



# GPU Programming

## HPC: Modern Architectures & Trends MA-INF 1106, Bonn

20 January 2023 | Dr. Andreas Herten | Accelerating Devices Lab, Forschungszentrum Jülich

# Outline

## Introduction

- GPU History

- JUPITER

- JUWELS

  - JUWELS Cluster

  - JUWELS Booster

## Platform

- Comparisons

- GPU Architecture

- Summary

## Programming GPUs

- Libraries

- Directives

- CUDA C/C++

  - Kernels

  - Grid, Blocks

  - Memory Management

  - Unified Memory

## Performance Analysis

## Beyond CUDA

  - Cooperative Groups

  - MPI

  - Thrust

  - Standard Parallelism

  - HIP

  - SYCL

  - MORE MODELS!!1

# History of GPUs

## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]

# History of GPUs

## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

# History of GPUs

## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA

# History of GPUs

## A short but unparalleled story

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



# History of GPUs

## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA
- 2009 OpenCL
- 2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]

# History of GPUs





## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA
- 2009 OpenCL
- 2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]
- 2022 : **Leonardo** (250 PFLOP/s\*, Italy), **NVIDIA** GPUs; **LUMI** (552 PFLOP/s, Finland), **AMD** GPUs  
: **Frontier** ( $R_{\max} = 1.102$  EFLOP/s, ORNL), **AMD** GPUs

\*: Effective FLOP/s, not theoretical peak

# History of GPUs

## A short but unparalleled story

- 1999 Graphics computation pipeline implemented in dedicated *graphics hardware*  
Computations using OpenGL graphics library [2]  
»GPU« coined by NVIDIA [3]
- 2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI
- 2007 CUDA
- 2009 OpenCL
- 2022 Top 500: 32 % with GPUs (#1, #2; 7 of top 10) [4], Green 500: 9 of top 10 with GPUs [5]
- 2022 : **Leonardo** (250 PFLOP/s\*, Italy), **NVIDIA** GPUs; **LUMI** (552 PFLOP/s, Finland), **AMD** GPUs  
: **Frontier** ( $R_{\max} = 1.102$  EFLOP/s, ORNL), **AMD** GPUs
- Soon : JUPITER ( $\approx 1$  EFLOP/s, **JSC**)  
: **Aurora** ( $\approx 2$  EFLOP/s, Argonne), **Intel** GPUs; **El Capitan** ( $\approx 2$  EFLOP/s, LLNL), **AMD** GPUs

\*: Effective FLOP/s, not theoretical peak

# JUPITER

## Recent Developments

- Jülich (JSC) selected as Hosting Entity for first European Exascale supercomputer: **JUPITER**
- System procured together with EuroHPC JU
- After long preparation, RFI finally published on Monday!
- Now we start the **exciting** negotiations...

# JUPITER

## Recent Developments

- Jülich (JSC) selected as Host for JUPITER
- System procured together with the Jülich Exascale supercomputer
- After long preparation, RFI for JUPITER is now open
- Now we start the exciting RFP phase



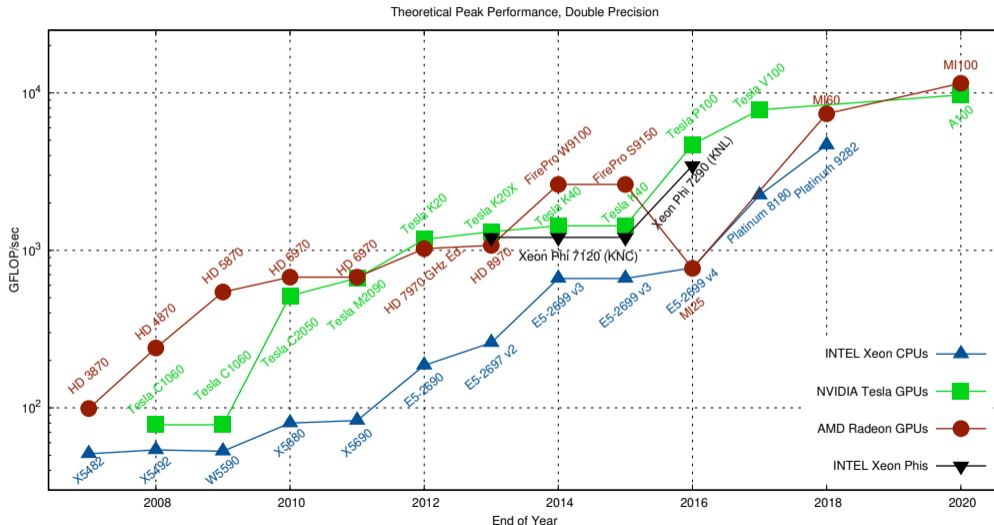
The JUPITER software components must enable a dynamic MSA architecture without software restrictions. Scheduler, resource manager and MPI runtime, among others, must enable advanced dynamic MSA features and deliver the features and benefits documented by the DEEP research projects.

For some control plane components (see above), JSC wishes to take advantage of its own xOPS software stack complemented with components that are a result of the collaborative work developed in the ParaStation consortium. Proposals for further (possibly module-specific) solutions that are optimized for the hardware offered will also be accepted if the above requirements can be realized in a superior or equal manner.

Considering the core capabilities of JUPITER's hardware and software, the centre stage is taken by accelerated devices: At the heart of JUPITER is the highly scalable **Booster** module, with the option of having two accelerated, tightly connected modules, built on the same accelerator technologies and networks, if that provides additional value. Given the technology developments in recent years, graphical processing unit (GPU) based accelerators are expected to provide at least one exaflop of double precision floating point performance as to the sustained HPL's  $R_{\max}$  within the aforementioned power footprint. Lower floating-point precision will reach even higher performance numbers. Those accelerators are able to handle both classical HPC and novel AI workloads and are well suited for HPDA. A high-bandwidth and low-latency interconnect for the Booster will be required to provide network capabilities for highly parallel workloads and, at the same time, fast access to the storage backends. Based on positive experience with the JUWELS Booster Dragonfly topology that provides 200 Gbit per GPU (or 800 Gbit per node) this is considered also an appropriate

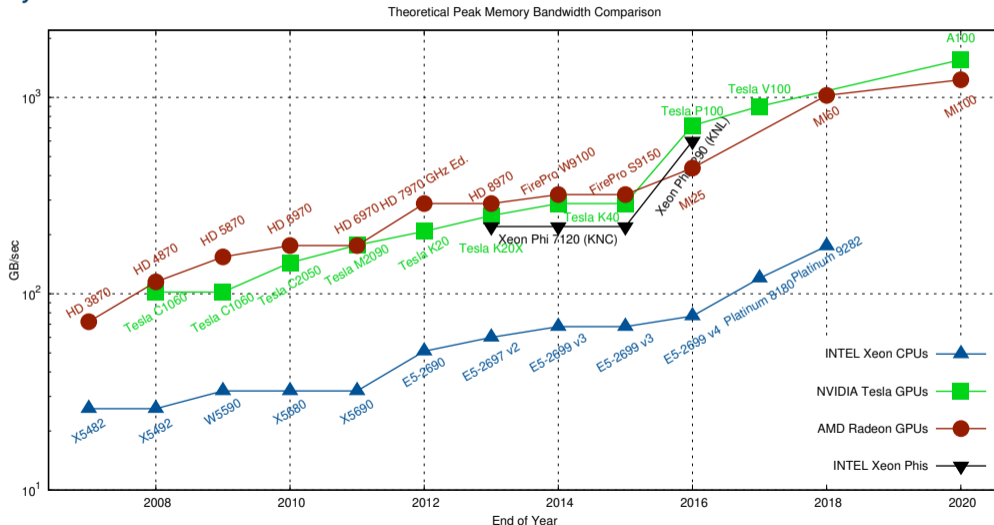
# Status Quo Across Architectures

## Performance



# Status Quo Across Architectures

## Memory Bandwidth





## JUWELS Cluster – Jülich's Scalable System

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



## JUWELS Booster – Scaling Higher!

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



### Top500 List Nov 2022:

- #1 Europe
- #8 World
- #4\* Top/Green500



### JUWELS Booster – Scaling Higher!

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

# Platform

# CPU vs. GPU

A matter of specialties



# CPU vs. GPU

A matter of specialties



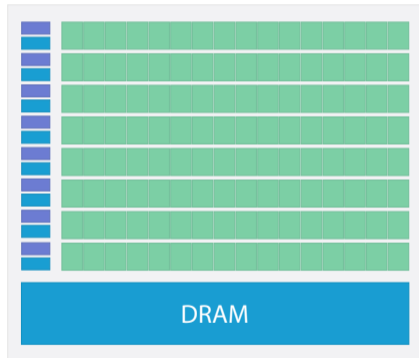
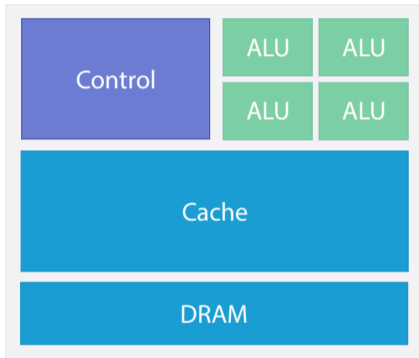
Transporting one



Transporting many

# CPU vs. GPU

## Chip

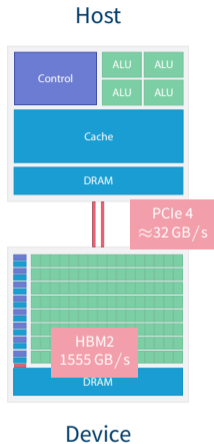


# GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)

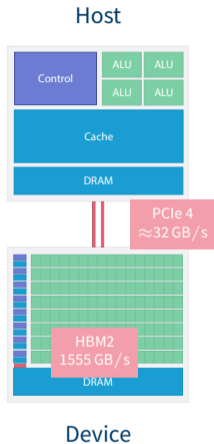


# GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)

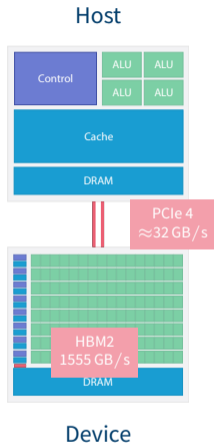


# GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually



# GPU Architecture Design

GPU optimized to **hide latency**

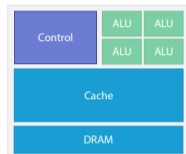
- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

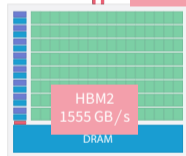
- Two engines: Overlap compute and copy



Host



PCIe 4  
≈ 32 GB/s



Device

# GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

- Two engines: Overlap compute and copy



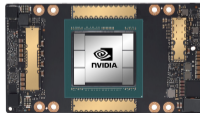
**V100**

32 GB RAM, 900 GB/s

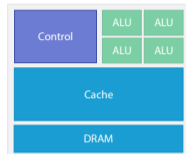


**A100**

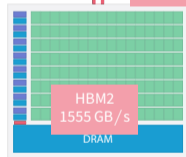
40 GB RAM, 1555 GB/s



Host



PCIe 4  
≈ 32 GB/s



Device

# GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

- Two engines: Overlap compute and copy



- SIMT

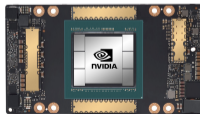
**V100**

32 GB RAM, 900 GB/s

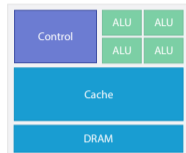


**A100**

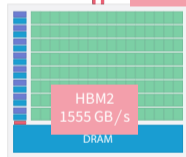
40 GB RAM, 1555 GB/s



Host



PCIe 4  
≈ 32 GB/s



Device

# SIMT

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

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

*Scalar*

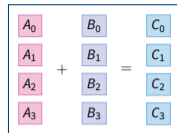
$A_0$	+	$B_0$	=	$C_0$
$A_1$	+	$B_1$	=	$C_1$
$A_2$	+	$B_2$	=	$C_2$
$A_3$	+	$B_3$	=	$C_3$

# SIMT

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

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

*Vector*

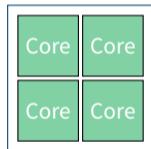
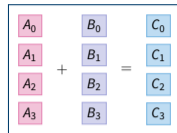


# SIMT

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

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

*Vector*

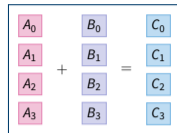


# SIMT

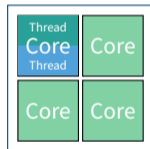
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

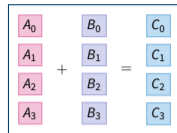


# SIMT

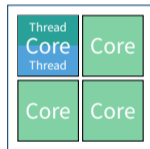
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

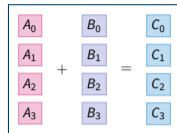


# SIMT

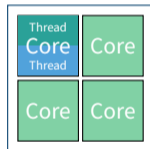
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

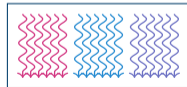
Vector



SMT




SIMT

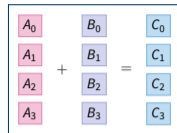


# SIMT

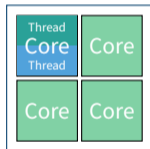
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

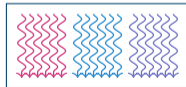
Vector



SMT



SIMT



# SIMT

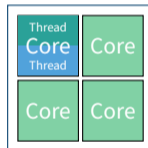
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



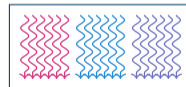
Vector

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

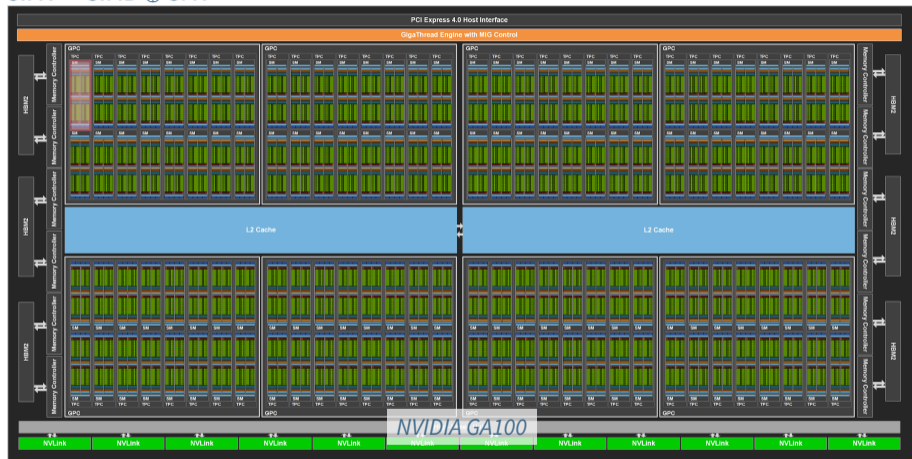
SMT



SIMT



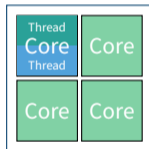
Graphics: Nvidia Corporation [11]

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


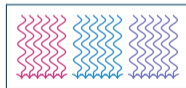
## Vector

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

*SMT*



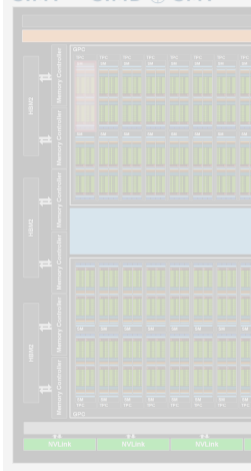
*SIMT*



Graphics: Nvidia Corporation [11]

# SIMT

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



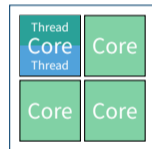
## Multiprocessor



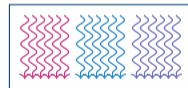
## Vector

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

## SMT



## SIMT

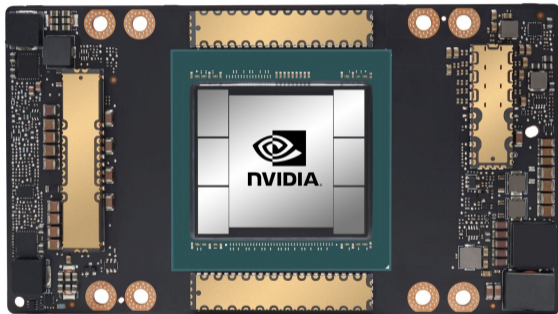


Graphics: Nvidia Corporation [11]

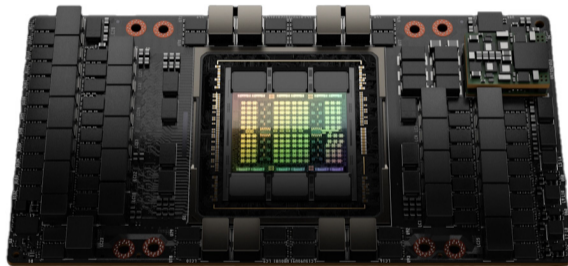
# A100 vs H100

Comparison of current vs. next generation

A100



H100



# A100 vs H100

Comparison of current vs. next generation

## A100



## H100



# A100 vs H100

## Comparison of current vs. next generation

### A100



### H100



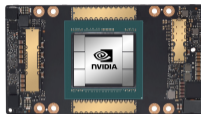
# CPU vs. GPU

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

# Preface: CPU

A simple CPU program!

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

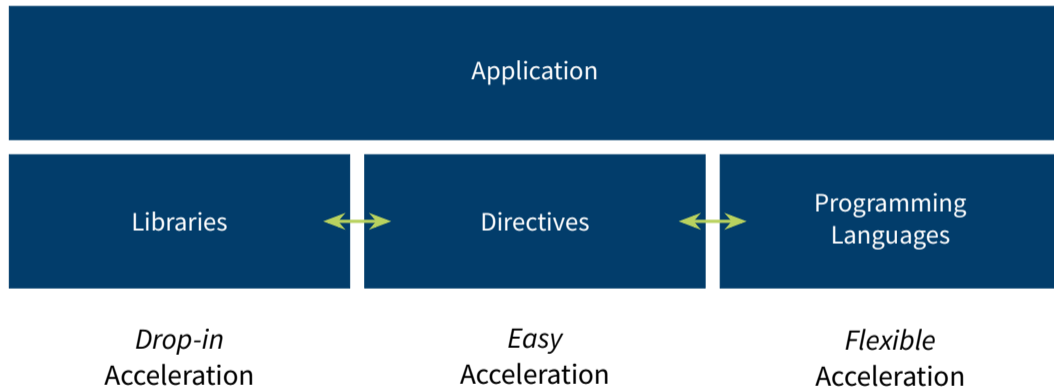
Part of LAPACK BLAS Level 1

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

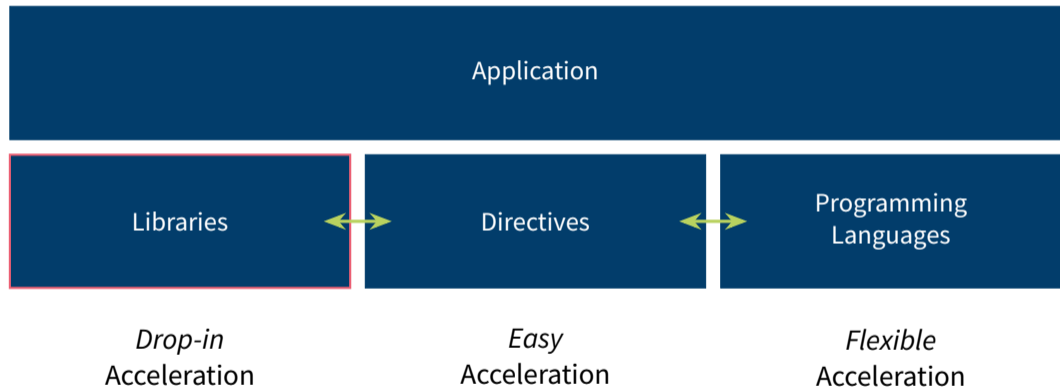
```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y
```

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

# Summary of Acceleration Possibilities



# Summary of Acceleration Possibilities



# Libraries

Programming GPUs is easy: Just don't!

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



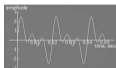
cuBLAS



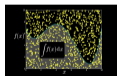
cuSPARSE



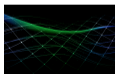
cuDNN



cuFFT



cuRAND



CUDA Math



{A} ARRAYFIRE

Numba

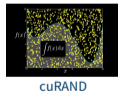
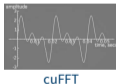


CuPy

# Libraries

Programming GPUs is easy: Just don't!

*Use applications & libraries*



Numba



CuPy



- 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(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);
```



# cuBLAS

## Code example

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

```
cublasHandle_t handle;  
cublasCreate(&handle);
```

Initialize

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

Allocate GPU memory

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

Call BLAS routine

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

Copy result to host

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

Finalize

# 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



# OpenACC / OpenMP

## 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];  
}  
  
float a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```

# OpenACC / OpenMP

## Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma omp target map(to:x[0:n]) map(tofrom:y[0:n]) loop  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
float a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```



# Programming GPUs Directly

Finally...

# Programming GPUs Directly

Finally...

**OpenCL** Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, ...) *2009*

- Platform: Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source

# Programming GPUs Directly

Finally...

**OpenCL** Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, ...) 2009

- Platform: Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source

**CUDA** NVIDIA's GPU platform 2007

- Platform: Drivers, programming language (CUDA C/C++), API, compiler, tools, ...
- Only NVIDIA GPUs
- Compilation with `nvcc` (free, but not open)
  - `clang` has CUDA support, but CUDA needed for last step
- Also: CUDA Fortran; and more in NVIDIA HPC SDK

# Programming GPUs Directly

Finally...

**OpenCL** Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, ...) *2009*

- Platform: Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source

**CUDA** NVIDIA's GPU platform *2007*

- Platform: Drivers, programming language (CUDA C/C++), API, compiler, tools, ...
- Only NVIDIA GPUs
- Compilation with `nvcc` (free, but not open)  
    `clang` has CUDA support, but CUDA needed for last step
- Also: CUDA Fortran; and more in NVIDIA HPC SDK

**HIP** AMD's unified programming model for AMD (via ROCm) and NVIDIA GPUs *2016+*

**SYCL** Intel's unified programming model for CPUs and GPUs (also: DPC++)

# Programming GPUs Directly

Finally...

**OpenCL** Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, ...) *2009*

- Platform: Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source

**CUDA** NVIDIA's GPU platform *2007*

- Platform: Drivers, programming language (CUDA C/C++), API, compiler, tools, ...
- Only NVIDIA GPUs
- Compilation with `nvcc` (free, but not open)  
    `clang` has CUDA support, but CUDA needed for last step
- Also: CUDA Fortran; and more in NVIDIA HPC SDK

**HIP** AMD's unified programming model for AMD (via ROCm) and NVIDIA GPUs *2016+*

**SYCL** Intel's unified programming model for CPUs and GPUs (also: DPC++)

- Choose what flavor you like, what colleagues/collaboration is using
- **Hardest: Come up with parallelized algorithm**



# Programming GPUs Directly

Finally...

**OpenCL** Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, ...) 2009

- Platform: Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source

**CUDA** NVIDIA's GPU platform 2007

- Platform: Drivers, programming language (CUDA C/C++), API, compiler, tools, ...
- Only NVIDIA GPUs
- Compilation with `nvcc` (free, but not open)  
    `clang` has CUDA support, but CUDA needed for last step
- Also: CUDA Fortran; and more in NVIDIA HPC SDK

**HIP** AMD's unified programming model for AMD (via ROCm) and NVIDIA GPUs 2016+

**SYCL** Intel's unified programming model for CPUs and GPUs (also: DPC++)

- Choose what flavor you like, what colleagues/collaboration is using
- **Hardest: Come up with parallelized algorithm**



# 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 → Block



# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Block



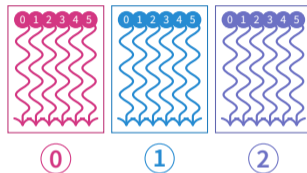
# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks



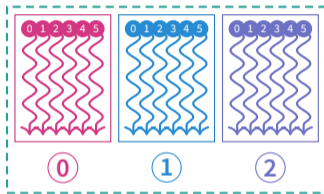
# CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid



# CUDA's Parallel Model

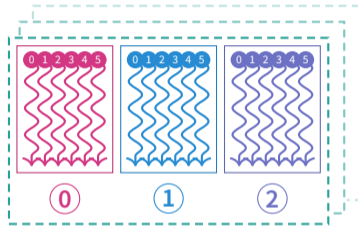
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



# CUDA's Parallel Model

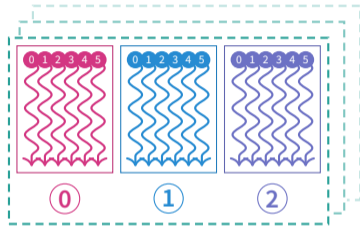
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



- Parallel function: **kernel**

- `__global__ kernel(int a, float * b) { }`

- Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...

- Execution entity: **threads**

- Lightweight → fast switching!

- 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 → 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**

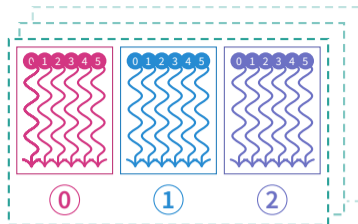
- Small GPU memory space; share data in block (high bandwidth)
- Shared memory: allocate statically (compile time) or dynamically (run time)
- `size_t shared`: bytes of shared memory allocated per block (in addition to static shared memory)

### `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

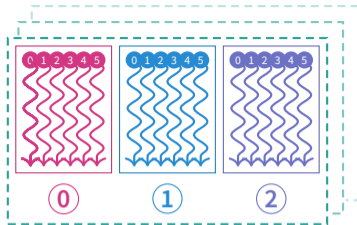
- Threads & blocks in 3D



# Grid Dimensions

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

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



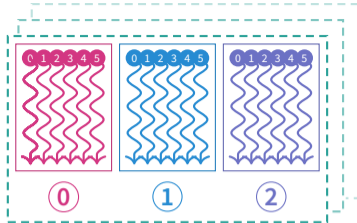
# Grid Dimensions

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

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

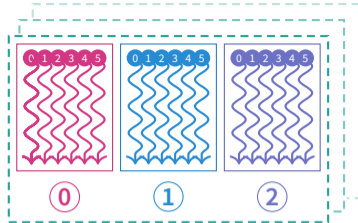
- Example:

```
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)
```

- 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

Block

- $\vec{N}_{\text{Thread}} \leq (1024_x, 1024_y, 64_z)$
- $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

# Grid Sizes

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

Block     ■  $\vec{N}_{\text{Thread}} \leq (1024_x, 1024_y, 64_z)$

            ■  $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

Grid       ■  $\vec{N}_{\text{Blocks}} \leq (2147483647_x, 65535_y, 65535_z) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

# Grid Sizes

- Block and grid sizes are hardware-dependent

- For JSC GPUs: Tesla V100, A100

Block     ■  $\vec{N}_{\text{Thread}} \leq (1024_x, 1024_y, 64_z)$

            ■  $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

Grid       ■  $\vec{N}_{\text{Blocks}} \leq (2147483647_x, 65535_y, 65535_z) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

- Find out yourself: `deviceQuery` example from CUDA Samples

# Grid Sizes

- Block and grid sizes are hardware-dependent

- For JSC GPUs: Tesla V100, A100

Block     ■  $\vec{N}_{\text{Thread}} \leq (1024_x, 1024_y, 64_z)$

           ■  $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

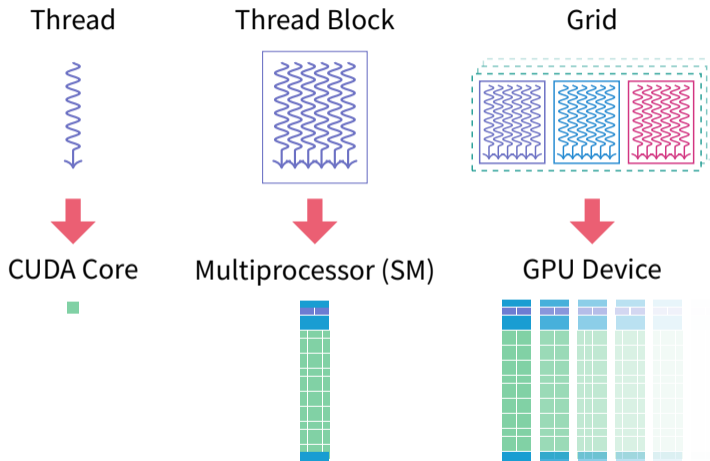
Grid       ■  $\vec{N}_{\text{Blocks}} \leq (2147483647_x, 65535_y, 65535_z) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

- 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



# 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));
```

# 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 ↔ 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 ↔ 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);
```

# 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)

# 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)

# 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?!*

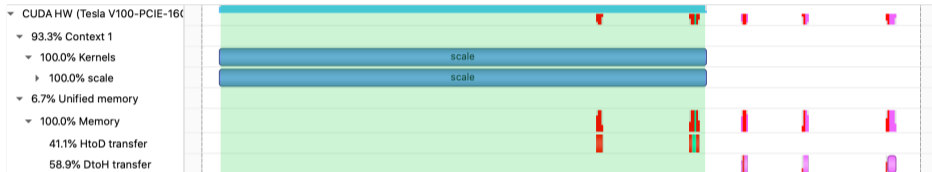
What's going wrong here?

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.

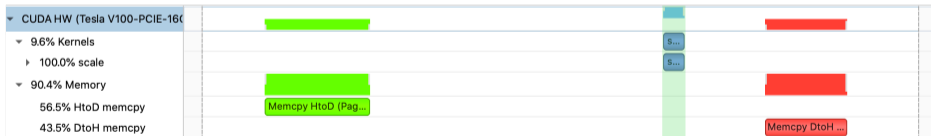
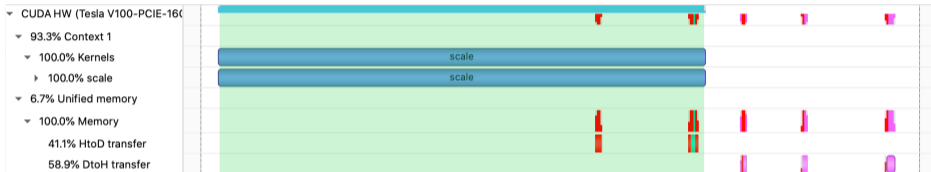


Manual

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.



# Comparing UM and Explicit Transfers

**UM** Kernel is launched, data is needed by kernel, data migrates host→device

⇒ Run time of kernel **incorporates** time for data transfers

**Explicit** Data will be needed by kernel – data migrates host→device **before** kernel launch

⇒ 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

⇒ Run time of kernel **incorporates** time for data transfers

**Explicit** Data will be needed by kernel – data migrates host→device **before** kernel launch

⇒ Run time of **kernel** without any transfers

- UM more convenient
- Total run time of whole program does not principally change  
*Except: Fault handling costs  $\mathcal{O}(10\ \mu\text{s})$ , 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→device

⇒ Run time of kernel **incorporates** time for data transfers

**Explicit** Data will be needed by kernel – data migrates host→device **before** kernel launch

⇒ Run time of **kernel** without any transfers

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

⇒ 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);  
  
    cudaMemcpyPrefetchAsync(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);  
  
    cudaMemcpyPrefetchAsync(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 ex-  
pensive GPU page faults

# Programming GPUs

## Performance Analysis

# GPU Tools

## The helpful helpers helping helpless (and others)

- NVIDIA

- `cuda-gdb` GDB-like command line utility for debugging

- `compute-sanitizer` Check memory accesses, race conditions, ...

- `Nsight` IDE for GPU developing, based on Eclipse (Linux, OS X) or Visual Studio (Windows) or VScode

- `Nsight Systems` GPU program profiler with timeline

- `Nsight Compute` GPU kernel profiler

- AMD

- `rocProf` Profiler for AMD's ROCm stack

- `uProf` Analyzer for AMD's CPUs and GPUs

# Nsight Systems

## CLI

```
$ nsys profile --stats=true ./poisson2d 10 # (shortened)
```

### CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
90.9	160,407,572	30	5,346,919.1	1,780	25,648,117	cuStreamSynchronize

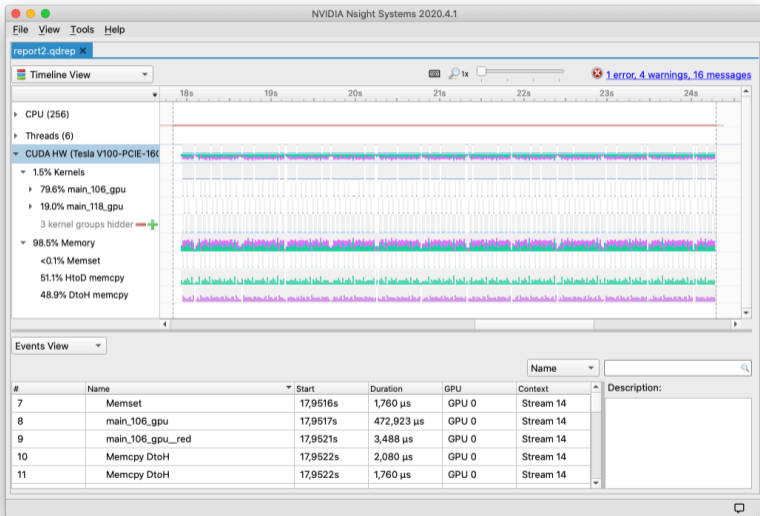
### CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	158,686,617	10	15,868,661.7	14,525,819	25,652,783	main_106_gpu
0.0	25,120	10	2,512.0	2,304	3,680	main_106_gpu__red

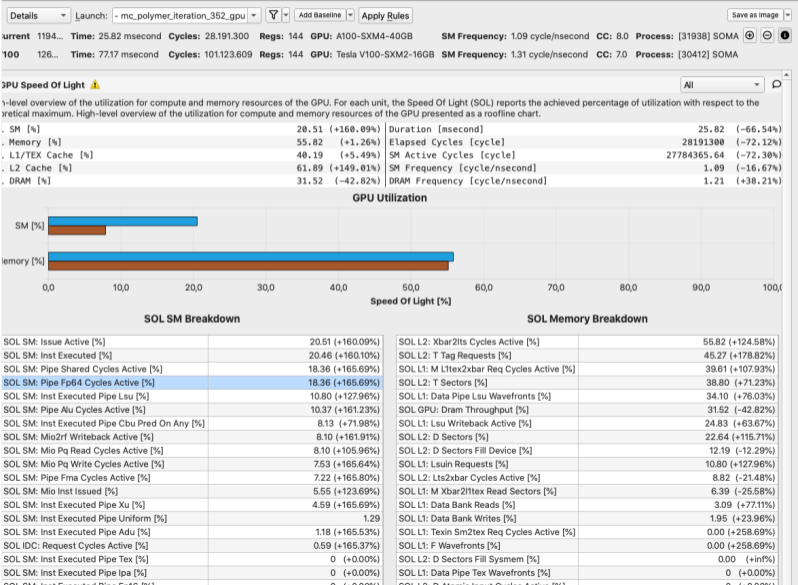


# Nsight Systems

## GUI



# Nsight Compute GUI



# Programming GPUs

## Beyond CUDA

# Programming GPUs

## Beyond CUDA: *Cooperative Groups*

# New Model: Cooperative Groups

- Motivation to extend classical model

**Algorithmic** Not all algorithms map easily to available synchronization methods;  
**synchronization** should be more flexible

**Design** Make groups of threads explicit **entities**

**Hardware** Access new **hardware features** (*Independent Thread Scheduling, Thread Block Clusters*)

→ **Cooperative Groups** (CG)

*A flexible model for synchronization and communication within groups of threads.*

# New Model: Cooperative Groups

- Motivation to extend classical model

**Algorithmic** Not all algorithms map easily to available synchronization methods;  
**synchronization** should be more flexible

**Design** Make groups of threads explicit **entities**

**Hardware** Access new **hardware features** (*Independent Thread Scheduling, Thread Block Clusters*)

## → Cooperative Groups (CG)

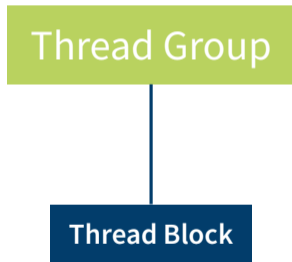
*A flexible model for synchronization and communication within groups of threads.*

- All in **namespace cooperative\_groups** (cooperative\_groups.h header)
- Following in text: `cooperative_groups::func()` → `cg::func()`  
**namespace cg = cooperative\_groups;**

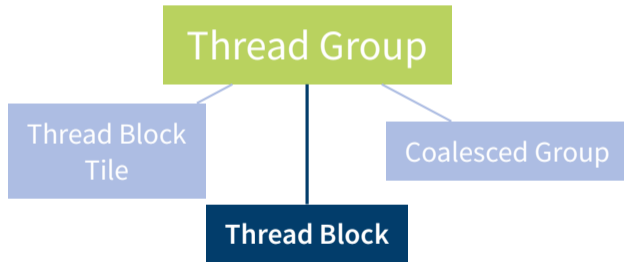
# Thread Group Landscape

Thread Group

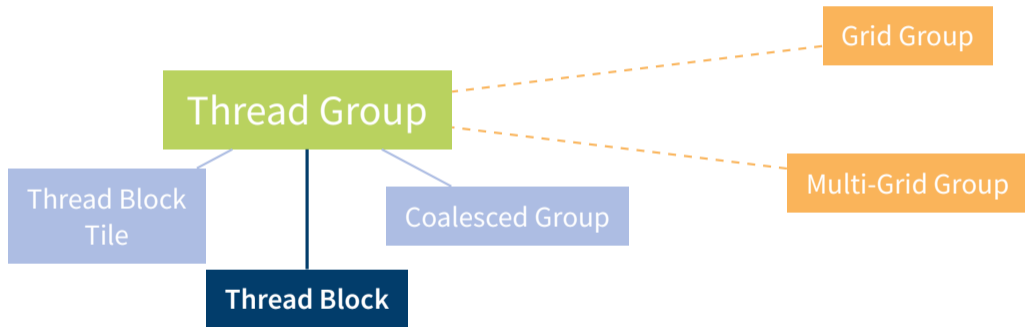
# Thread Group Landscape



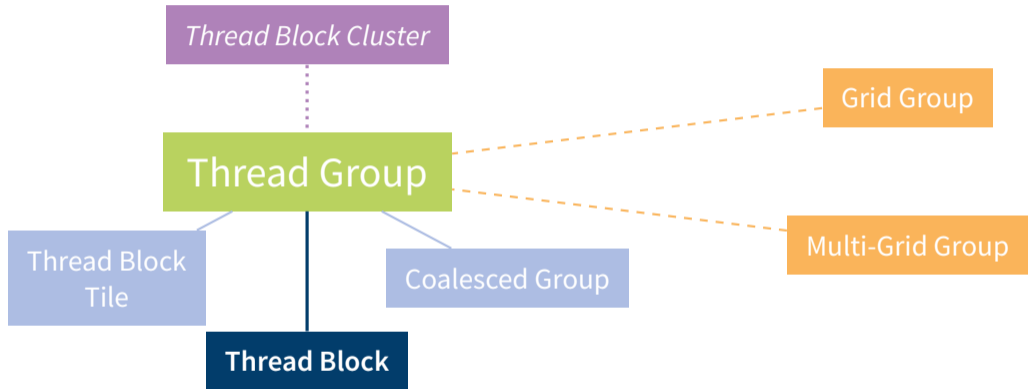
# Thread Group Landscape



# Thread Group Landscape



# Thread Group Landscape



# Common Methods of Cooperative Groups

- Fundamental type: `thread_group`
- Every CG has following member functions
  - `sync()` Synchronize the threads of this group (alternative `cg::sync(g)`)  
*Before: `__syncthreads()` for whole block*
  - `thread_rank()` Get unique ID of current thread in this group (*local index*)  
*Before: `threadIdx.x` for index in block*
  - `size()` Number of threads in this group  
*Before: `blockDim.x` for number of threads in block*
  - `is_valid()` *Group is technically ok*

# Simple Example: Print Rank

```
__device__ void printRank(cg::thread_group g) {  
    printf("Rank %d\n", g.thread_rank());  
}  
  
__global__ void allPrint() {  
    cg::thread_block b = cg::this_thread_block();  
  
    printRank(b);  
}  
  
int main() {  
    allPrint<<<1, 23>>>();  
}
```

# Advanced Example: Cooperative Reduce Collective

```
__shared__ int reduction_s[BLOCKSIZE];  
cg::thread_block cta = cg::this_thread_block();  
cg::thread_block_tile<32> tile = cg::tiled_partition<32>(cta);  
  
const int tid = cta.thread_rank();  
int value = A[tid];  
reduction_s[tid] = cg::reduce(tile, value, cg::plus<int>());  
// reduction_s contains tile-sum at all positions associated to tile  
cg::sync(cta);  
// Still to do: sum partial tile sums
```

# Programming GPUs

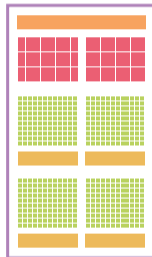
## Beyond CUDA: *MPI*

# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node

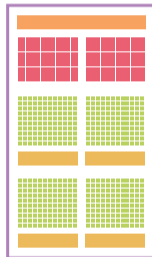
# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node



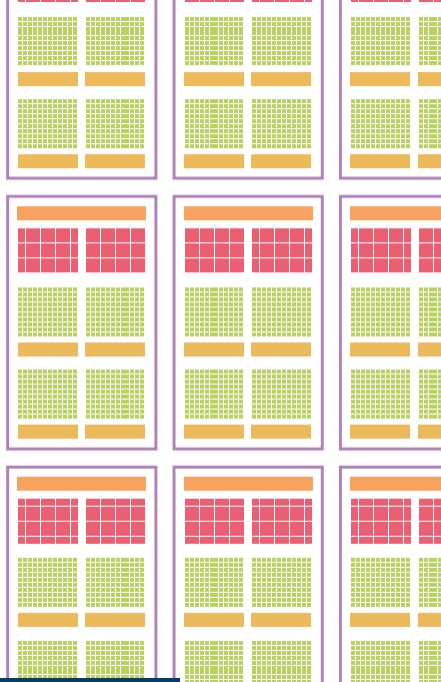
# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node
- HPC: multiple nodes



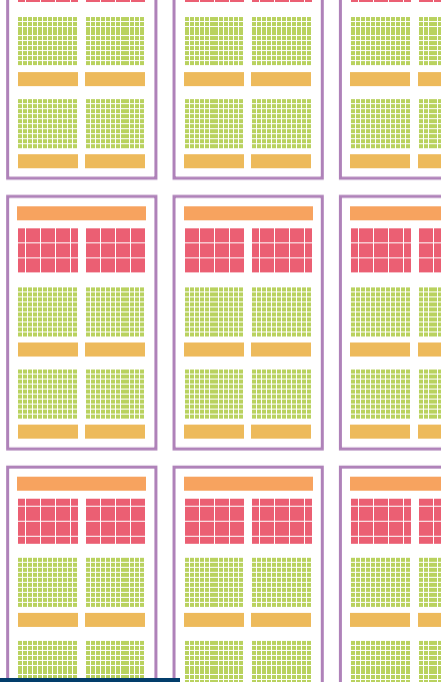
# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node
- HPC: multiple nodes



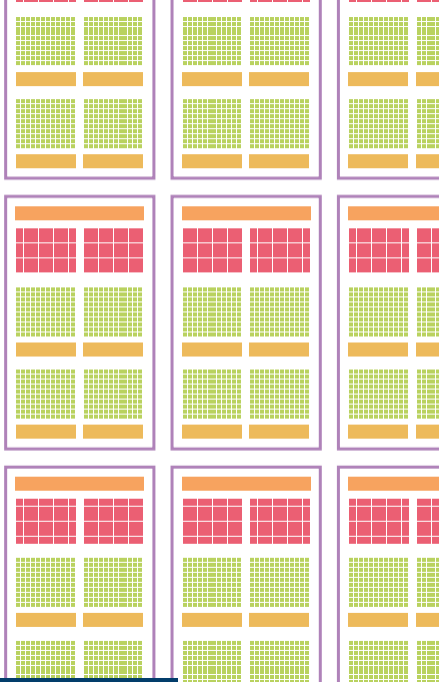
# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node
- HPC: multiple nodes
- Technology for distribution: MPI
- MPI also for multi-GPU computing!



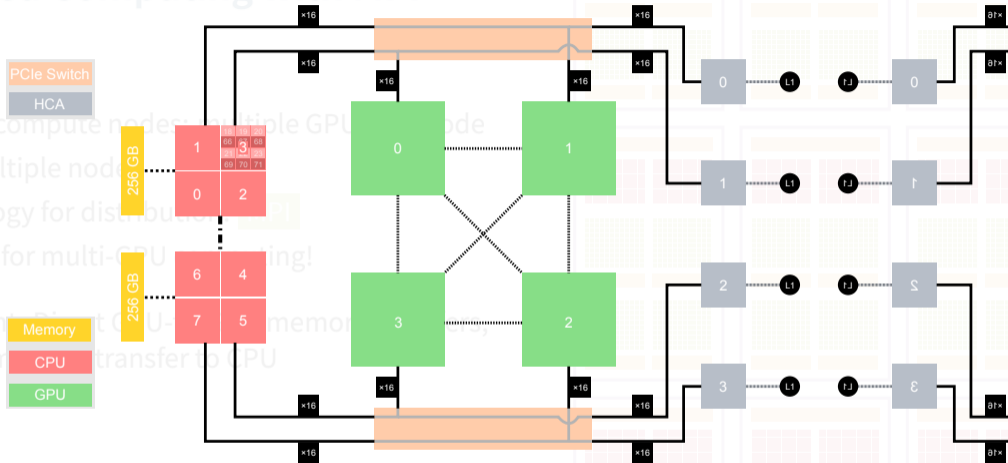
# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node
- HPC: multiple nodes
- Technology for distribution: MPI
- MPI also for multi-GPU computing!
- Important: Direct GPU-to-GPU memory transfers,  
no intermediate transfer to CPU



# Distributed Computing with MPI

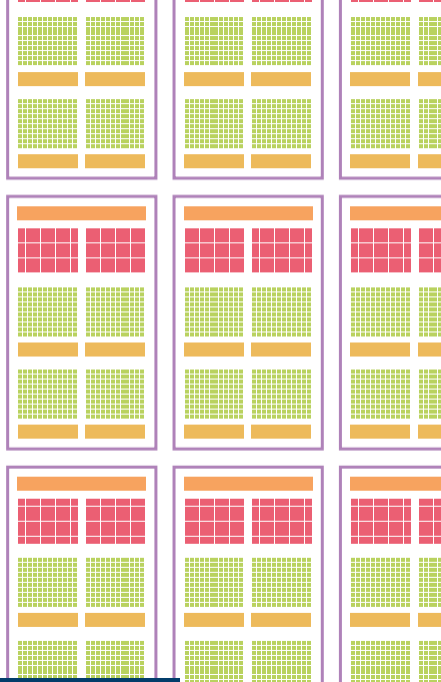
- Modern compute nodes: multiple GPU
- HPC: multiple nodes
- Technology for distributed computing
- MPI also for multi-GPU computing!
- Important: Direct CPU-to-GPU memory transfer to CPU



JUWELS Booster node topology

# Distributed Computing with MPI

- Modern compute nodes: multiple GPUs per node
- HPC: multiple nodes
- Technology for distribution: MPI
- MPI also for multi-GPU computing!
- Important: Direct GPU-to-GPU memory transfers,  
no intermediate transfer to CPU
- Modern MPIs can be GPU-aware and do the right thing



# MPI Sketch (*Pseudo-C*)

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank, size;
    // Init MPI
    MPI_Init(&argc, &argv);
    // Get current rank ID and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Call routines
    cudaMalloc(&buffer, n*sizeof(double));
    computeKernel<<<dim_grid,dim_block>>>(buffer);
    MPI_Sendrecv(buffer, n, MPI_REAL_TYPE, top, 0, buffer+n, n, MPI_REAL_TYPE, bottom, 0,
    ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // Shutdown
    MPI_Finalize();
    return 0;
}
```



# Programming GPUs

## Beyond CUDA: *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;
```

# Standard Parallelism

- By now, GPUs (and other accelerators) ubiquitous; around for long time
- Dedicated, custom parallelism concepts move into standards of languages
  - C++ Parallel STL since C++17 (2017)
  - Fortran do concurrent
- Both allow for execution on GPU
- Programmer identifies, exposes parallel code; compiler generates GPU-capable binary
- Compiler: NVHPC best, but also Intel oneDPL and others

# pSTL Standard Parallelism Example

```
int a = 42;
int n = 10;
std::vector x(N), y(N);
// fill x, y

std::transform(std::execution::par_unseq, x.begin(), x.end(), y.begin(), y.begin(),
    [=] (auto x, auto y) {
        return a*x+y;
    }
);
```

# Programming GPUs

Beyond CUDA: *HIP*

# Current GPU Leadership Systems Landscape

Picture by OLCF at ORNL on Flickr

- Current fastest supercomputer: **Frontier** at Oak Ridge (USA) with 38 000 AMD MI250X GPUs – 1.102 EFLOP/s; also most energy-efficient!



# Current GPU Leadership Systems Landscape

- Current fastest supercomputer: **Frontier** at Oak Ridge (USA) with 38 000 AMD MI250X GPUs – 1.102 EFLOP/s; also most energy-efficient!
- 2023: Aurora at Argonne with > 60 000 Intel Ponte Vecchio GPUs – > 2 EFLOP/s
- 2023: El Capitan at Lawrence Livermore with AMD MI300 GPUs – > 2 EFLOP/s



# Current GPU Leadership Systems Landscape

Picture by OLCF at ORNL on Flickr

- Current fastest supercomputer: **Frontier** at Oak Ridge (USA) with 38 000 AMD MI250X GPUs – 1.102 EFLOP/s; also most energy-efficient!
- 2023: Aurora at Argonne with > 60 000 Intel Ponte Vecchio GPUs – > 2 EFLOP/s
- 2023: El Capitan at Lawrence Livermore with AMD MI300 GPUs – > 2 EFLOP/s
- 2024: JUPITER at JSC – > 1 EFLOP/s! GPUs, details TBD



# Current GPU Leadership Systems Landscape

Picture by OLCF at ORNL on Flickr

- Current fastest supercomputer: **Frontier** at Oak Ridge (USA) with 38 000 **AMD** MI250X GPUs – 1.102 EFLOP/s; also most energy-efficient!
- 2023: Aurora at Argonne with > 60 000 **Intel** Ponte Vecchio GPUs – > 2 EFLOP/s
- 2023: El Capitan at Lawrence Livermore with **AMD** MI300 GPUs – > 2 EFLOP/s
- 2024: JUPITER at JSC – > 1 EFLOP/s! GPUs, details **TBD**



# AMD GPUs: HIP

- HIP: AMD's framework to utilize HPC GPUs
- *Heterogeneous Interface for Portability*
- Similar to CUDA, very similar `sed -i 's/cuda/hip/'`
- Can be compiled to run on NVIDIA GPUs (with CUDA) or AMD GPUs (ROCm)
- Includes C++ runtime API, kernel language; CUDA conversion tools
- Open Source
- Very similar performance on NVIDIA GPUs like CUDA

```
HIP_PLATFORM=amd hipcc --offload-arch=gfx90a -std=c++14 -o daxpy daxpy.cpp
```

# HIP SAXPY

```
#include <cuda.h>

__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();
```

# HIP SAXPY

```
#include "hip/hip_runtime.h"

__global__ void saxpy_hip (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
hipMallocManaged(&x, n * sizeof(float));
hipMallocManaged(&y, n * sizeof(float));

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

hipDeviceSynchronize();
```

# Programming GPUs

## Beyond CUDA: *SYCL*

- oneAPI: Intel's framework to utilize HPC GPUs and other parallel processors
- Large, open-source-ish ecosystem
  - **oneAPI**: Umbrella name for programming models and libraries, *open*; also a "specification"
  - **DPC++**: Data-Parallel C++; language built on C++ to target parallel devices, implements SYCL and prototypes extensions
  - **SYCL**: C++17-based model to target parallel devices, by Khronos group, open
  - **Intel oneAPI DPC++/C++ Compiler**: New LLVM-based Intel compiler to compile DPC++
  - **oneMKL, oneDNN, ...**: Specific libraries for domains, some open
  - **oneAPI DPC++ Library (*oneDPL*)**: DPC++-accompanying library with algorithms etc.
- Programming with iterators, lambdas, queues, views
- Since OSS: Not only for Intel GPUs but also AMD, NVIDIA backends
- Higher Level: Might even give better performance than legacy CUDA

→ [github.com/oneapi-src](https://github.com/oneapi-src)

# DPC++ Example

```
int a = 42;
int n = 10;
std::vector x(N), y(N);
// fill x, y
{
    sycl::queue q(sycl::gpu_selector{});
    sycl::buffer<float, 1> d_x { x.data(), sycl::range<1>(x.size())}, d_y...;
    q.submit([&] (sycl::handler& h) {
        auto x_access = d_x.get_access<sycl::access::mode::read> (h);
        auto y_access = d_y.get_access<sycl::access::mode::read_write> (h);
        h.parallel_for<class xpy>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            auto i = it.get_id(0);
            y_access[i] += a * x_access[i] + y_access[i];
        });
    });
}
```

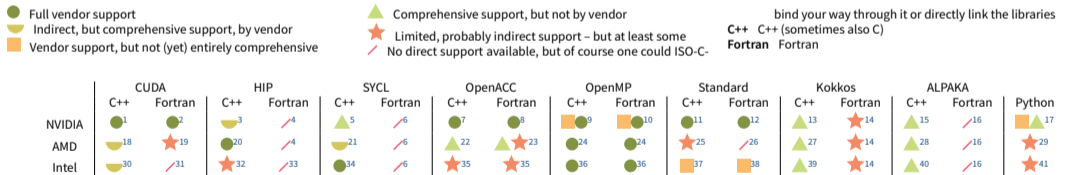
Adapted from [github.com/jeffhammond/dpcpp-tutorial](https://github.com/jeffhammond/dpcpp-tutorial)

# Programming GPUs

Beyond CUDA: *MORE MODELS!!1*

# State-of-the-Art GPU Programming Models

- GPU programming not only *programming with CUDA* anymore
- Much more, and CUDA only one solution
- New GPU vendors in the game now *hungry for a piece of the cake*
- Many models, most offer translation from CUDA



See [appendix](#) for details or [doi:10.34732/xdvblg-r1bvif](https://doi.org/10.34732/xdvblg-r1bvif)

# Conclusions

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- OpenACC can give first entry point
- Full power with CUDA
- Threads, Blocks to expose parallelism for a kernel
- Several API routines exist
- Cooperative Groups: new entry point
- Beyond CUDA: Thrust, pSTL, HIP, SYCL, Kokkos, ...

# Conclusions

- GPUs achieve performance by specialized hardware
- Acceleration can be done by different means
- Libraries are the easiest
- OpenACC can give first entry point
- Full power with CUDA
- Threads, Blocks to expose parallelism for a kernel
- Several API routines exist
- Cooperative Groups: new entry point
- Beyond CUDA: Thrust, pSTL, HIP, SYCL, Kokkos, ...

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

# Appendix

## Appendix

GPU Model/Vendor Compatibility Table

References

Glossary

# Appendix

## GPU Model/Vendor Compatibility Table

# GPU Programming Models: Table

- Full vendor support
- ◐ Indirect, but comprehensive support, by vendor
- ◑ Vendor support, but not (yet) entirely comprehensive

- ▲ Comprehensive support, but not by vendor
- ★ Limited, probably indirect support – but at least some
- / No direct support available, but

of course one could ISO-C-bind your way through it or directly link the libraries

C++ C++ (sometimes also C)  
Fortran Fortran

	CUDA		HIP		SYCL		OpenACC		OpenMP	
	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran
NVIDIA	●1	●2	◐3	/4	▲5	/6	●7	●8	◑9	●10
AMD	◐11	★12	●13	/4	◐14	/6	▲15	▲★16	●17	●17
Intel	◐18	/19	★20	/21	●22	/6	★23	★23	●24	●24
	Standard		Kokkos		ALPAKA		Python			
	C++	Fortran	C++	Fortran	C++	Fortran				
NVIDIA	●25	●26	▲27	★28	▲29	/30	◑▲31			
AMD	★32	/33	▲34	★28	▲35	/30	★36			
Intel	◑37	◑38	▲39	★28	▲40	/30	★41			

# GPU Programming Models: Footnotes I

- 1: CUDA C/C++ is supported on NVIDIA GPUs through the [CUDA Toolkit](#)
- 2: CUDA Fortran, a proprietary Fortran extension, is supported on NVIDIA GPUs via the [NVIDIA HPC SDK](#)
- 3: [HIP](#) programs can directly use NVIDIA GPUs via a CUDA backend; HIP is maintained by AMD
- 4: No such thing like HIP for Fortran, but AMD offers Fortran interfaces to HIP and ROCm libraries in [hipfort](#)
- 5: SYCL can be used on NVIDIA GPUs with *experimental* support either in [SYCL](#) directly or in [DPC++](#), or via [hipSYCL](#)
- 6: No such thing like SYCL for Fortran
- 7: OpenACC C/C++ supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by [GCC C compiler](#) and in LLVM through [Clacc](#)
- 8: OpenACC Fortran supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC Fortran compiler and [Flacc](#)
- 9: OpenMP in C++ supported on NVIDIA GPUs through NVIDIA HPC SDK (albeit [with a few limits](#)), by GCC, and Clang; see [OpenMP ECP BoF on status in 2022](#).
- 10: OpenMP in Fortran supported on NVIDIA GPUs through NVIDIA HPC SDK (but not full OpenMP feature set available), by GCC, and Flang
- 25: pSTL features supported on NVIDIA GPUs through [NVIDIA HPC SDK](#)
- 26: Standard Language parallel features supported on NVIDIA GPUs through NVIDIA HPC SDK
- 27: [Kokkos](#) supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 28: Kokkos is a C++ model, but an official compatibility layer ([Fortran Language Compatibility Layer, FLCL](#)) is available.

# GPU Programming Models: Footnotes II

- 29: [Alpaka](#) supports NVIDIA GPUs by calling CUDA as part of the compilation process; also, an OpenMP backend can be used
- 30: Alpaka is a C++ model
- 31: There is a vast community of offloading Python code to NVIDIA GPUs, like [CuPy](#), [Numba](#), [cuNumeric](#), and many others; NVIDIA actively supports a lot of them, but has no direct product like *CUDA for Python*; so, the status is somewhere in between
- 11: [hipify](#) by AMD can translate CUDA calls to HIP calls which runs natively on AMD GPUs
- 12: AMD offers a Source-to-Source translator to convert some CUDA Fortran functionality to OpenMP for AMD GPUs ([gpufort](#)); in addition, there are ROCm library bindings for Fortran in [hipfort](#) OpenACC/CUDA Fortran Source-to-Source translator
- 13: [HIP](#) is the preferred native programming model for AMD GPUs
- 14: SYCL can use AMD GPUs, for example with [hipSYCL](#) or [DPC++ for HIP AMD](#)
- 15: OpenACC C/C++ can be used on AMD GPUs via GCC or Clacc; also, [Intel's OpenACC to OpenMP Source-to-Source translator](#) can be used to generate OpenMP directives from OpenACC directives
- 16: OpenACC Fortran can be used on AMD GPUs via GCC; also, AMD's [gpufort](#) Source-to-Source translator can move OpenACC Fortran code to OpenMP Fortran code, and also Intel's translator can work
- 17: AMD offers a dedicated, Clang-based compiler for using OpenMP on AMD GPUs: [AOMP](#); it supports both C/C++ (Clang) and Fortran (Flang, [example](#))

# GPU Programming Models: Footnotes III

- 32: Intel's DPC++ (oneAPI) can be [compiled with an experimental HIP AMD backend](#), allowing to launch STL algorithms to AMD GPUs; caveats from Intel's STL support apply
- 33: Currently, no (known) way to launch Standard-based parallel algorithms on AMD GPUs
- 34: Kokkos supports AMD GPUs through HIP
- 35: Alpaka supports AMD GPUs through HIP or through an OpenMP backend
- 36: AMD does not officially support GPU programming with Python (also not semi-officially like NVIDIA), but third-party support is available, for example through [Numba](#) (currently inactive) or a [HIP version of CuPy](#)
- 18: [SYCLomatic](#) translates CUDA code to SYCL code, allowing it to run on Intel GPUs; also, Intel's [DPC++ Compatibility Tool](#) can transform CUDA to SYCL
- 19: No direct support, only via ISO C bindings, but at least an example can be [found on GitHub](#); it's pretty scarce and not by Intel itself, though
- 20: [CHIP-SPV](#) supports mapping CUDA and HIP to OpenCL and Intel's Level Zero, making it run on Intel GPUs
- 21: No such thing like HIP for Fortran
- 22: [SYCL](#) is the prime programming model for Intel GPUs; actually, SYCL is only a standard, while Intel's implementation of it is called [DPC++](#) (*Data Parallel C++*), which extends the SYCL standard in various places; actually actually, Intel namespaces everything *oneAPI* these days, so the *full* proper name is Intel oneAPI DPC++ (which incorporates a C++ compiler and also a library)
- 23: OpenACC can be used on Intel GPUs by translating the code to OpenMP with [Intel's Source-to-Source translator](#)

# GPU Programming Models: Footnotes IV

- 24: Intel has [extensive support for OpenMP](#) through their latest compilers
- 37: Intel supports pSTL algorithms through their [DPC++ Library](#) (oneDPL; [GitHub](#)). It's heavily namespaced and not yet on the same level as NVIDIA
- 38: With [Intel oneAPI 2022.3](#), Intel supports DO CONCURRENT with GPU offloading
- 39: Kokkos supports Intel GPUs through SYCL
- 40: [Alpaka v0.9.0](#) introduces experimental SYCL support; also, Alpaka can use OpenMP backends
- 41: Not a lot of support available at the moment, but notably [DPNP](#), a SYCL-based drop-in replacement for Numpy, and [numba-dpex](#), an extension of Numba for DPC++.

# Appendix

## References

# 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 13, 14).
- [13] Wes Breazell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/> (pages 46–50).

# References: Images, Graphics I

- [1] Héctor J. Rivas. *Color Reels*. Freely available at Unsplash. URL: <https://unsplash.com/photos/87hFrPk3V-s>.
- [7] Forschungszentrum Jülich GmbH (Ralf-Uwe Limbach). *JUWELS Booster*.
- [8] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (pages 19, 20).
- [9] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (pages 19, 20).
- [10] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: <https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf>.

# References: Images, Graphics II

- [11] Nvidia Corporation. *Pictures: Ampere GPU*. Ampere Architecture Whitepaper. URL: <http://www.nvidia.com/nvidia-ampere-architecture-whitepaper> (pages 35–37).
- [12] Nvidia Corporation. *Pictures: Hopper GPU*. Nvidia Developer Technical Blog: NVIDIA Hopper Architecture In-Depth. URL: <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>.
- [14] OLCF at ORNL. *Picture: Frontier*. Flickr. URL: <https://www.flickr.com/photos/olcf/52117623843/>.

# Appendix

## Glossary

# Glossary I

**AMD** Manufacturer of CPUs and GPUs. 3, 4, 5, 6, 7, 8, 9, 60, 61, 62, 63, 64, 65, 198, 199

**Ampere** GPU architecture from NVIDIA (announced 2019). 16, 17

**API** A programmatic interface to software by well-defined functions. Short for application programming interface. 60, 61, 62, 63, 64, 65, 199

**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, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 94, 95, 96, 101, 102, 103, 104, 105, 164, 181, 182, 199

# Glossary II

**HIP** GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 60, 61, 62, 63, 64, 65

**JUWELS** Jülich's new supercomputer, the successor of JUQUEEN. 15, 16, 17

**NVIDIA** US technology company creating GPUs. 3, 4, 5, 6, 7, 8, 9, 15, 16, 17, 35, 36, 37, 60, 61, 62, 63, 64, 65, 137, 197, 198, 200

**NVLink** NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 200

**OpenACC** Directive-based programming, primarily for many-core machines. 55, 56, 57, 58, 59, 181, 182

# Glossary III

- 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, 60, 61, 62, 63, 64, 65
- 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. 55, 56, 57, 58, 59
- ROCm** AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 60, 61, 62, 63, 64, 65
- SAXPY** Single-precision  $A \times X + Y$ . A simple code example of scaling a vector and adding an offset. 43, 67, 174, 175

# Glossary IV

- Tesla** The GPU product line for general purpose computing computing of NVIDIA. 15, 101, 102, 103, 104, 105
- Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 164
- 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. 101, 102, 103, 104, 105
- Volta** GPU architecture from NVIDIA (announced 2017). 200
- CG** Cooperative Groups. 143, 144, 150

# Glossary V

**CPU** Central Processing Unit. 15, 19, 20, 21, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 43, 60, 61, 62, 63, 64, 65, 107, 108, 109, 115, 116, 117, 118, 119, 127, 128, 129, 130, 131, 132, 135, 197, 198, 199

**GPU** Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 9, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 42, 46, 47, 48, 49, 50, 51, 54, 55, 56, 57, 60, 61, 62, 63, 64, 65, 66, 77, 78, 79, 94, 95, 96, 101, 102, 103, 104, 105, 107, 108, 109, 110, 111, 112, 115, 116, 117, 118, 119, 127, 128, 129, 130, 131, 132, 134, 135, 136, 137, 141, 142, 153, 163, 168, 176, 179, 181, 182, 197, 198, 199, 200

**SIMD** Single Instruction, Multiple Data. 28, 29, 30, 31, 32, 33, 34, 35, 36, 37

**SIMT** Single Instruction, Multiple Threads. 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37

# Glossary VI

**SM** Streaming Multiprocessor. [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#)

**SMT** Simultaneous Multithreading. [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#)