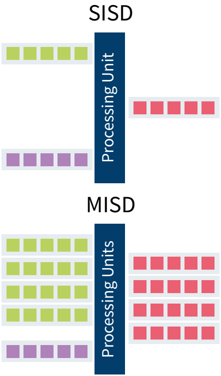
- Michael Flynn (1966/1972): classification of computer architectures
- Define by number of instructions operating on data elements
- $\begin{pmatrix} \textbf{Single} \\ \textbf{Multiple} \end{pmatrix} \otimes \begin{pmatrix} \text{Instruction} \\ \text{Data} \end{pmatrix}$

  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
- $\begin{pmatrix}\text{Single}\\ \text{Multiple}\end{pmatrix} \otimes \begin{pmatrix}\text{Instruction}\\ \text{Data}\end{pmatrix}$

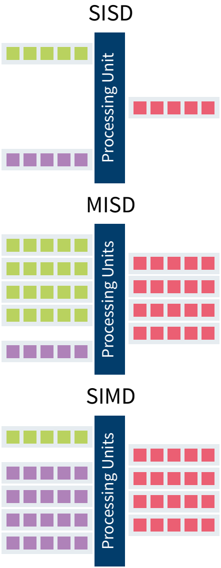
  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data



Flynn's Taxonomy

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

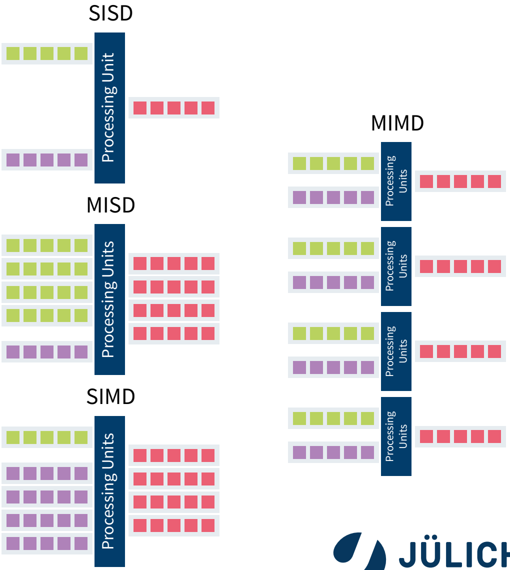
  SISD Single Instruction, Single Data
  MISD Multiple Instructions, Single Data
  SIMD Single Instruction, Multiple Data



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

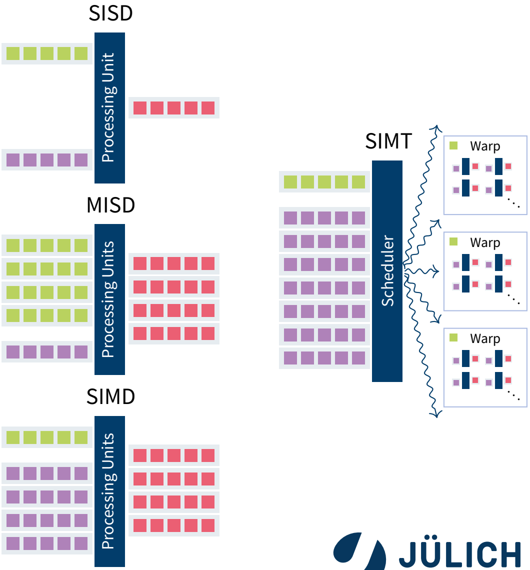


JÜLICH Forschungszentrum

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*



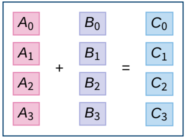
JÜLICH Forschungszentrum

SIMT

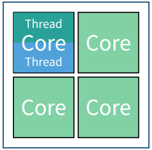
SIMT = SIMD ⊕ SMT

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

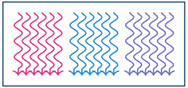
Vector



SMT

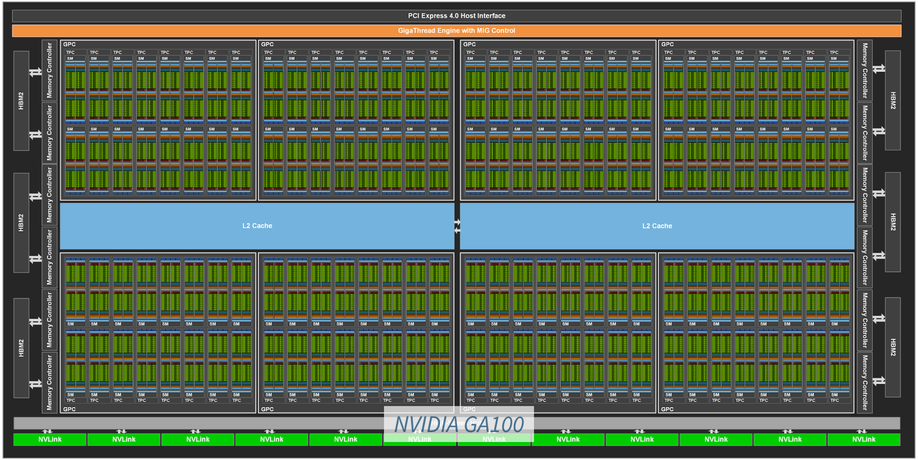


SIMT

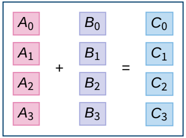


SIMT

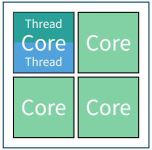
SIMT = SIMD ⊕ SMT



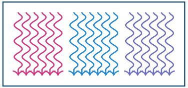
Vector



SMT



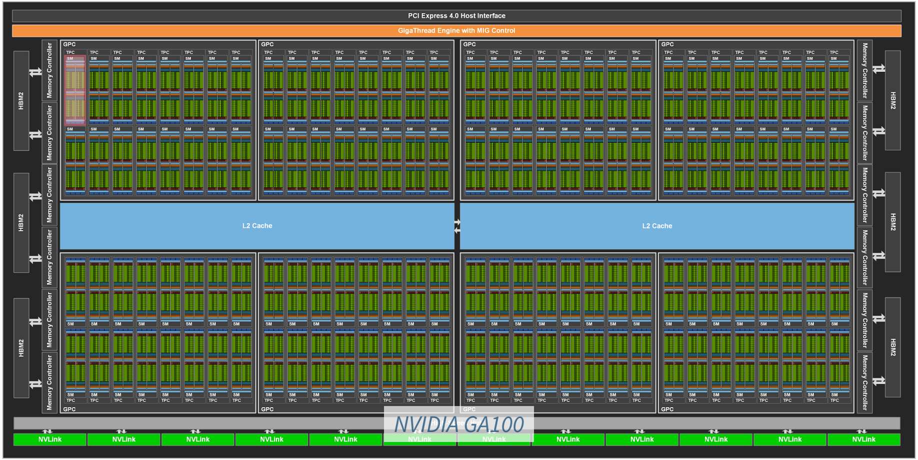
SIMT



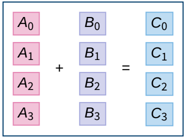
Graphics: img:amperepictures

SIMT

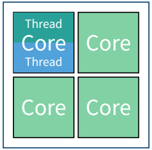
SIMT = SIMD ⊕ SMT



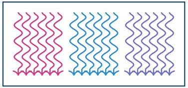
Vector



SMT



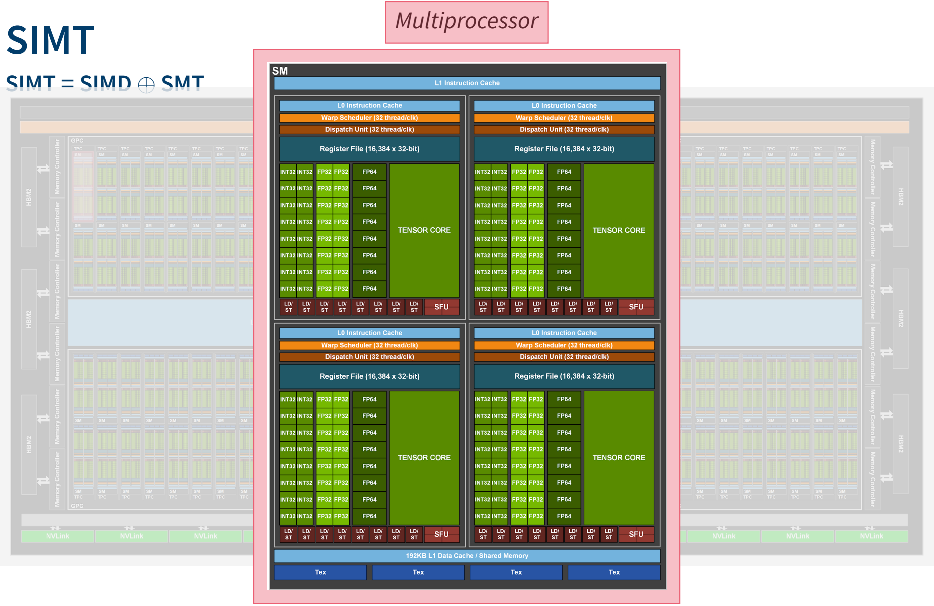
SIMT

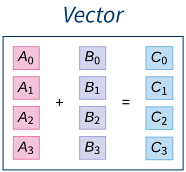


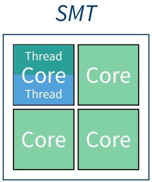
Graphics: img:amperepictures

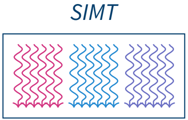
SIMT

SIMT = SIMD ⊕ SMT







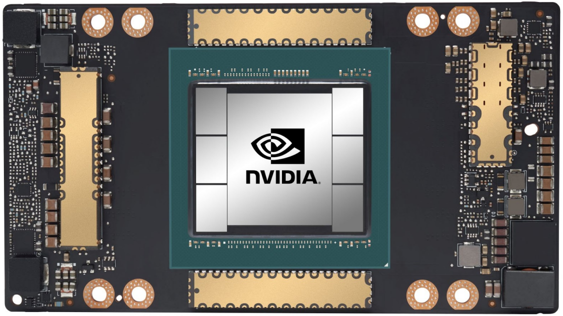


Graphics: img:amperepictures

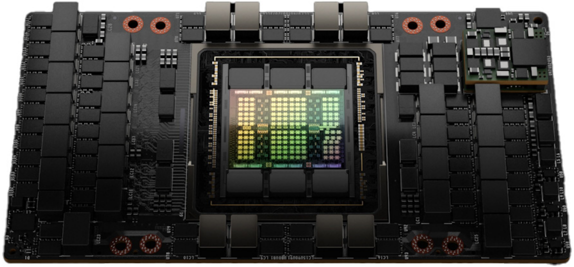
A100 vs H100

Comparison of last vs. current generation

A100



H100



JÜLICH Forschungszentrum

A100 vs H100

Comparison of last vs. current generation

A100

H100

A100 vs H100

Comparison of last vs. current 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

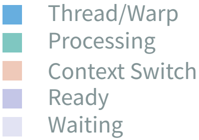
Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency



Low Latency vs. High Throughput

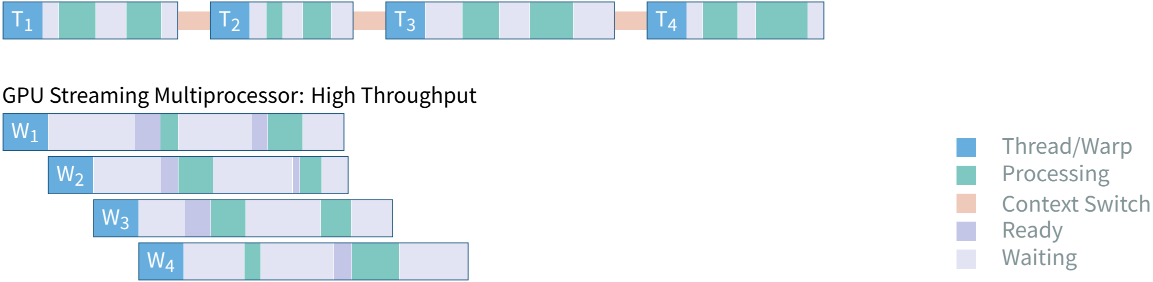
Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency

GPU Streaming Multiprocessor: High Throughput

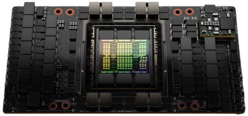


CPU vs. GPU

Let’s summarize this!

Optimized for **low latency**

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



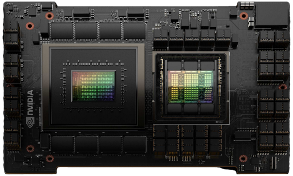
Optimized for **high throughput**

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

CPU-GPU Convergence

NVIDIA GH200, AMD MI300A

- Recent trend: combine CPU and GPU into *one package*
- NVIDIA Grace-Hopper Superchip GH200
  - **Grace**: NVIDIA's first CPU (Arm-based, 72 cores, 512 GB LPDDR5X RAM)
  - **Hopper**: NVIDIA's current GPU (usually: H100; 132 multiprocessors)
  - **GH200**: CPU and GPU in one package, fused together into *superchip*; 900 GB/s CPU-GPU bandwidth

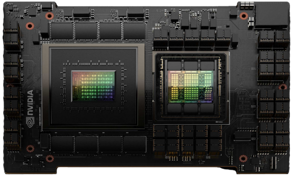


GH200 Superchip (NVIDIA)

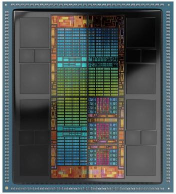
CPU-GPU Convergence

NVIDIA GH200, AMD MI300A

- Recent trend: combine CPU and GPU into *one package*
- NVIDIA Grace-Hopper Superchip GH200
  - **Grace**: NVIDIA’s first CPU (Arm-based, 72 cores, 512 GB LPDDR5X RAM)
  - **Hopper**: NVIDIA’s current GPU (usually: H100; 132 multiprocessors)
  - **GH200**: CPU and GPU in one package, fused together into *superchip*; 900 GB/s CPU-GPU bandwidth
- AMD Instinct MI300A APU
  - **MI300A**: MI300 GPU chiplets (228 compute units) with Zen CPU chiplets (24 cores)
  - One shared memory (HBM: 128 GB, 5.3 TB/s)



GH200 Superchip (NVIDIA)



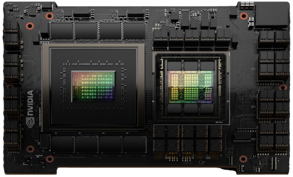
MI300A APU (AMD)

JÜLICH Forschungszentrum

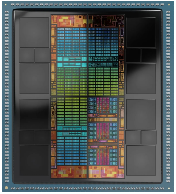
CPU-GPU Convergence

NVIDIA GH200, AMD MI300A

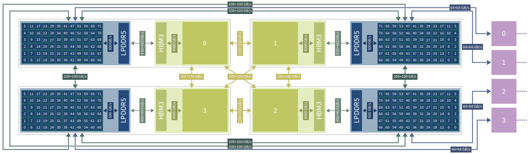
- Recent trend: combine CPU and GPU into *one package*
- NVIDIA Grace-Hopper Superchip GH200
  - **Grace**: NVIDIA's first CPU (Arm-based, 72 cores, 512 GB LPDDR5X RAM)
  - **Hopper**: NVIDIA's current GPU (usually: H100; 132 multiprocessors)
  - **GH200**: CPU and GPU in one package, fused together into *superchip*; 900 GB/s CPU-GPU bandwidth
- AMD Instinct MI300A APU
  - **MI300A**: MI300 GPU chiplets (228 compute units) with Zen CPU chiplets (24 cores)
  - One shared memory (HBM: 128 GB, 5.3 TB/s)



GH200 Superchip (NVIDIA)



MI300A APU (AMD)

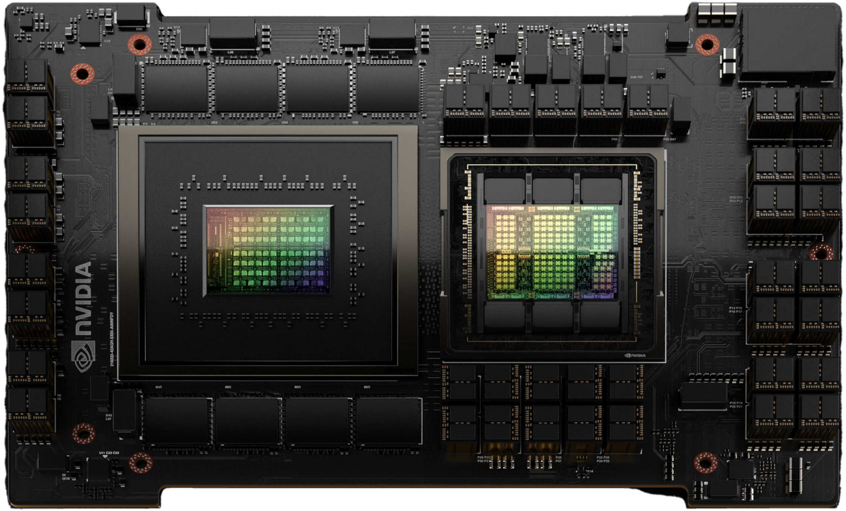


JUPITER node design

CPU-G

NVIDIA GH2

- Rece
  pack
- NVID
- AMD
  - One shared memory (HBM: 128 GB, 5.3 TB/s)



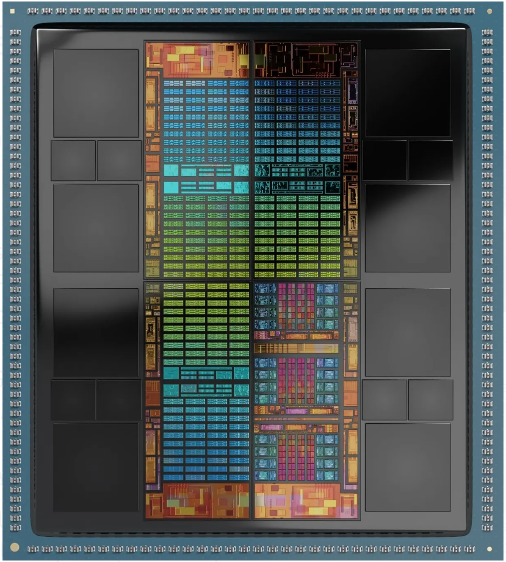
A APU (AMD)

JÜLICH Forschungszentrum

CPU-GPU Converg

NVIDIA GH200, AMD MI300A

- Recent trend: combin
  package
- NVIDIA Grace-Hopper
  - Grace: NVIDIA's firs
    512 GB LPDDR5X R
  - Hopper: NVIDIA's c
    132 multiprocesso
  - GH200: CPU and G
    together into *supe*
    bandwidth
- AMD Instinct MI300A A
  - MI300A: MI300 GPU
    units) with Zen CP
  - One shared memory (HBM: 128 GB, 5.3 TB/s)

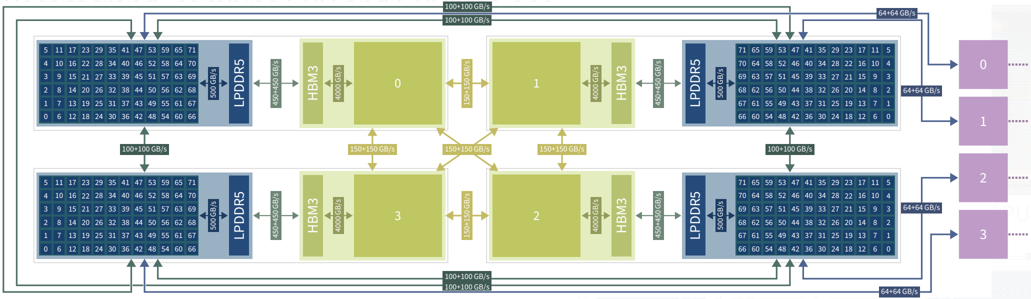


ip (NVIDIA) MI300A APU (AMD)

PITER node design

CPU-GPU Convergence

NVIDIA GH200, AMD MI300A



bandwidth

- AMD Instinct MI300A APU
  - MI300A: MI300 GPU chiplets (228 compute units) with Zen CPU chiplets (24 cores)
  - One shared memory (HBM: 128 GB, 5.3 TB/s)

JUPITER node design

Programming GPUs

Preface: CPU

A simple CPU program!

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

Part of LAPACK BLAS Level 1

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

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

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

Summary of Acceleration Possibilities

Application

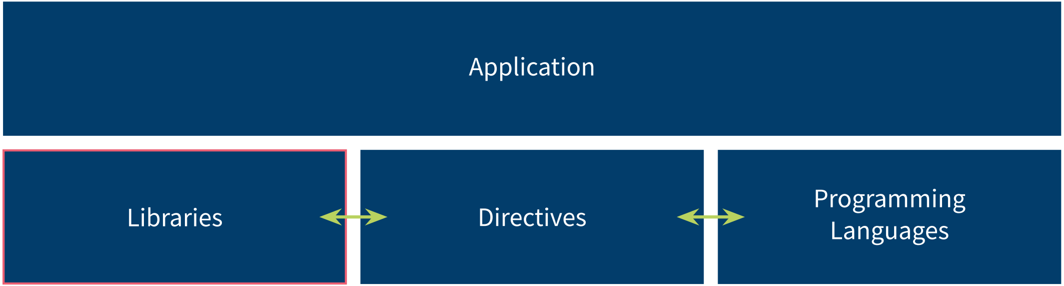
Libraries ↔ Directives ↔ Programming Languages

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Summary of Acceleration Possibilities



Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

JÜLICH Forschungszentrum

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!

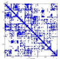
Use applications & libraries

Wizard: Breazell [10]

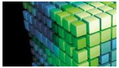
Libraries

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

Use applications & libraries



cuSPARSE



cuBLAS

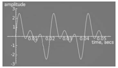
cuDNN

OpenCV

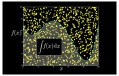
Thrust

ArrayFire

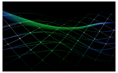
Numba



cuFFT



cuRAND



CUDA Math

CuPy

Wizard: Breazell [10]

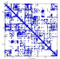
JÜLICH Forschungszentrum

Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries

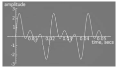
cuBLAS



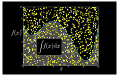
cuSPARSE

cuDNN

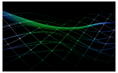
OpenCV



cuFFT



cuRAND



CUDA Math

Thrust

ArrayFire

Numba

CuPy

Wizard: Breazell [10]

cuBLAS

Parallel algebra

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

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

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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


JÜLICH Forschungszentrum

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

Allocate GPU memory

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

Allocate GPU memory

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

Allocate GPU memory

Call BLAS routine

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

Allocate GPU memory

Call BLAS routine

Copy result to host

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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

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

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

Initialize

Allocate GPU memory

Call BLAS routine

Copy result to host

Finalize

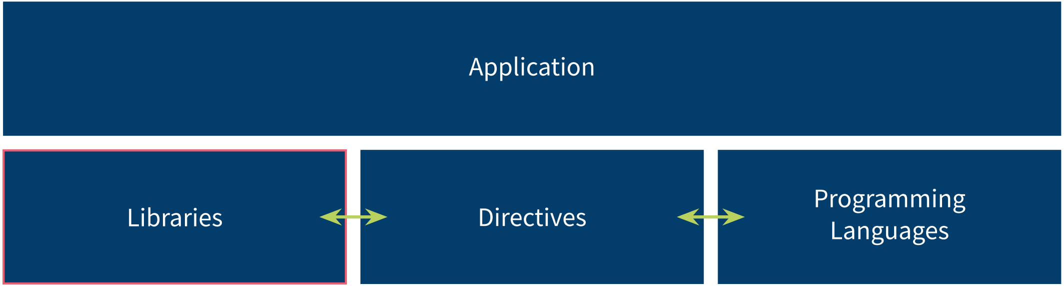
cuBLAS Task

TASK

Implement a matrix-matrix multiplication

- Location of code: `01-Basics/exercises/tasks/02-cuBLAS`
- Look at `Instructions.ipynb` Notebook for instructions
  1. Implement call to double-precision GEMM of cuBLAS
  2. Build with `make` (load modules of this task via `source setup.sh`!)
  3. Run with `make run`
- Check cuBLAS documentation for details on `cublasDgemm()`

Summary of Acceleration Possibilities



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

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

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

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

Primer on Parallel Scaling

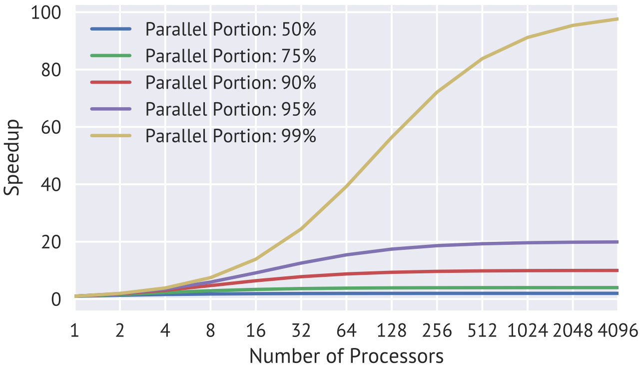
Amdahl's Law

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

Directives

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

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

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives
  ```
  #pragma acc loop
  for (int i = 0; i < 1; i++) {};
  ```
- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- Compiler interprets directives, creates according instructions

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

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

- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- Compiler interprets directives, creates according instructions

Pro

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

Con

- Only few compilers
- Not all the raw power available
- A little harder to debug

GPU Programming with Directives

The power of… two.

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

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

OpenACC Similar to OpenMP, but more specifically for GPUs

For C/C++ and Fortran

OpenACC

Code example

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

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

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

OpenACC

Code example

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

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

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

Programming GPUs

Thrust

Thrust

Iterators! Iterators everywhere! 🚀

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- *A precursor to a GPU-accelerated pSTL?*
- Based on iterators
- Data-parallel primitives (`scan()`, `sort()`, `reduce()`, ...)
- Fully compatible with plain CUDA C (comes with CUDA Toolkit)
- Great with `[](){}` lambdas!

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

Thrust

Code example

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

thrust::device_vector d_x = x, d_y = y;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), [=]
↪  __device__ (auto x, auto y) {return a*x+y;});
// or:
using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 +
↪  _2);

x = d_x;
```

Thrust Task

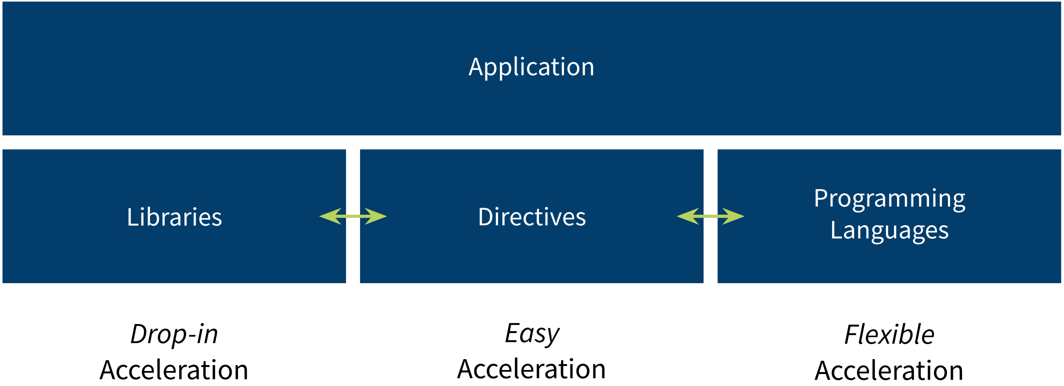
TASK

Let's sort some randomness

- Location of code: `01-Basics/exercises/tasks/03-Thrust`
- Look at `Instructions.ipynb` for instructions
  1. Sort random numbers with Thrust on CPU and GPU
  2. Build with `make`
     Reset environment to original; call `source setup.sh` or re-login!
  3. Run with `make run`
- Check Thrust documentation for details on `thrust::sort()`

JÜLICH Forschungszentrum

Summary of Acceleration Possibilities



Programming GPUs

CUDA C/C++

CUDA SAXPY

With runtime-managed data transfers

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

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

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

cudaDeviceSynchronize();
```

CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

CUDA's Parallel Model

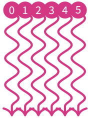
In software: Threads, Blocks

- Methods to exploit parallelism:
  - Thread

CUDA’s Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads



CUDA's Parallel Model

In software: Threads, Blocks

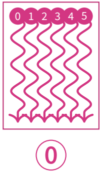
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$

JÜLICH Forschungszentrum

CUDA’s Parallel Model

In software: Threads, Blocks

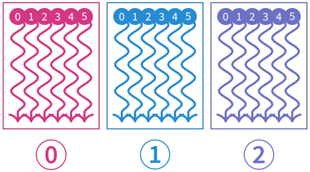
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$
  - $\boxed{\text{Block}}$



CUDA's Parallel Model

In software: Threads, Blocks

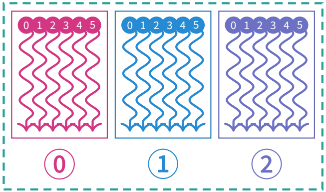
- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$
  - $\boxed{\text{Blocks}}$



CUDA's Parallel Model

In software: Threads, Blocks

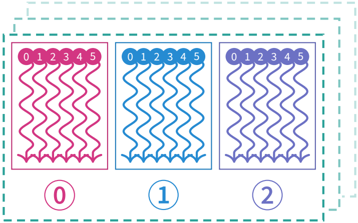
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid



CUDA's Parallel Model

In software: Threads, Blocks

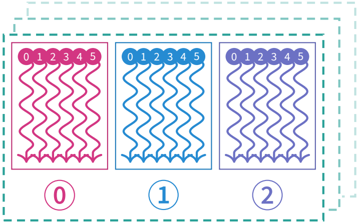
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



CUDA's Parallel Model

In software: Threads, Blocks

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



- Parallel function: **kernel**
  - `__global__ kernel(int a, float * b) { }`
  - Access own ID by global variables `threadIdx.x`, `blockIdx.y`, …
- Execution entity: **threads**
  - Lightweight → fast switchting!
  - 1000s threads execute simultaneously → order non-deterministic!

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device
- All threads execute same code; but can take different paths in program flow (some penalty)

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device
- All threads execute same code; but can take different paths in program flow (some penalty)
- Info about thread: local, global IDs

```
int currentThreadId = threadIdx.x;
float x = input[currentThreadId];
output[currentThreadId] = x*x;
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (
        int i = 0;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
          ;
        i++
    )
        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove for

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove for | Add global

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function $\rightarrow$ CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove for | Add global

Replace i by threadIdx.x

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x;





        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Summary

- C function with explicit loop

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

- CUDA kernel with implicit loop

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N)
        out[i] = scale * in[i];
}
```

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!
- Example:

```
int nThreads = 32;
scale<<<N/nThreads, nThreads>>>(23, in, out, N)
```

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!
- Example:

```
int nThreads = 32;
scale<<<N/nThreads, nThreads>>>(23, in, out, N)
```

- Possibility for too many threads; include termination condition into kernel!

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

`shared` Dynamic **shared memory**

- Small GPU memory space; share data in block (high bandwidth)
- Shared memory: allocate statically (compile time) or dynamically (run time)
- `size_t shared`: bytes of shared memory allocated per block (in addition to static shared memory)

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

`shared` Dynamic **shared memory**

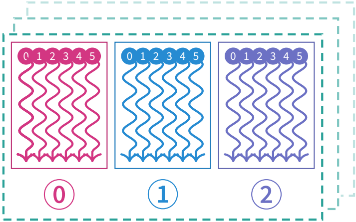
- Small GPU memory space; share data in block (high bandwidth)
- Shared memory: allocate statically (compile time) or dynamically (run time)
- `size_t shared`: bytes of shared memory allocated per block (in addition to static shared memory)

`stream` Associated **CUDA stream**

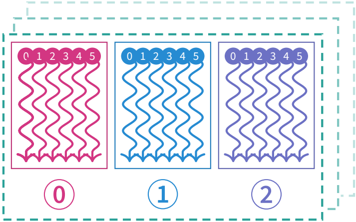
- CUDA streams enable different channels of communication with GPU
- Can overlap in some cases (communication, computation)
- `cudaStream_t stream`: ID of stream to use for this kernel launch

Grid Dimensions

- Threads & blocks in 3D



Grid Dimensions

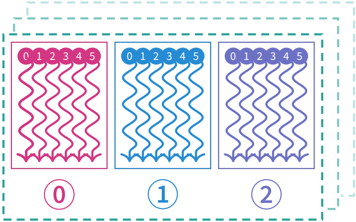


- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

Grid Dimensions



- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

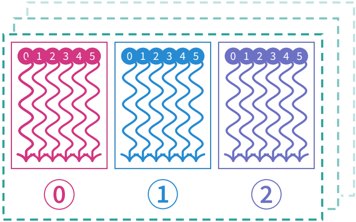
```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

- Example:

```
dim3 blockDim(32, 32);
dim3 gridDim = {1000, 100};
```

Grid Dimensions



- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

- Example:

```
dim3 blockDim(32, 32);
dim3 gridDim = {1000, 100};
```

- Kernel call with `dim3`

```
kernel<<<dim3 gridDim, dim3 blockDim>>>(...)
```

Grid Sizes

- Block and grid sizes are hardware-dependent

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

Block
- $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
- $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

JÜLICH Forschungszentrum

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

Block
- $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
- $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

Grid
- $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

  Block
  - $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
  - $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

  Grid
  - $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$
- Find out yourself: `deviceQuery` example from CUDA Samples

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

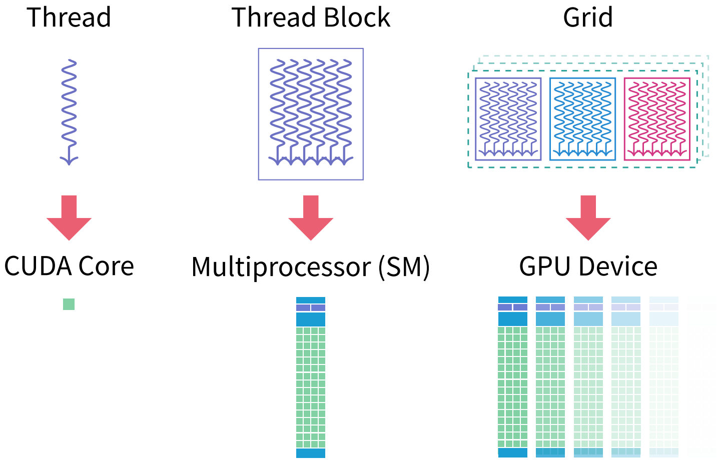
  Block
  - $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
  - $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

  Grid
  - $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$
- Find out yourself: `deviceQuery` example from CUDA Samples
- Workflow: Chose 128 or 256 as block dim; calculate grid dim from problem size

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16);
int gx = (Nx % blockDim.x == 0)  Nx / blockDim.x : Nx / blockDim.x + 1;
int gy = (Ny % blockDim.y == 0)  Ny / blockDim.y : Ny / blockDim.y + 1;
dim3 gridDim(gx, gy);
kernel<<<gridDim, blockDim>>>();
```

Hardware Threads

Mapping Software Threads to Hardware



GPU Memory

- Data needs to reach the GPU; many ways to do so
- Progression

`cudaMalloc()`	First: Manual transfers via dedicated API
`cudaMallocManaged()`	Then: Automated transfers via dedicated API
`malloc()`	Now: Automated transfers via usual API

- `malloc()` has some caveats (system support) → *Full CUDA Unified Memory Support*

→ CUDA documentation *Unified Memory Programming*

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*
- Example:

```
float * a;
int N = 2048;
cudaMallocManaged(&a, N * sizeof(float));
```


JÜLICH Forschungszentrum

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*
- Example:

```
float * a;
int N = 2048;
cudaMallocManaged(&a, N * sizeof(float));
```

- Free device memory

```
cudaFree(void* ptr)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

- Copy data between host $\leftrightarrow$ device

```
cudaMemcpy(void* dst, void* src, size_t nByte, enum cudaMemcpyKind dir)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

- Copy data between host $\leftrightarrow$ device

```
cudaMemcpy(void* dst, void* src, size_t nByte, enum cudaMemcpyKind dir)
```

- Example:

```
float * a, * a_d;
int N = 2048;
// fill a
cudaMalloc(&a_d, N * sizeof(float));
cudaMemcpy(a_d, a,   N * sizeof(float), cudaMemcpyHostToDevice);
kernel<<<1,1>>>(a_d, N);
cudaMemcpy(a   , a_d, N * sizeof(float), cudaMemcpyDeviceToHost);
```

Task: Scale Vector

TASK

Work on an Array of Data

- Location of code: `01-Basics/exercises/tasks/04-Scale-Vector`
- Look at `Instructions.ipynb` for instructions
  1. Implement the whole CUDA flow (allocation, kernel configuration, kernel launch)
  2. Build with `make`
  3. Run with `make run`
- Additional task: Look at the version with explicit transfers (`_et`)

Task: Jacobi

TASK

Implement Manual Memory Handling

- Location of code: `01-Basics/exercises/tasks/05-Jacobi-Explicit-Transfers`
- Look at `Instructions.ipynb` for instructions
  1. Port the application from Unified Memory to manual memory handling
  2. Build with `make`
  3. Run with `make run`

Unified Memory

Overview

- Everything started with manual data management
- First Unified Memory since CUDA 6.0
- Better Unified Memory better since CUDA 8.0
- Now: Unified Memory great default, explicit memory only a possible optimization

Manual Memory vs. Unified Memory

```
void sortfile(FILE *fp, int N) {
    char *data;
    char *data_d;

    data = (char *)malloc(N);
    cudaMalloc(&data_d, N);

    fread(data, 1, N, fp);

    cudaMemcpy(data_d, data, N, cudaMemcpyHostToDevice);
    kernel<<<...>>>(data, N);

    cudaMemcpy(data, data_d, N, cudaMemcpyDeviceToHost);
    host_func(data)
    cudaFree(data_d); free(data);
}
```

```
void sortfile(FILE *fp, int N) {
    char *data;


    cudaMallocManaged(&data, N);


    fread(data, 1, N, fp);


    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data);
}
```

Implementation Details

Under the hood

```
cudaMallocManaged(&ptr, ...);
```

```
*ptr = 1;
```

```
kernel<<<...>>>(ptr);
```

Implementation Details

Under the hood

```
cudaMallocManaged(&ptr, ...);
```

Empty! No pages anywhere yet (like `malloc()`)

```
*ptr = 1;
```

```
kernel<<<...>>>(ptr);
```

Implementation Details

Under the hood

`cudaMallocManaged(&ptr, ...);` ← Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← CPU page fault: data allocates on CPU

`kernel<<<...>>>(ptr);`

Implementation Details

Under the hood

`cudaMallocManaged(&ptr, ...);` ← Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← CPU page fault: data allocates on CPU

`kernel<<<...>>>(ptr);` ← GPU page fault: data migrates to GPU

Implementation Details

Under the hood

`cudaMallocManaged(&ptr, ...);` ← Empty! No pages anywhere yet (like `malloc()`)

`*ptr = 1;` ← CPU page fault: data allocates on CPU

`kernel<<<...>>>(ptr);` ← GPU page fault: data migrates to GPU

- Pages populate on **first touch**
- Pages migrate on-demand
- GPU memory over-subscription possible
- Concurrent access from CPU and GPU to memory (page-level)

JÜLICH Forschungszentrum

Performance Analysis

Comparing `scale_vector_um` (Unified Memory) and `scale_vector` (manual copy) for 20 480 `float` elements.

UM

```
Time(%)  Total Time (ns)  Name
-------  ---------------  ---------------------------------
  100.0          463,286  scale(float, float*, float*, int)
```

Manual

```
Time(%)  Total Time (ns)  Name
-------  ---------------  ---------------------------------
  100.0            4,792  scale(float, float*, float*, int)
```


JÜLICH Forschungszentrum

Performance Analysis

Comparing `scale_vector_um` (Unified Memory) and `scale_vector` (manual copy) for 20 480 `float` elements.

UM

```
Time(%)  Total Time (ns)  Name
-------  ---------------  ----------------------------------
  100.0          463,286  scale(float, float*, float*, int)
```

100× *slower?!*

What's going wrong here?

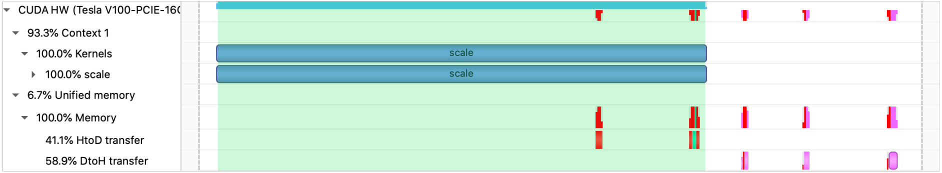
Manual

```
Time(%)  Total Time (ns)  Name
-------  ---------------  ----------------------------------
  100.0            4,792  scale(float, float*, float*, int)
```

Performance Analysis

Comparing `scale_vector_um` (Unified Memory) and `scale_vector` (manual copy) for 20 480 `float` elements.

UM



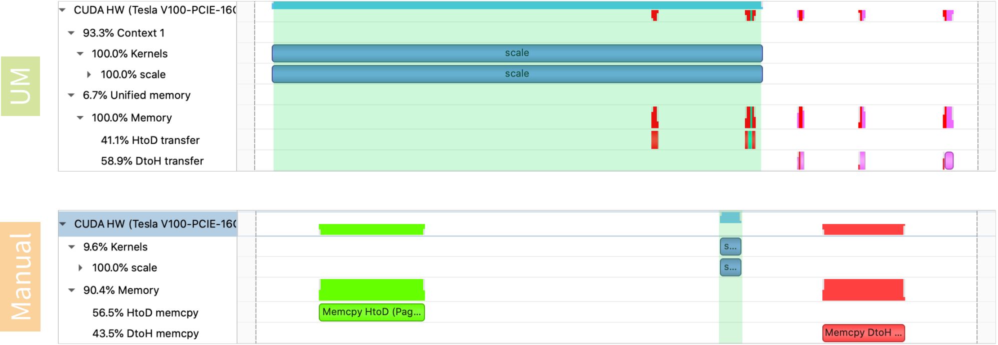
Manual

```
Time(%)  Total Time (ns)  Name
-------  ---------------  ---------------------------------
  100.0            4,792  scale(float, float*, float*, int)
```


JÜLICH Forschungszentrum

Performance Analysis

Comparing `scale_vector_um` (Unified Memory) and `scale_vector` (manual copy) for 20 480 `float` elements.



Comparing UM and Explicit Transfers

UM: Kernel is launched, data is needed by kernel, data migrates host$\rightarrow$device
$\Rightarrow$ Run time of kernel **incorporates** time for data transfers

Explicit: Data will be needed by kernel – data migrates host$\rightarrow$device **before** kernel launch
$\Rightarrow$ Run time of **kernel** without any transfers

Comparing UM and Explicit Transfers

UM Kernel is launched, data is needed by kernel, data migrates host→device
$\Rightarrow$ Run time of kernel **incorporates** time for data transfers

Explicit Data will be needed by kernel – data migrates host→device **before** kernel launch
$\Rightarrow$ Run time of **kernel** without any transfers

- UM more convenient
- Total run time of whole program does not principally change
  Except: Fault handling costs $\mathcal{O}$ (10 µs), stalls execution
- But data transfers sometimes sorted to kernel launch

Comparing UM and Explicit Transfers

UM Kernel is launched, data is needed by kernel, data migrates host$\rightarrow$device
$\Rightarrow$ Run time of kernel **incorporates** time for data transfers

Explicit Data will be needed by kernel – data migrates host$\rightarrow$device **before** kernel launch
$\Rightarrow$ Run time of **kernel** without any transfers

- UM more convenient
- Total run time of whole program does not principally change
  Except: Fault handling costs $\mathcal{O}(10\,\mu s)$*, stalls execution*
- But data transfers sometimes sorted to kernel launch

$\Rightarrow$ Improve UM behavior with performance hints!

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously
- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, `advice`:

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously
- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, `advice`:
  - `cudaMemAdviseSetReadMostly`: Read-only copy is kept

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously
- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, `advice`:
  - `cudaMemAdviseSetReadMostly`: Read-only copy is kept
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously
- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, `advice`:
  - `cudaMemAdviseSetReadMostly`: Read-only copy is kept
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping
  - `cudaMemAdviseSetAccessedBy`: Data is accessed by *this* device; will pre-map data to avoid page fault

Performance Hints for UM

New API routines

API calls to augment data location knowledge of runtime

- `cudaMemPrefetchAsync(data, length, device, stream)`
  Prefetches data to `device` (on `stream`) asynchronously
- `cudaMemAdvise(data, length, advice, device)`
  Advise about usage of given data, `advice`:
  - `cudaMemAdviseSetReadMostly`: Read-only copy is kept
  - `cudaMemAdviseSetPreferredLocation`: Set preferred location to avoid migrations; first access will establish mapping
  - `cudaMemAdviseSetAccessedBy`: Data is accessed by *this* device; will pre-map data to avoid page fault
- Use `cudaCpuDeviceId` for `device` CPU, or use `cudaGetDevice()` as usual to retrieve current GPU device id (default: 0)

Hints in Code

```
void sortfile(FILE *fp, int N) {
    char *data;
    // ...
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);


    cudaMemPrefetchAsync(data, N, device);
    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data); }
```

Hints in Code

```
void sortfile(FILE *fp, int N) {
    char *data;
    // ...
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);


    cudaMemPrefetchAsync(data, N, device);
    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data); }
```

Prefetch data to avoid expensive GPU page faults

Hints in Code

```
void sortfile(FILE *fp, int N) {
    char *data;
    // ...
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);

    cudaMemAdvise(data, N, cudaMemAdviseSetReadMostly, device);
    cudaMemPrefetchAsync(data, N, device);
    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data); }
```

Read-only copy of data is created on GPU during prefetch
→ CPU and GPU reads will not fault

Prefetch data to avoid expensive GPU page faults

Tuning `scale_vector_um`

TASK

Express data movement

- Location of code: `01-Basics/exercises/tasks/06-Scale-Vector-Hints/`
- Look at `Instructions.ipynb` for instructions
  1. Task: Advise CUDA runtime that data should be migrated to GPU before kernel call
  2. Build with `make`
  3. Run with `make run`
  4. Glimpse at profile with `make profile`
- See also CUDA C programming guide (L.3.) for details on data performance tunig

JÜLICH Forschungszentrum

System-Allocated Memory

- If supported by system (*Full CUDA Unified Memory Support*), `malloc()` (and `mmap`, and `new`, etc.) is unified
- Use performance hints, etc.
- Example

```
void sortfile(FILE *fp, int N) {
    char *data = (*data)malloc(sizeof(char) * N);

    fread(data, 1, N, fp);

    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    free(data); }
```

Conclusions

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- Thrust, OpenACC can give first entry point
- Full power with CUDA
- Threads, Blocks to expose parallelism for a kernel
- Several API routines exist
- Unified Memory productive, possibly with hints

Conclusions

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- Thrust, OpenACC can give first entry point
- Full power with CUDA
- Threads, Blocks to expose parallelism for a kernel
- Several API routines exist
- Unified Memory productive, possibly with hints

Thank you for your attention!

a.herten@fz-juelich.de

JÜLICH Forschungszentrum

Appendix

Appendix

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. 189

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, 95, 104, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 136, 137, 138, 143, 144, 145, 146, 147, 156, 181, 183, 184, 188

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 188

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, 53, 54, 55, 187, 188, 189, 190

NVLink NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 190

OpenACC Directive-based programming, primarily for many-core machines. 95, 97, 98, 99, 100, 101, 102, 183, 184

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, 95

Glossary III

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. 95, 97, 98, 99, 100

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 70, 109

Tesla The GPU product line for general purpose computing computing of NVIDIA. 12, 143, 144, 145, 146, 147

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 95, 104, 106, 183, 184

Glossary IV

V100 A large GPU with the Volta architecture from NVIDIA. It employs NVLink 2 as its interconnect and has fast *HBM2* memory. Additionally, it features *Tensorcores* for Deep Learning and Independent Thread Scheduling. 143, 144, 145, 146, 147

Volta GPU architecture from NVIDIA (announced 2017). 190

CPU Central Processing Unit. 12, 15, 21, 22, 23, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 52, 53, 54, 55, 70, 100, 106, 150, 151, 152, 160, 161, 162, 163, 164, 172, 173, 174, 175, 176, 177, 180, 187, 188

GPU Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43, 52, 53, 54, 55, 59, 60, 61, 69, 73, 74, 75, 76, 77, 78, 89, 96, 97, 98, 99, 100, 103, 106, 108, 119, 120, 121, 136, 137, 138, 143, 144, 145, 146, 147, 150, 151, 152, 153, 154, 155, 160, 161, 162, 163, 164, 172, 173, 174, 175, 176, 177, 179, 180, 181, 183, 184, 187, 188, 189, 190

JÜLICH Forschungszentrum

Glossary V

SIMD Single Instruction, Multiple Data. 52, 53, 54, 55

SIMT Single Instruction, Multiple Threads. 24, 25, 26, 39, 40, 42, 43, 52, 53, 54, 55

SM Streaming Multiprocessor. 52, 53, 54, 55

SMT Simultaneous Multithreading. 52, 53, 54, 55

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*. Nov. 2016. URL: `https://www.top500.org/lists/2016/11/` (pages 3–9).

References II

[5] Jack Dongarra et al. *Green500*. Nov. 2016. URL: `https://www.top500.org/green500/lists/2016/11/` (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 73–77).

References: Images, Graphics I

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

[7] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (pages 21, 22).

[8] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (pages 21, 22).

[9] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf.