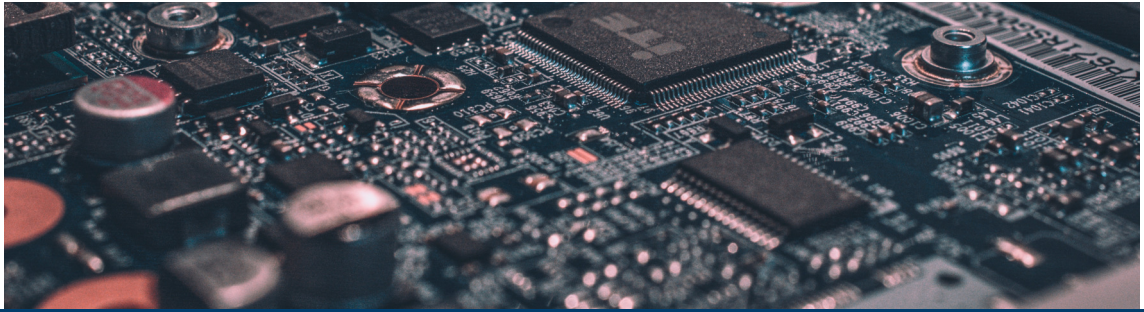


GPU ACCELERATORS AT JSC
SUPERCOMPUTING INTRODUCTION COURSE

26 November 2020 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

Outline

- GPUs at JSC
  - JUWELS
  - JURECA DC
- GPU Architecture
  - Empirical Motivation
  - Comparisons
  - GPU Architecture
- Programming GPUs
  - Libraries
  - Directives
  - CUDA C/C++
  - Performance Analysis
  - Advanced Topics
- Using GPUs on JSC Systems
  - Compiling
  - Resource Allocation

JUWELS Cluster – Jülich's Scalable System

- 2500 nodes with Intel Xeon CPUs (2×24 cores)
- $46 + 10$ nodes with 4 NVIDIA Tesla V100 cards (32 GB memory)
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance (Top500: #44)

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: *FP64TC:* 19.5 *FP64:* 9.7 TFLOP/s)
- InfiniBand DragonFly+ HDR-200 network; 4×200 Gbit/s per node

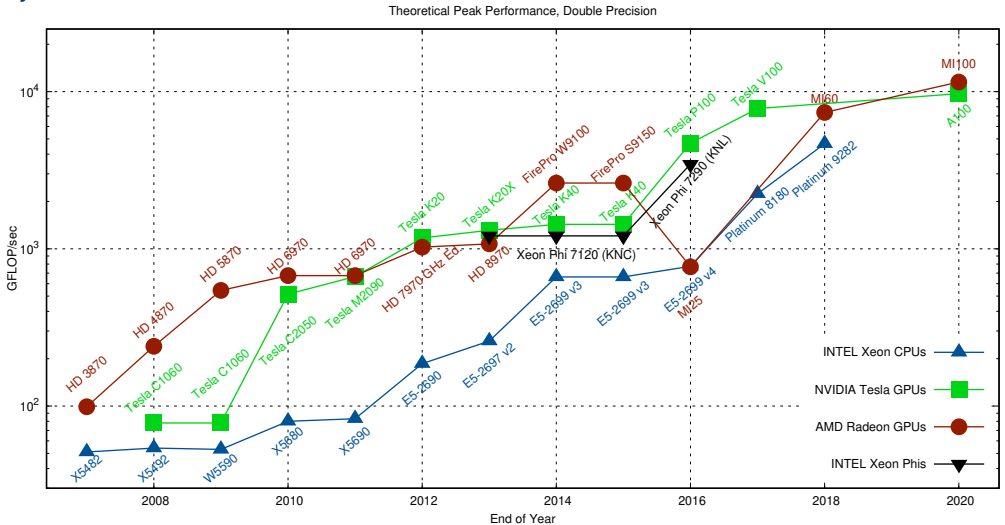
JURECA DC – Multi-Purpose

- 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

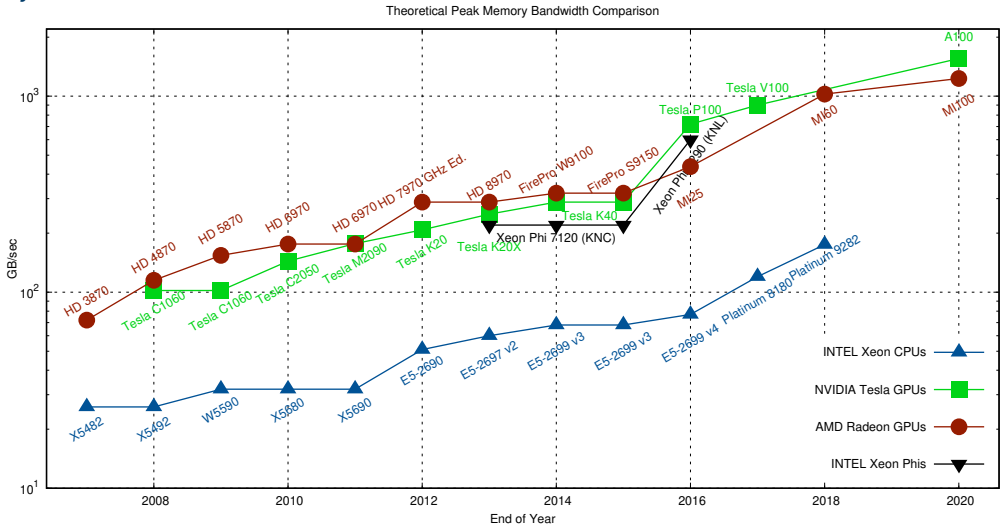
Memory Bandwidth



Graphic: Rupp [2]

Status Quo Across Architectures

Memory Bandwidth



Graphic: Rupp [2]

CPU vs. GPU

A matter of specialties

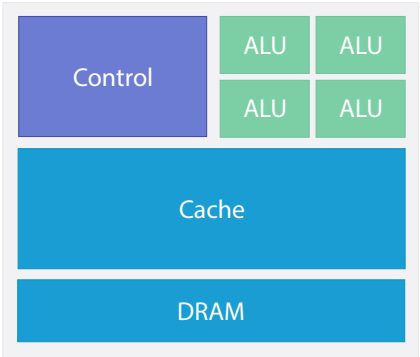
Transporting one

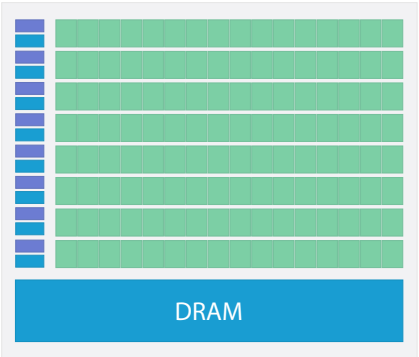
Transporting many

Graphics: Lee [3] and Shearings Holidays [4]

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)
  - Stage automatically (*Unified Memory*), or manually
- Two engines: Overlap compute and copy

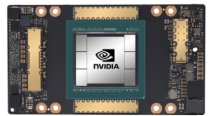
- Single Instruction, Multiple Threads (SIMT)

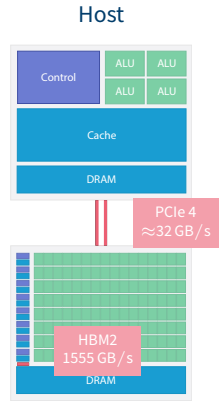
V100

32 GB RAM, 900 GB/s

A100

40 GB RAM, 1555 GB/s





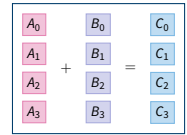
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

SIMT

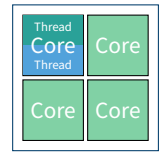
SIMT = SIMD ⊕ SMT

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

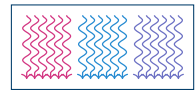
Vector



SMT

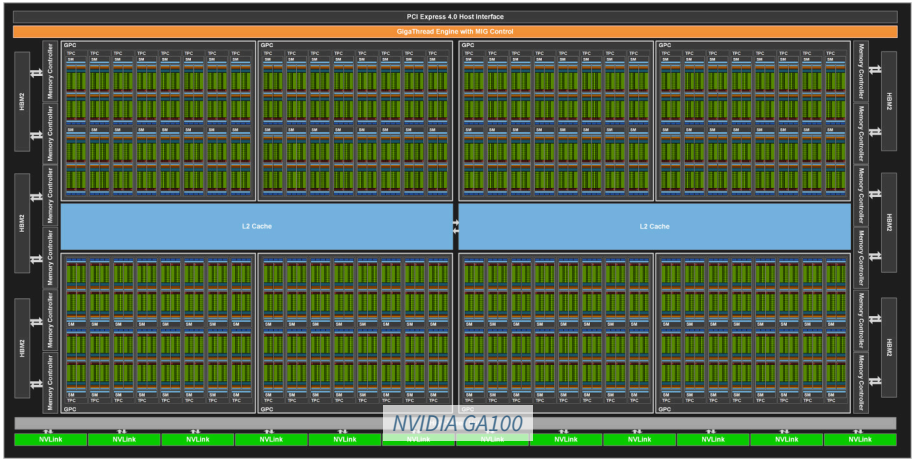


SIMT

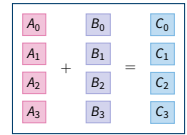


SIMT

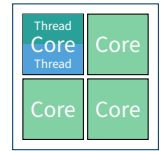
SIMT = SIMD ⊕ SMT



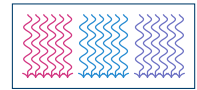
Vector



SMT



SIMT



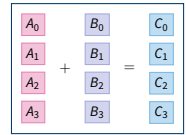
Graphics: img.amperepictures

SIMT

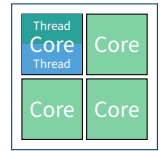
SIMT — SIMD ⊕ SMT

Multiprocessor

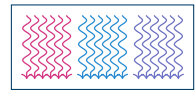
Vector



SMT



SIMT



Graphics: img.amperepictures

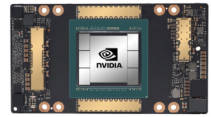
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

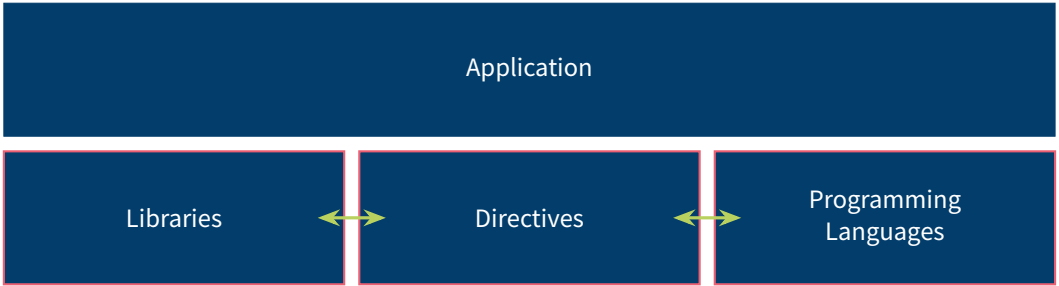
Preface: CPU

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];
}

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

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

Summary of Acceleration Possibilities



Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Libraries

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

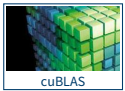
Use applications & libraries

Wizard: Breazell [5]

Libraries

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

Use applications & libraries


cuBLAS


cuSPARSE

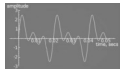

cuDNN

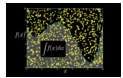

OpenCV

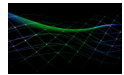

Thrust

ArrayFire

Numba


cuFFT


cuRAND

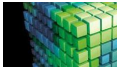

CUDA Math

theano

Wizard: Breazell [5]

cuBLAS

Parallel algebra



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

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

cuBLAS

Code example

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

cublasHandle_t handle;
cublasCreate(&handle);

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

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);
```

Initialize

Allocate GPU memory

Copy data to GPU

Call BLAS routine

Copy result to host

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++) {};
```

- **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

- Compiler support only raising
- Not all the raw power available
- Harder to debug
- Easy to program wrong

OpenACC

Code example

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

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

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

OpenACC

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 GPU 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**

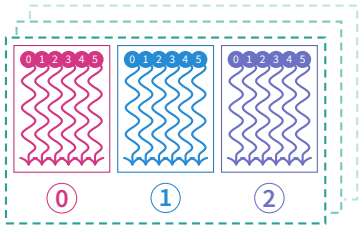
HIP AMD's new unified programming model for AMD (via ROCm) and NVIDIA GPUs *2016+*

- 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:
  - Thread → Block
  - Block → Grid
  - Threads & blocks in 3D



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

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();
```

Specify kernel

ID variables

Guard against too many threads

Allocate GPU-capable memory

Call kernel 2 blocks, each 5 threads

Wait for kernel to finish

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)
Nsight Systems	GPU program profiler with timeline
Nsight Compute	GPU kernel profiler

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

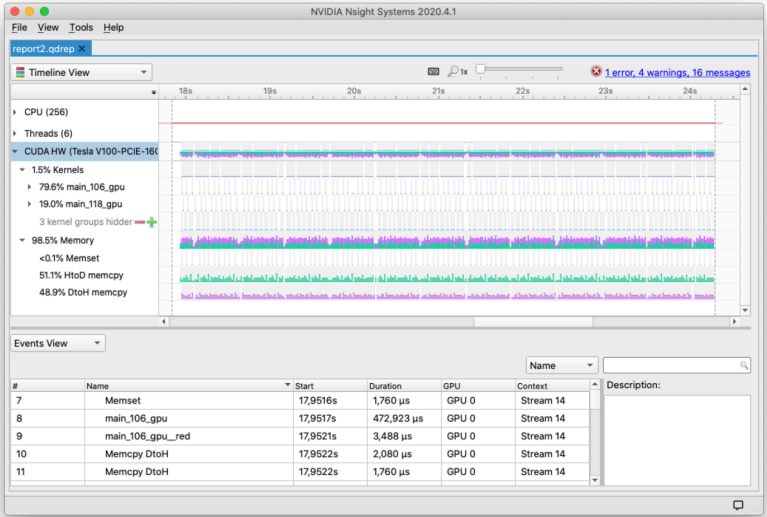
Nsight Systems

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
```

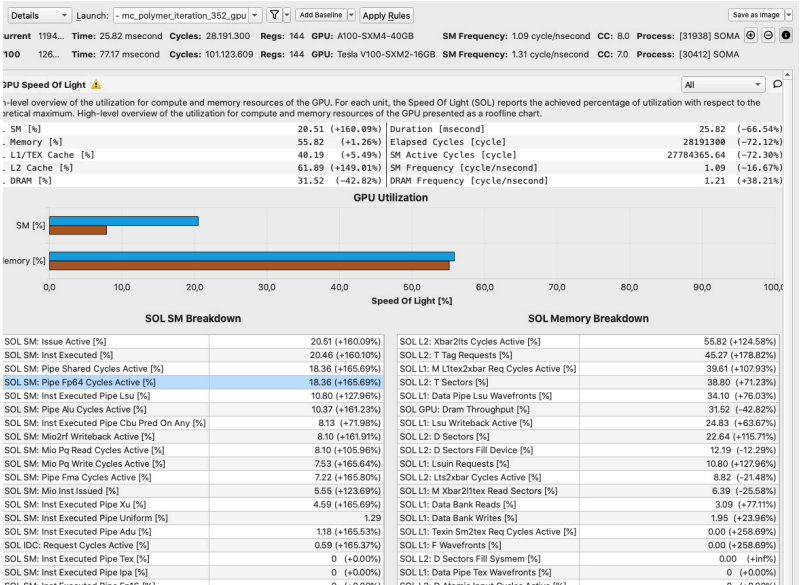
Nsight Systems

GUI



Nsight Compute

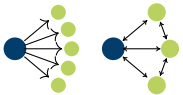
GUI



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: $\leq$1 d)
`--partition=develgpus`	10 nodes (Job limits: $\leq$2 h, $\leq$ 2 nodes)

  JUWELS Booster

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

  JURECA DC

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

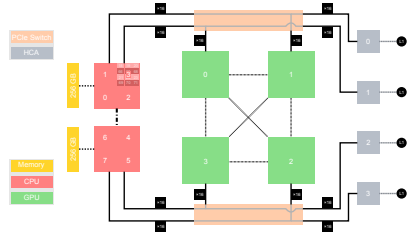
- Needed: Resource configuration with `--gres=gpu:4`

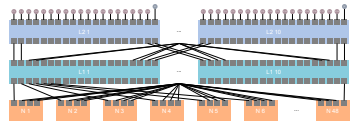
$\rightarrow$ See online documentation

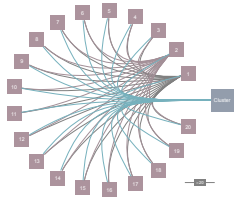
Running

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!
- Training courses by JSC

  CUDA Course April 2021
  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. 29, 47, 49

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

API A programmatic interface to software by well-defined functions. Short for application programming interface. 29

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 28, 29, 30, 31, 39, 44, 48

HIP GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 29

Glossary II

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 2, 44, 48

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. 5

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 3, 4, 40

MPI The Message Passing Interface, a API definition for multi-node computing. 37, 39

NVIDIA US technology company creating GPUs. 3, 4, 5, 13, 14, 29, 33, 44, 47, 49

OpenACC Directive-based programming, primarily for many-core machines. 25, 26, 27, 39, 44

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 29, 33

Glossary III

OpenMP Directive-based programming, primarily for multi-threaded machines. 25

ROCm AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 29

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 17, 31

Tesla The GPU product line for general purpose computing computing of NVIDIA. 3

CPU Central Processing Unit. 3, 5, 9, 10, 12, 13, 14, 17, 29, 47, 48

GPU Graphics Processing Unit. 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 19, 20, 21, 24, 25, 28, 29, 31, 32, 33, 37, 38, 40, 41, 42, 44, 47, 48, 49

Glossary IV

SIMD Single Instruction, Multiple Data. 12, 13, 14

SIMT Single Instruction, Multiple Threads. 11, 12, 13, 14

SM Streaming Multiprocessor. 12, 13, 14

SMT Simultaneous Multithreading. 12, 13, 14

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 7, 8).

[5] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 19, 20).

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/ (page 9).

[4] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (page 9).