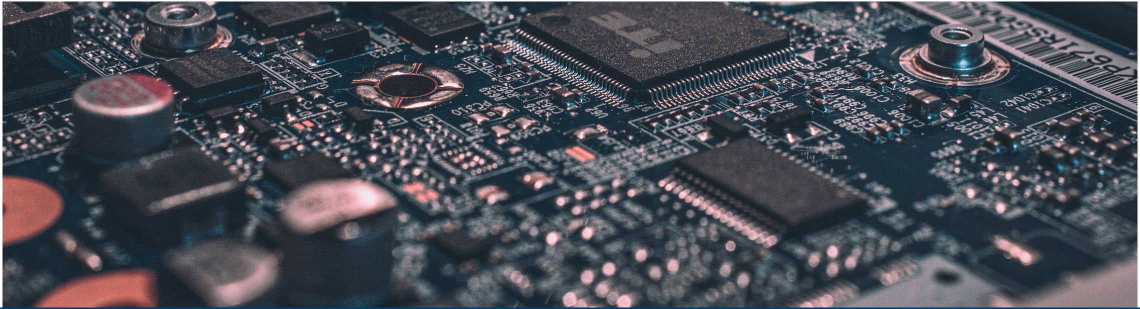


GPU ACCELERATORS AT JSC

SUPERCOMPUTING INTRODUCTION COURSE

29 November 2019 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Outline

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

2020:
Booster!

JUWELS – Jülich's New Large System

- 2500 nodes with Intel Xeon CPUs (2 × 24 cores)
- 46 + 10 nodes with 4 NVIDIA Tesla V100 cards
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance (Top500: #26)

JURECA – Jülich's Multi-Purpose Supercomputer

- 1872 nodes with Intel Xeon E5 CPUs (2 × 12 cores)
- 75 nodes with 2 NVIDIA Tesla K80 cards (look like 4 GPUs)
- JURECA Booster: 1640 nodes with Intel Xeon Phi *Knights Landing*
- 1.8 (CPU) + 0.44 (GPU) + 5 (KNL) PFLOP/s peak performance (Top500: #44)
- Mellanox EDR InfiniBand

JURON – A Human Brain Project *Pilot System*

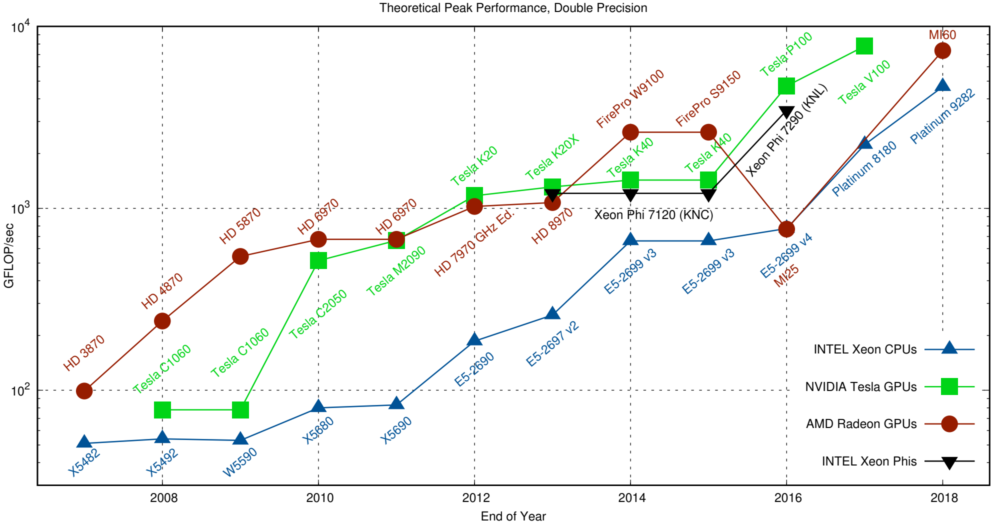
- 18 nodes with IBM POWER8*NVL* CPUs (2 × 10 cores)
- Per Node: 4 NVIDIA Tesla P100 cards (16 GB HBM2 memory), connected via NVLink
- GPU: 0.38 PFLOP/s peak performance

GPU Architecture

Why?

Status Quo Across Architectures

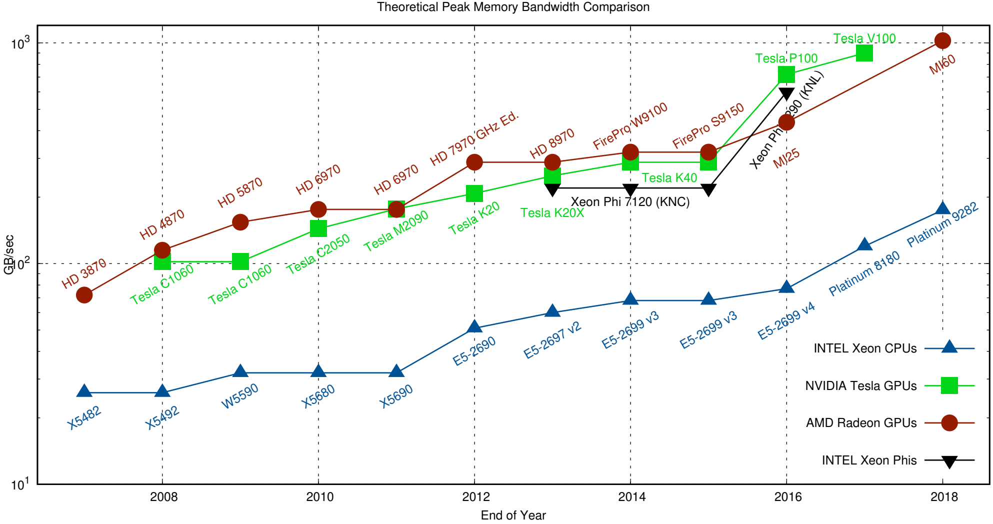
Memory Bandwidth



Graphic: Rupp [2]

Status Quo Across Architectures

Memory Bandwidth



Graphic: Rupp [2]

CPU vs. GPU

A matter of specialties

Transporting one

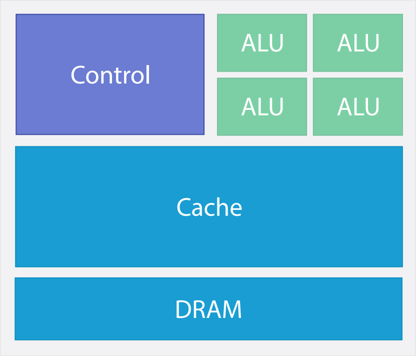
Transporting many

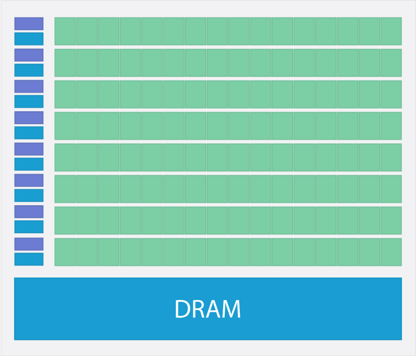
Graphics: Lee [3] and Bob Adams [4]

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

CPU vs. GPU

Chip





GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Memory

GPU memory ain't no CPU memory

Unified Virtual Addressing

- GPU: accelerator / extension card

→ Separate device from CPU

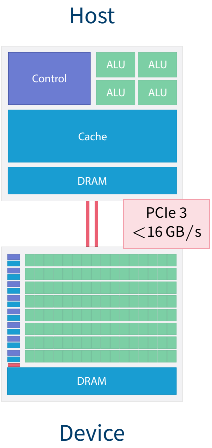
Separate memory, but UVA

- Memory transfers need special consideration!

  Do as little as possible!

- Formerly: Explicitly copy data to/from GPU

  Now: Done automatically (performance…?)

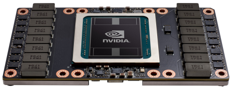


P100

16 GB RAM, 720 GB/s

V100

32 GB RAM, 900 GB/s



Memory

GPU memory ain't no CPU memory

Unified Memory

- GPU: accelerator / extension card

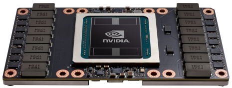
→ Separate device from CPU
  Separate memory, but UVA and UM
- Memory transfers need special consideration!
  Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
  Now: Done automatically (performance…?)

P100

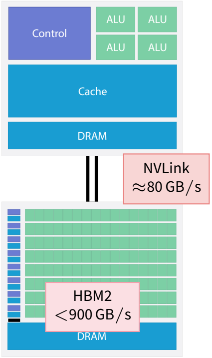
16 GB RAM, 720 GB/s

V100

32 GB RAM, 900 GB/s



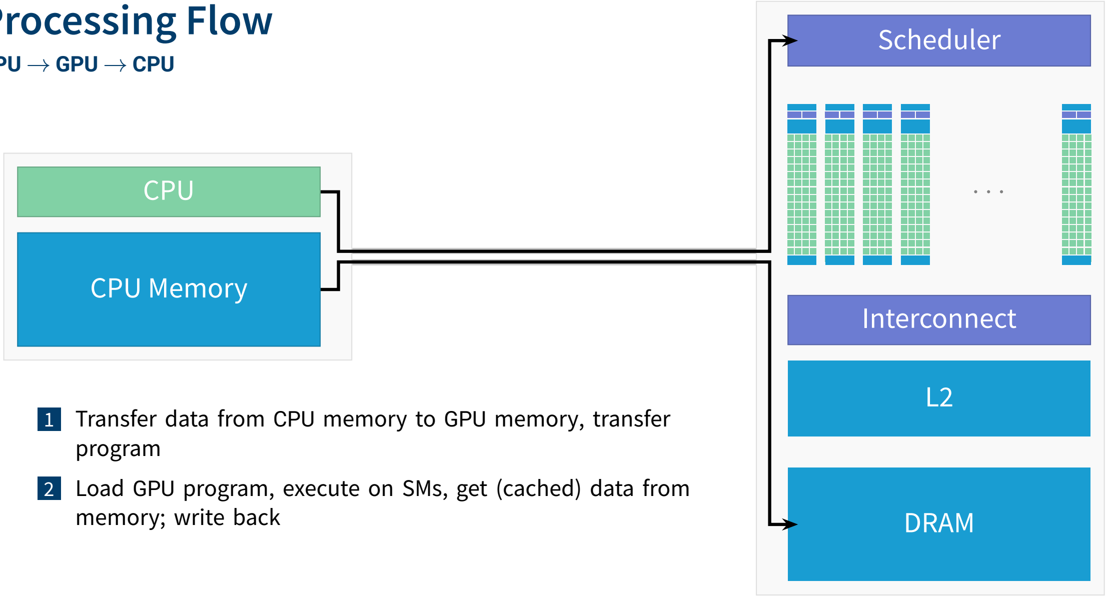
Host



Device

Processing Flow

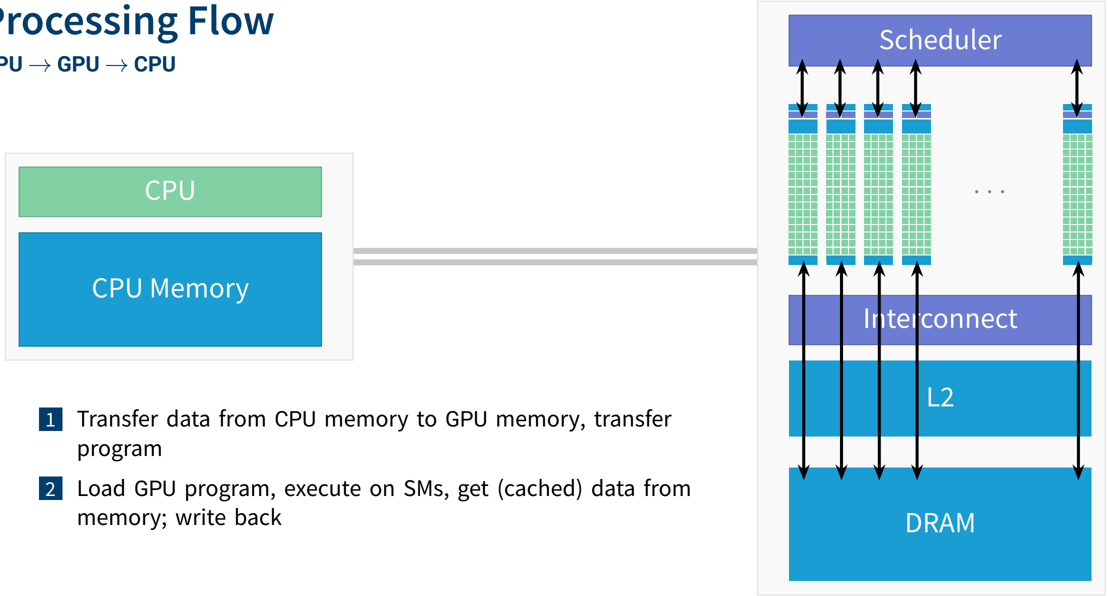
CPU → GPU → CPU



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

Processing Flow

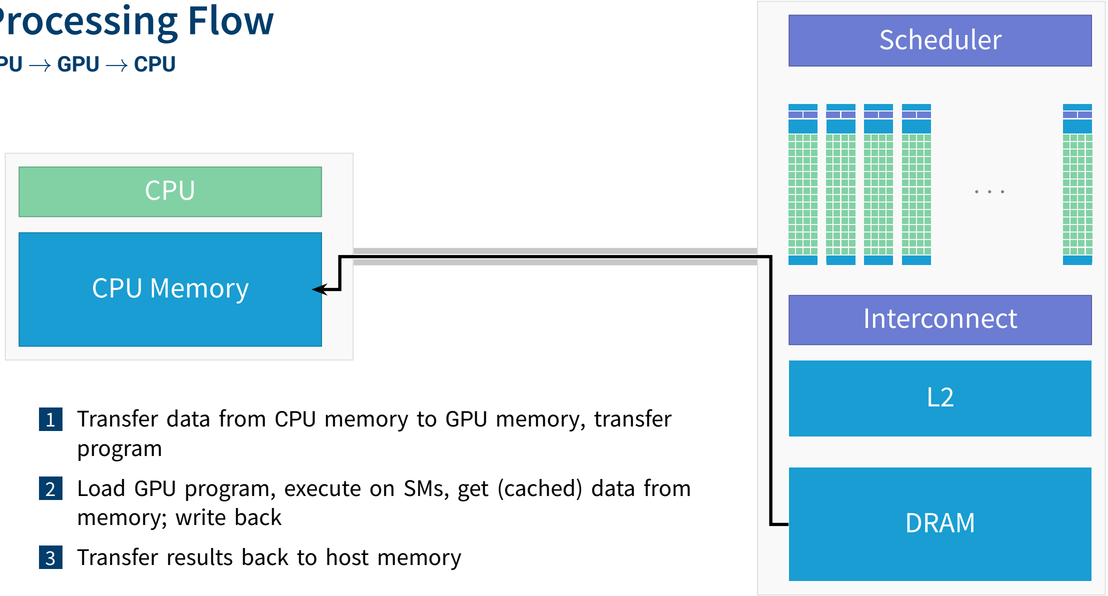
CPU $\rightarrow$ GPU $\rightarrow$ CPU



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

Processing Flow

CPU → GPU → CPU



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

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Async

Following different streams

- Problem: Memory transfer is comparably slow
  Solution: Do something else in meantime (**computation**)!

→ Overlap tasks

- Copy and compute engines run separately (*streams*)

- GPU needs to be fed: Schedule many computations
- CPU can do other work while GPU computes; synchronization

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

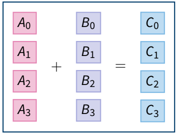
Memory

SIMT

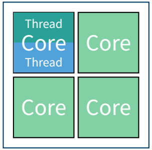
SIMT $=$ SIMD $\oplus$ SMT

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

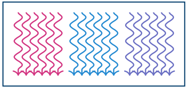
Vector



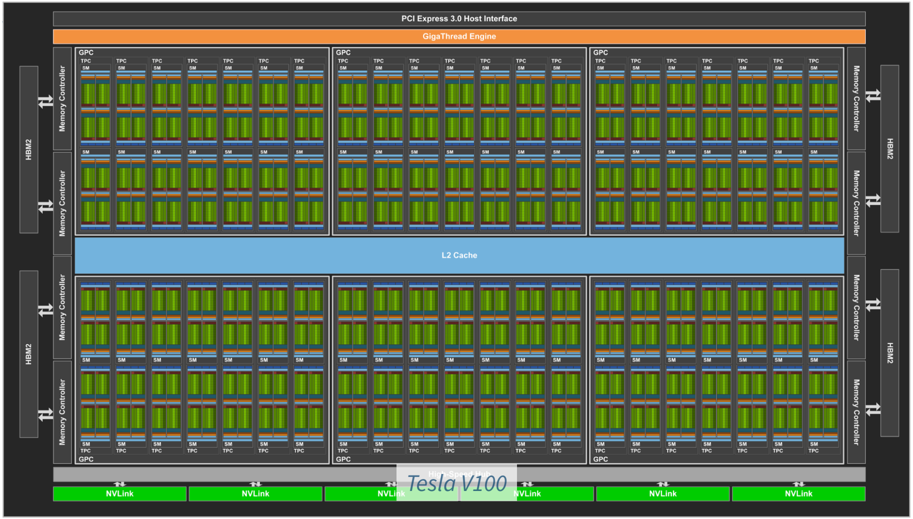
SMT



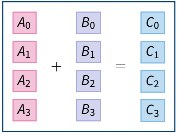
SIMT



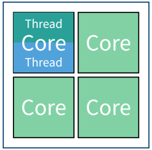
SIMT



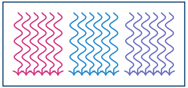
Vector



SMT



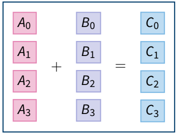
SIMT



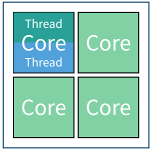
Graphics: Nvidia Corporation [5]

SIMT

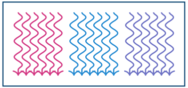
Vector



SMT



SIMT



Graphics: Nvidia Corporation [5]

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

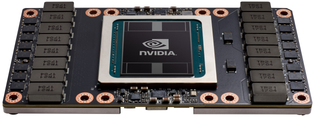
CPU Core: Low Latency

GPU Streaming Multiprocessor: High Throughput

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);
```

Libraries

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

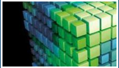
Use applications & libraries

Wizard: Breazell [6]

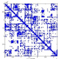
Libraries

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

Use applications & libraries



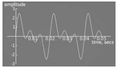
cuBLAS



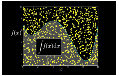
cuSPARSE

cuDNN

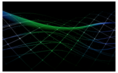
OpenCV



cuFFT



cuRAND



CUDA Math

Thrust

ArrayFire

Numba

theano

Wizard: Breazell [6]

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

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

```c
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 acc parallel loop copy(y) copyin(x)
  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**

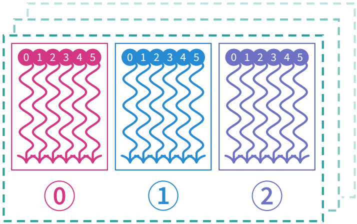
HIP AMD's new unified programming model for AMD (via ROCm) and NVIDIA GPUs *2016+*

- 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 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();
```

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
`cuda-memcheck`	Like Valgrind's `memcheck`, for checking errors in memory accesses
Nsight	IDE for GPU developing, based on Eclipse (Linux, OS X) or Visual Studio (Windows)
`nvprof`	**Command line profiler, including detailed performance counters**
Visual Profiler	**Timeline profiling and annotated performance experiments**
New	Nsight Systems and Nsight Compute; successors of Visual Profiler

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

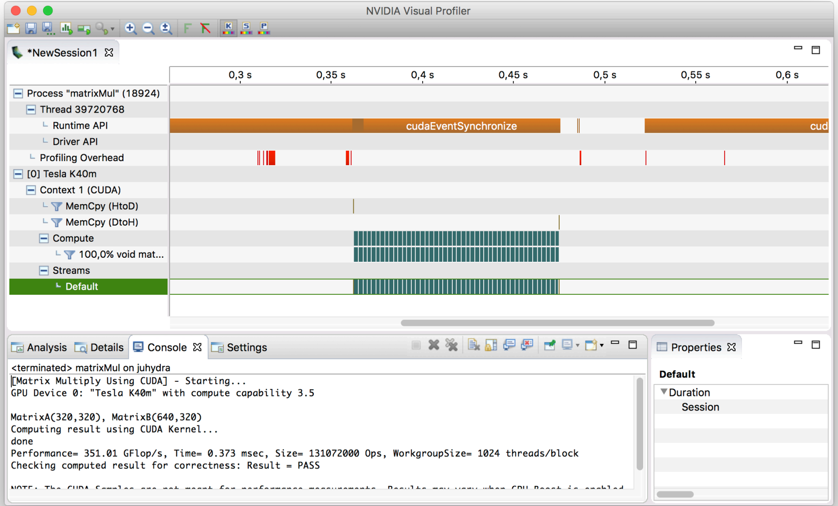
nvprof

Command that line

```
$ nvprof ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
==37064== Profiling application: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
==37064== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.19%  262.43ms       301  871.86us  863.88us  882.44us  void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
  0.58%  1.5428ms         2  771.39us  764.65us  778.12us  [CUDA memcpy HtoD]
  0.23%  599.40us         1  599.40us  599.40us  599.40us  [CUDA memcpy DtoH]

==37064== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 61.26%  258.38ms         1  258.38ms  258.38ms  258.38ms  cudaEventSynchronize
 35.68%  150.49ms         3  50.164ms  914.97us  148.65ms  cudaMalloc
  0.73%  3.0774ms         3  1.0258ms  1.0097ms  1.0565ms  cudaMemcpy
  0.62%  2.6287ms         4  657.17us  655.12us  660.56us  cuDeviceTotalMem
  0.56%  2.3408ms       301  7.7760us  7.3810us  53.103us  cudaLaunch
  0.48%  2.0111ms       364  5.5250us     235ns  201.63us  cuDeviceGetAttribute
  0.21%  872.52us         1  872.52us  872.52us  872.52us  cudaDeviceSynchronize
  0.15%  612.20us      1505     406ns     361ns  1.1970us  cudaSetupArgument
  0.12%  499.01us         3  166.34us  140.45us  216.16us  cudaFree
```

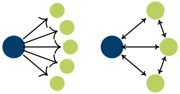
Visual Profiler



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, C++AMP, …
- Use multiple GPUs
  - On one node
  - Across many nodes → MPI



- …
- Some of that: Addressed at **dedicated training courses**

Using GPUs on JURECA & JUWELS

Compiling

CUDA

- Module: `module load CUDA/10.1.105`
- Compile: `nvcc file.cu`
  Default host compiler: `g++`; use `nvcc_pgc++` for PGI compiler
- cuBLAS: `g++ file.cpp -I$CUDA_HOME/include -L$CUDA_HOME/lib64 -lcublas -lcudart`

OpenACC

- Module: `module load PGI/19.3-GCC-8.3.0`
- Compile: `pgc++ -acc -ta=tesla file.cpp`

MPI

- Module: `module load MVAPICH2/2.3.2-GDR` (also needed: `GCC/8.3.0`)
  Enabled for CUDA (*CUDA-aware*); no need to copy data to host before transfer

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Running

- Dedicated GPU partitions

  JUWELS

`--partition=gpus`	46 nodes (Job limits: $<$1 d)
`--partition=develgpus`	10 nodes (Job limits: $<$2 h, $\leq$ 2 nodes)

  JURECA

`--partition=gpus`	70 nodes (Job limits: $<$1 d, $\leq$ 32 nodes)
`--partition=develgpus`	4 nodes (Job limits: $<$2 h, $\leq$ 2 nodes)

- Needed: Resource configuration with `--gres`

  `--gres=gpu:4`

  `--gres=mem1024,gpu:2 --partition=vis` *only JURECA*

→ See online documentation

Example

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

```
#!/bin/bash -x
#SBATCH --nodes=4
#SBATCH --ntasks=96
#SBATCH --ntasks-per-node=24
#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, Resources

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

  CUDA Course 4 - 6 May 2020

  OpenACC Course 26 - 27 October 2019
- Generally: see online documentation and sc@fz-juelich.de
- Further consultation via our lab: NVIDIA Application Lab in Jüli
- Interested in JURON? Get access!

Thank you for your attention!

a.herten@fz-juelich.de

Appendix

Appendix
- Glossary
- References

Glossary I

AMD Manufacturer of CPUs and GPUs. 39, 54, 56

API A programmatic interface to software by well-defined functions. Short for application programming interface. 39

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 38, 39, 40, 41, 48, 51, 56

HIP GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 39

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 2, 51, 55

Glossary II

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. 2, 4, 47, 49

JURON One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. 5

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 2, 3, 47, 49

MPI The Message Passing Interface, a API definition for multi-node computing. 46, 48

NVIDIA US technology company creating GPUs. 3, 4, 5, 39, 43, 51, 54, 55, 56, 57

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with high bandwidth. 5, 56

OpenACC Directive-based programming, primarily for many-core machines. 35, 36, 37, 48, 51

Glossary III

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 39, 43

OpenMP Directive-based programming, primarily for multi-threaded machines. 35

P100 A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast *HBM2* memory. 5

Pascal GPU architecture from NVIDIA (announced 2016). 56

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 56

POWER8 Version 8 of IBM's POWER processor, available also within the OpenPOWER Foundation. 5, 56

ROCm AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 39

Glossary IV

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 28, 41

Tesla The GPU product line for general purpose computing computing of NVIDIA. 3, 4, 5

CPU Central Processing Unit. 3, 4, 5, 10, 11, 14, 15, 16, 17, 18, 22, 23, 24, 28, 39, 54, 55, 56

GPU Graphics Processing Unit. 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 27, 29, 30, 31, 34, 35, 38, 39, 41, 42, 43, 46, 47, 49, 50, 51, 54, 55, 56, 57

HBP Human Brain Project. 55

SIMD Single Instruction, Multiple Data. 22, 23, 24

Glossary V

SIMT Single Instruction, Multiple Threads. 12, 13, 19, 21, 22, 23, 24

SM Streaming Multiprocessor. 22, 23, 24

SMT Simultaneous Multithreading. 22, 23, 24

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).

[6] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 29, 30).

References: Images, Graphics I

[1] Alexandre Debiève. *Bowels of computer*. Freely available at Unsplash. URL: https://unsplash.com/photos/FO7JIlwjOtU.

[3] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/. License: Creative Commons BY-ND 2.0 (page 10).

[4] Bob Adams. *Picture: Hylton Ross Mercedes Benz Irizar coach*. URL: https://www.flickr.com/photos/satransport/13197324714/. License: Creative Commons BY-SA 2.0 (page 10).

[5] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf (pages 23, 24).