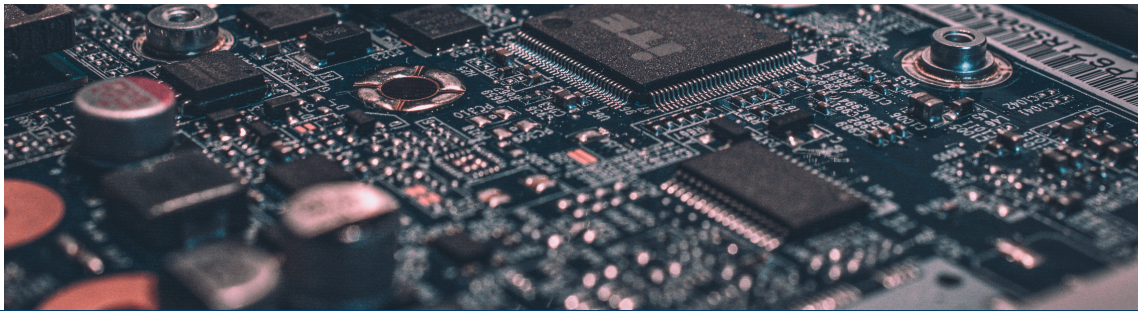




GPU ACCELERATORS AT JSC SUPERCOMPUTING INTRODUCTION COURSE

20 May 2021 | Andreas Herten | Forschungszentrum Jülich

Outline

GPUs at JSC

JUWELS

JUWELS Cluster

JUWELS Booster

JURECA DC

GPU Architecture

Empirical Motivation

Comparisons

GPU Architecture

Summary

Programming GPUs

Libraries

Directives

CUDA C/C++

Performance Analysis

Advanced Topics

Advanced Topics



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: #44)



JUWELS Booster – Scaling Higher!

- 936 nodes with AMD EPYC Rome CPUs (2×24 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; 4×200 Gbit/s per node



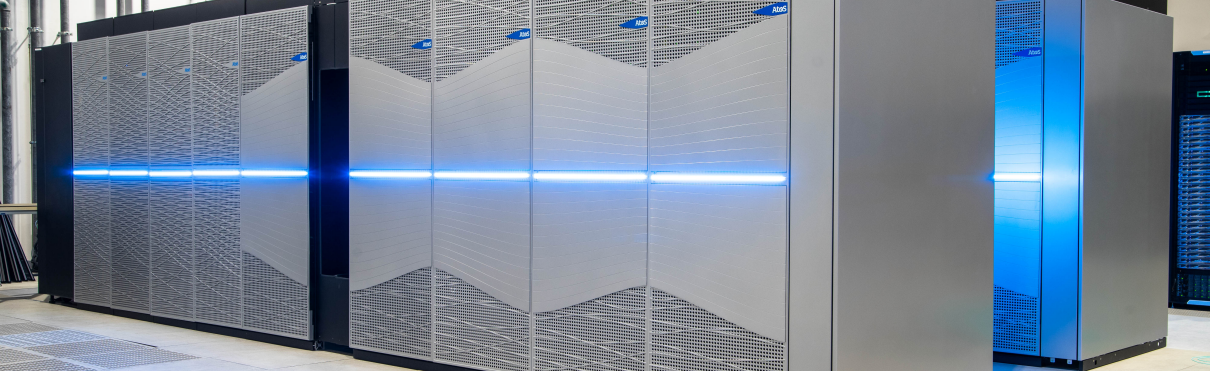
Top500 List Nov 2020:

- #1 Europe
- #7 World
- #3* Green500



JUWELS Booster – Scaling Higher!

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



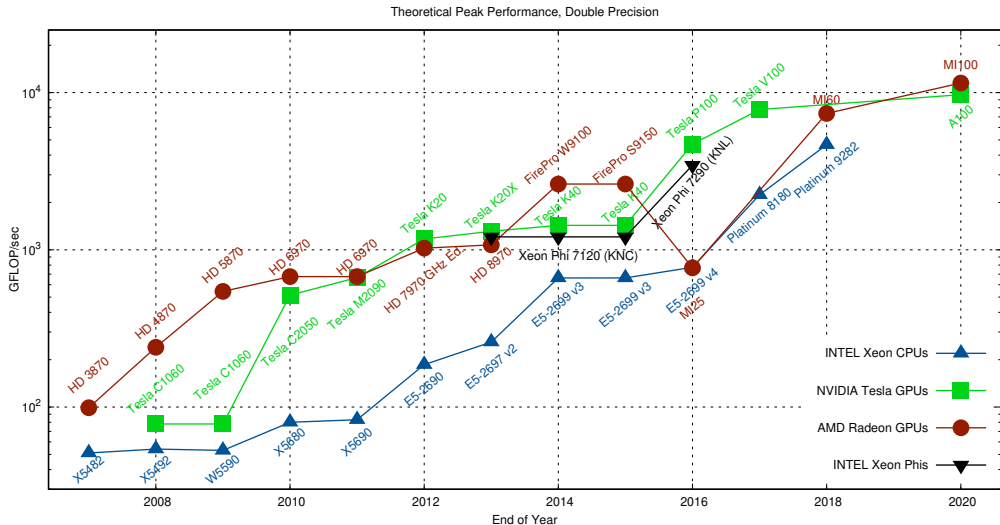
JURECA DC – Multi-Purpose

- 768 nodes with AMD EPYC Rome CPUs (2×64 cores)
- 192 nodes with 4 NVIDIA A100 Ampere GPUs
- InfiniBand DragonFly+ HDR-100 network
- Also: JURECA Booster: 1640 nodes with Intel Xeon Phi *Knights Landing*

GPU Architecture

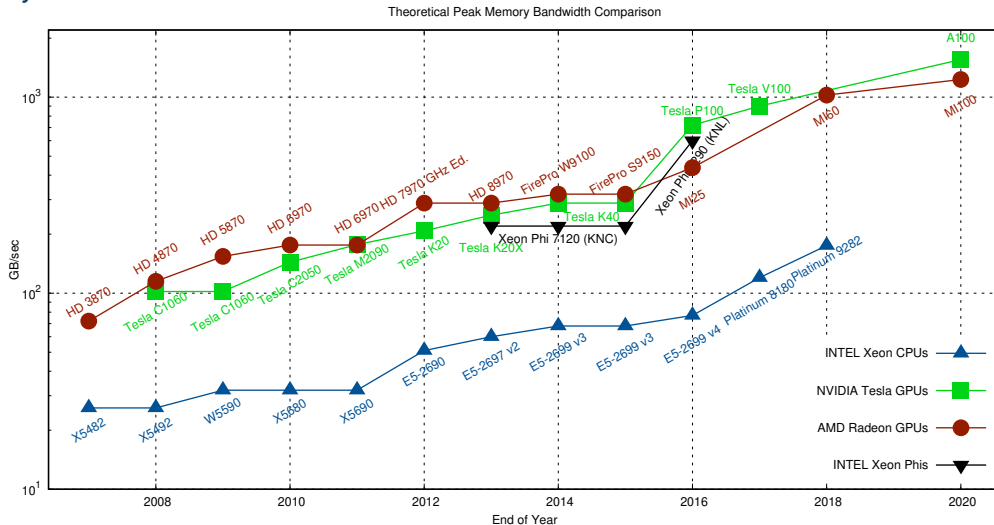
Status Quo Across Architectures

Performance



Status Quo Across Architectures

Memory Bandwidth



CPU vs. GPU

A matter of specialties



CPU vs. GPU

A matter of specialties



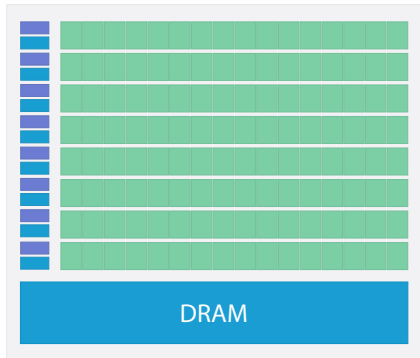
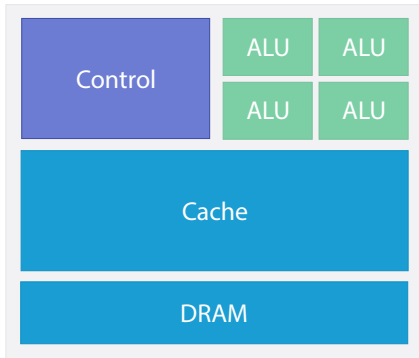
Transporting one



Transporting many

CPU vs. GPU

Chip

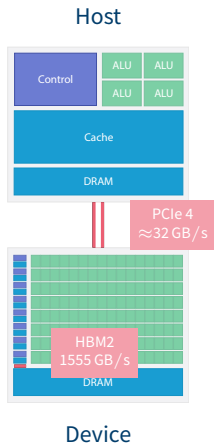


GPU Architecture Design

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)

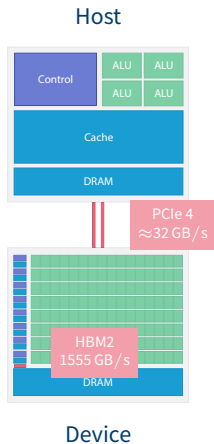


GPU Architecture Design

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)

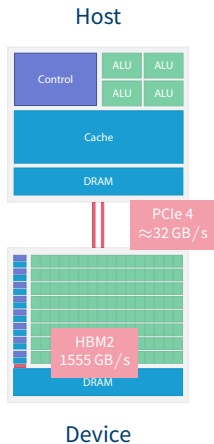


GPU Architecture Design

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**

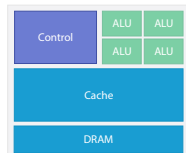
- Memory

- 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

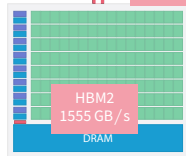
- Two engines: Overlap compute and copy



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



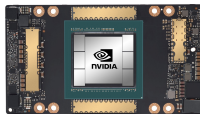
V100

32 GB RAM, 900 GB/s

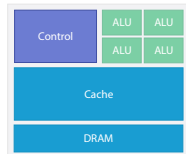


A100

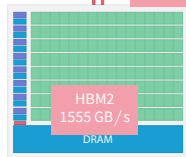
40 GB RAM, 1555 GB/s



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

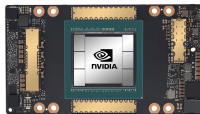
V100

32 GB RAM, 900 GB/s

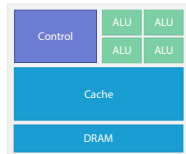


A100

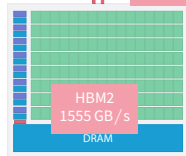
40 GB RAM, 1555 GB/s



Host



PCIe 4
≈ 32 GB/s



Device

SIMT

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

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

Scalar

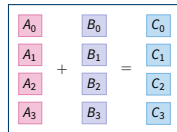
A_0	+	B_0	=	C_0
A_1	+	B_1	=	C_1
A_2	+	B_2	=	C_2
A_3	+	B_3	=	C_3

SIMT

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

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

Vector

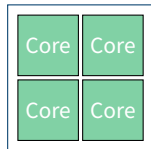
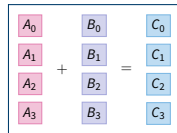


SIMT

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

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

Vector

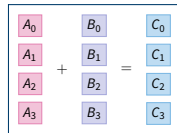


SIMT

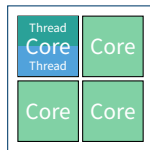
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

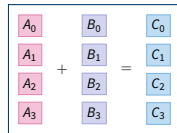


SIMT

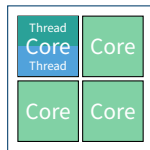
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

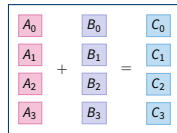


SIMT

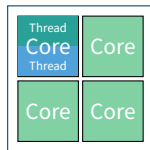
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

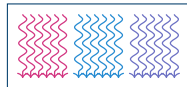
Vector



SMT



SIMT

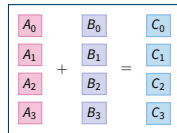


SIMT

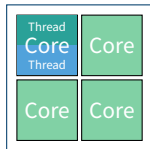
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

- 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 

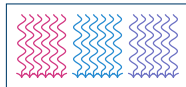
Vector



SMT

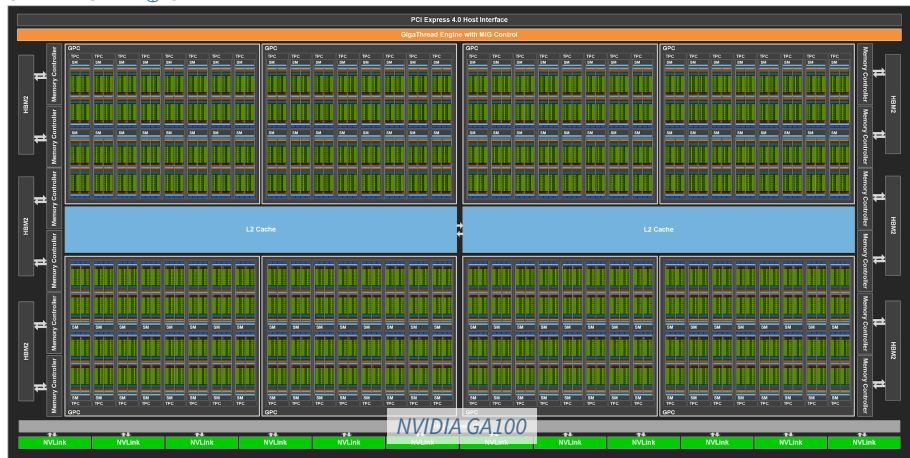


SIMT



SIMT

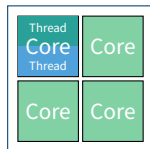
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



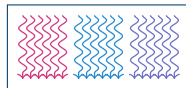
Vector

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

SMT



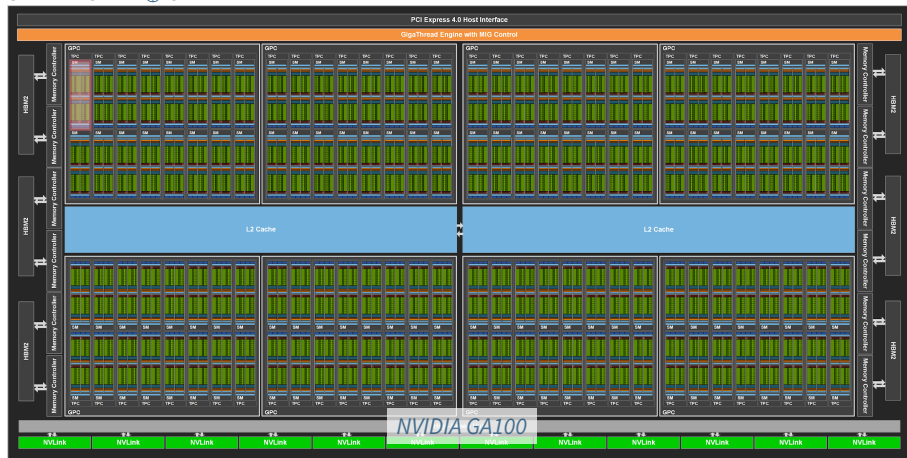
SIMT



Graphics: img:amperepictures

SIMT

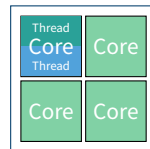
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



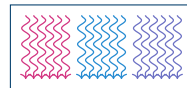
Vector

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

SMT



SIMT



Graphics: img:amperepictures

SIMT = SIMD $\oplus$ SMT

SM

L1 Instruction Cache

L0 Instruction Cache

Warp Scheduler (32 thread/cik)

Dispatch Unit (32 thread/cik)

Register File (16,384 x 32-bit)

INT32	INT32	FP32	FP32	FP64	TENSOR CORE
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	SFU
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	

L0 Instruction Cache

Warp Scheduler (32 thread/cik)

Dispatch Unit (32 thread/cik)

Register File (16,384 x 32-bit)

INT32	INT32	FP32	FP32	FP64	TENSOR CORE
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	SFU
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	

L1 Instruction Cache

L0 Instruction Cache

Warp Scheduler (32 thread/cik)

Dispatch Unit (32 thread/cik)

Register File (16,384 x 32-bit)

INT32	INT32	FP32	FP32	FP64	TENSOR CORE
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	SFU
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	

L0 Instruction Cache

Warp Scheduler (32 thread/cik)

Dispatch Unit (32 thread/cik)

Register File (16,384 x 32-bit)

INT32	INT32	FP32	FP32	FP64	TENSOR CORE
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
INT32	INT32	FP32	FP32	FP64	
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	SFU
LD/ST	LD/ST	LD/ST	LD/ST	LD/ST	

192KB L1 Data Cache / Shared Memory

Tex

Tex

Tex

Tex

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

Graphics: [img:amperepictures](#)

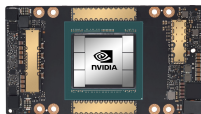
CPU vs. GPU

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

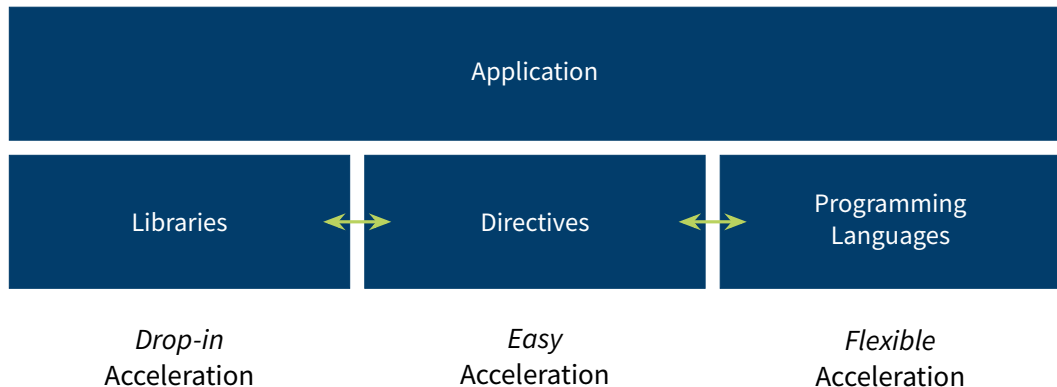
Part of LAPACK BLAS Level 1

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

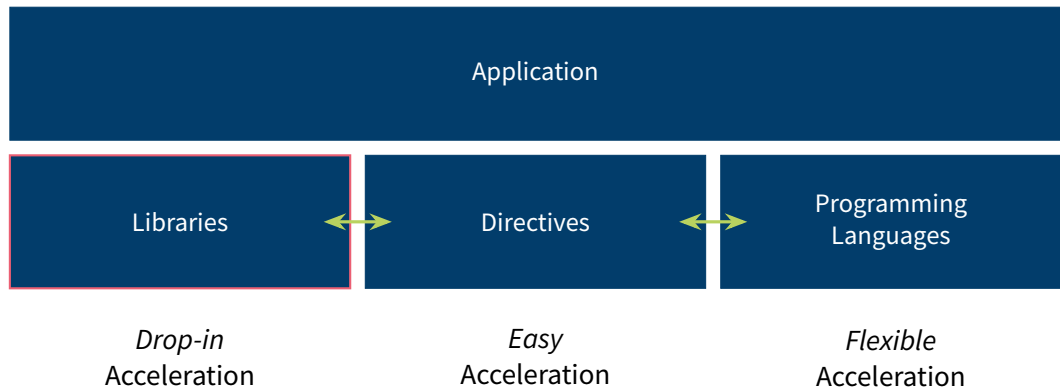
```
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y
```

```
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



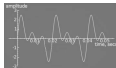
cuBLAS



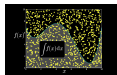
cuSPARSE



cuDNN



cuFFT



cuRAND



CUDA Math



{A} ARRAYFIRE

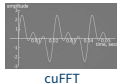
Numba

theano

Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries



Numba

theano



- 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]));
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);

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]));  
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
```

Allocate GPU memory

Copy data to GPU

```
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

CUDA C/C++

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+

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+*

- 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+

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



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



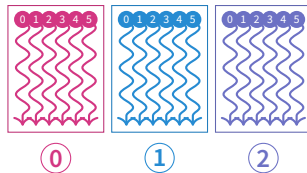
CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks



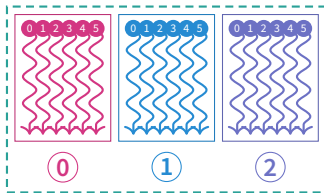
CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid



CUDA's Parallel Model

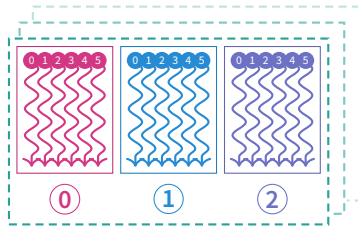
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



CUDA's Parallel Model

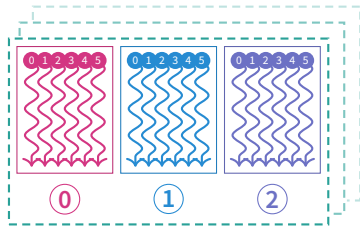
In software: Threads, Blocks

- 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!

CUDA SAXPY

With runtime-managed data transfers

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n)  
        y[i] = a * x[i] + y[i];  
}  
  
int a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));  
  
saxpy_cuda<<<2, 5>>>(n, a, x, y);  
  
cudaDeviceSynchronize();
```

CUDA SAXPY

With runtime-managed data transfers

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

Specify kernel

ID variables

Guard against
too many threads

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

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

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

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

Allocate GPU-capable
memory

Call kernel
2 blocks, each 5 threads

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

Wait for
kernel to finish

```
cudaDeviceSynchronize();
```

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

- OpenCL: `CodeXL` (Open Source, GPUOpen/AMD) – debugging, profiling.

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

CUDA API Statistics:

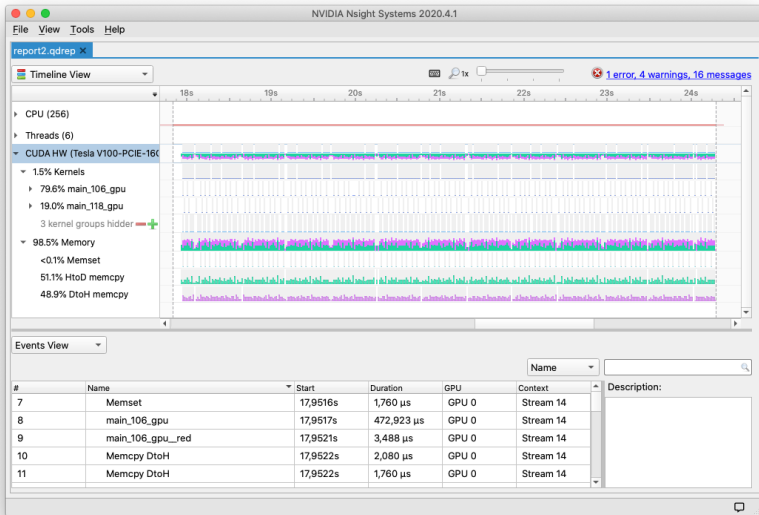
Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
90.9	160,407,572	30	5,346,919.1	1,780	25,648,117	cuStreamSynchronize

CUDA Kernel Statistics:

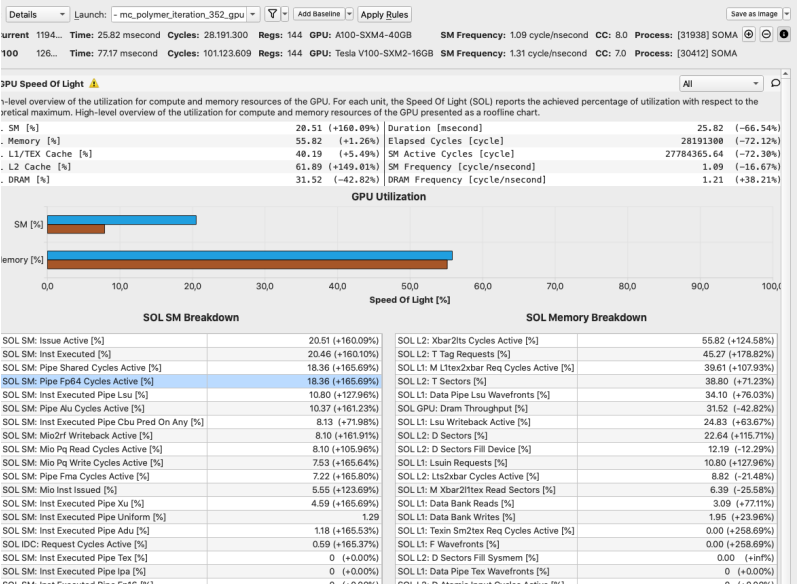
Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	158,686,617	10	15,868,661.7	14,525,819	25,652,783	main_106_gpu
0.0	25,120	10	2,512.0	2,304	3,680	main_106_gpu__red

Nsight Systems

GUI



Nsight Compute GUI



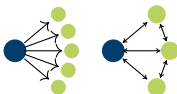
Programming GPUs

Advanced Topics

Advanced Topics

So much more interesting things to show!

- Optimize memory transfers to reduce overhead
- Optimize applications for GPU architecture
- Drop-in BLAS acceleration with NVBLAS (`$LD_PRELOAD`)
- Tensor Cores for Deep Learning
- Libraries, Abstractions: [Kokkos](#), [Alpaka](#), [Futhark](#), [HIP](#), [SYCL](#), [C++AMP](#), [C++ pSTL](#), ...
- Use multiple GPUs
 - On one node
 - Across many nodes → MPI
- ...
- Some of that: Addressed at **dedicated training courses**



Using GPUs on JSC Systems

Compiling

CUDA

- Module: `module load CUDA/11.0`
- Compile: `nvcc file.cu`
Default host compiler: `g++`; use `nvcc_pgc++` for PGI compiler
- Example cuBLAS: `g++ file.cpp -I$CUDA_HOME/include -L$CUDA_HOME/lib64 -lcublas -lcudart`

OpenACC

- Module: `module load NVHPC/20.9-GCC-9.3.0`
- Compile: `nvc++ -acc=gpu file.cpp`

MPI CUDA-aware MPIs (with direct Device-Device transfers)

ParaStationMPI `module load ParaStationMPI/5.4.7-1 mpi-settings/CUDA`

OpenMPI `module load OpenMPI/4.1.0rc1 mpi-settings/CUDA`

Running

- Dedicated GPU partitions

JUWELS

`--partition=gpus` 46 nodes (Job limits: ≤ 1 d)
`--partition=develgpus` 10 nodes (Job limits: ≤ 2 h, ≤ 2 nodes)

JUWELS Booster

`--partition=booster` 926 nodes
`--partition=develbooster` 10 nodes (Job limits: ≤ 1 d, ≤ 2 nodes)

JURECA DC

`--partition=dc-gpu` 192 nodes
`--partition=dc-gpu-devel` ?? nodes

- Needed: Resource configuration with `--gres=gpu:4`

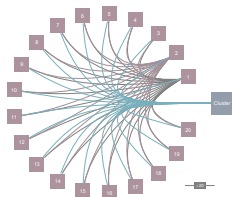
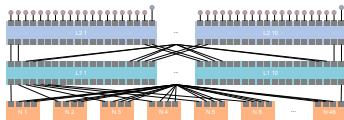
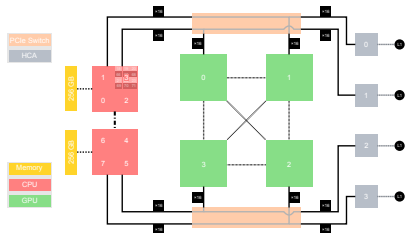
→ See [online documentation](#)

Running

JUWELS Booster Topology

- JUWELS Booster: NPS-4 (in total: 8 NUMA Domains)
- Not all have GPU or HCA affinity!
- Network is structured into two levels: In-Cell and Inter-Cell (DragonFly+ network)

→ Documentation:
apps.fz-juelich.de/jsc/hps/juwels/



Example

- 16 tasks in total, running on 4 nodes
- Per node: 4 GPUs

```
#!/bin/bash -x
#SBATCH --nodes=4
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=4
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:15:00
```

```
#SBATCH --partition=gpus
#SBATCH --gres=gpu:4
```

```
srun ./gpu-prog
```



Conclusion

Conclusion, Resources

- GPUs provide highly-parallel computing power
- We have many devices installed at JSC, ready to be used!

Conclusion, Resources

- GPUs provide highly-parallel computing power
- We have many devices installed at JSC, ready to be used!
- Training courses by JSC
 - CUDA Course April 2022
 - OpenACC Course October 2021
- Generally: see [online documentation](#) and sc@fz-juelich.de

Conclusion, Resources

- GPUs provide highly-parallel computing power
- We have many devices installed at JSC, ready to be used!
- Training courses by JSC
 - CUDA Course April 2022
 - OpenACC Course October 2021
- Generally: see [online documentation](#) and sc@fz-juelich.de
- Further consultation via our lab: NVIDIA Application Lab in Jülich; [contact me!](#)

Conclusion, Resources

- GPUs provide highly-parallel computing power
- We have many devices installed at JSC, ready to be used!
- Training courses by JSC
 - CUDA Course April 2022
 - OpenACC Course October 2021
- Generally: see [online documentation](#) and sc@fz-juelich.de
- Further consultation via our lab: NVIDIA Application Lab

**Thank you
for your attention!**
a.herten@fz-juelich.de

Appendix

Appendix
Glossary
References

Glossary I

AMD Manufacturer of CPUs and GPUs. 49, 50, 51, 52, 53, 54, 85, 87

Ampere GPU architecture from NVIDIA (announced 2019). 4, 5, 6

API A programmatic interface to software by well-defined functions. Short for application programming interface. 49, 50, 51, 52, 53, 54

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 74, 79, 80, 81, 82, 87

HIP GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 49, 50, 51, 52, 53, 54

Glossary II

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. [2](#), [79](#), [80](#), [81](#), [82](#), [86](#)

JURECA A multi-purpose supercomputer at JSC. [6](#)

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. [3](#), [4](#), [5](#), [75](#)

MPI The Message Passing Interface, a API definition for multi-node computing. [72](#), [74](#)

NVIDIA US technology company creating [GPUs](#). [3](#), [4](#), [5](#), [6](#), [26](#), [27](#), [28](#), [49](#), [50](#), [51](#), [52](#), [53](#), [54](#), [67](#), [79](#), [80](#), [81](#), [82](#), [85](#), [87](#)

OpenACC Directive-based programming, primarily for many-core machines. [43](#), [44](#), [45](#), [46](#), [47](#), [74](#), [79](#), [80](#), [81](#), [82](#)

Glossary III

- OpenCL** The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 49, 50, 51, 52, 53, 54, 67
- OpenMP** Directive-based programming, primarily for multi-threaded machines. 43, 44, 45, 46, 47
- ROCm** AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 49, 50, 51, 52, 53, 54
- SAXPY** Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 31, 64, 65
- Tesla** The GPU product line for general purpose computing computing of NVIDIA. 3

Glossary IV

CPU Central Processing Unit. 3, 6, 10, 11, 12, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 31, 49, 50, 51, 52, 53, 54, 85, 87

GPU Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 34, 35, 36, 37, 38, 39, 42, 43, 44, 45, 48, 49, 50, 51, 52, 53, 54, 65, 66, 67, 71, 72, 73, 75, 76, 77, 79, 80, 81, 82, 85, 86, 87

SIMD Single Instruction, Multiple Data. 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

SIMT Single Instruction, Multiple Threads. 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

SM Streaming Multiprocessor. 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

SMT Simultaneous Multithreading. 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

References I

- [2] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/> (pages 8, 9).
- [5] Wes Breazell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/> (pages 34–38).

References: Images, Graphics I

- [1] Forschungszentrum Jülich GmbH (Ralf-Uwe Limbach). *JUWELS Booster*.
- [3] Mark Lee. *Picture: kawasaki ninja*. URL:
<https://www.flickr.com/photos/pochacco20/39030210/> (pages 10, 11).
- [4] Shearings Holidays. *Picture: Shearings coach 636*. URL:
<https://www.flickr.com/photos/shearings/13583388025/> (pages 10, 11).