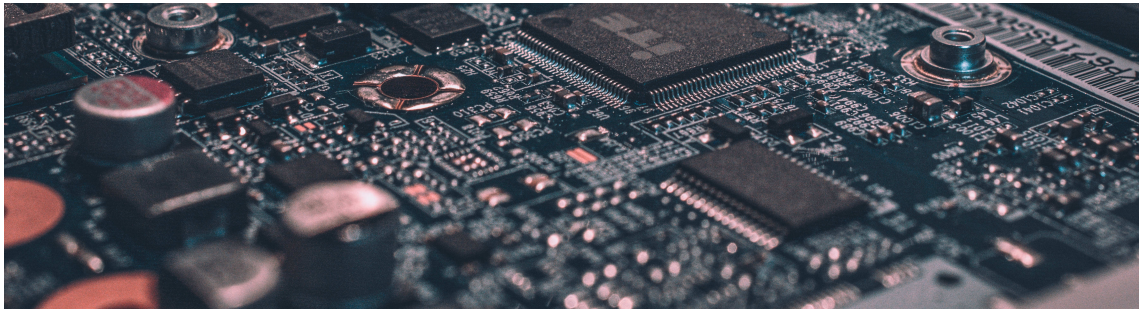




INTRODUCTION TO GPU PROGRAMMING OF THREADS AND KERNELS

7 February 2019 | Andreas Herten | ESM User Forum, Forschungszentrum Jülich

Outline

GPUs at JSC

JUWELS

JURECA

JURON

GPU Architecture

Empirical Motivation

Comparisons

3 Core Features

Memory

Asynchronicity

SIMT

High Throughput

Summary

Programming GPUs

Libraries

Directives

Languages

Abstraction Libraries/DSL

Tools

Advanced Topics

Using GPUs on JURECA & JUWELS

Compiling

Resource Allocation



Next:
Booster!

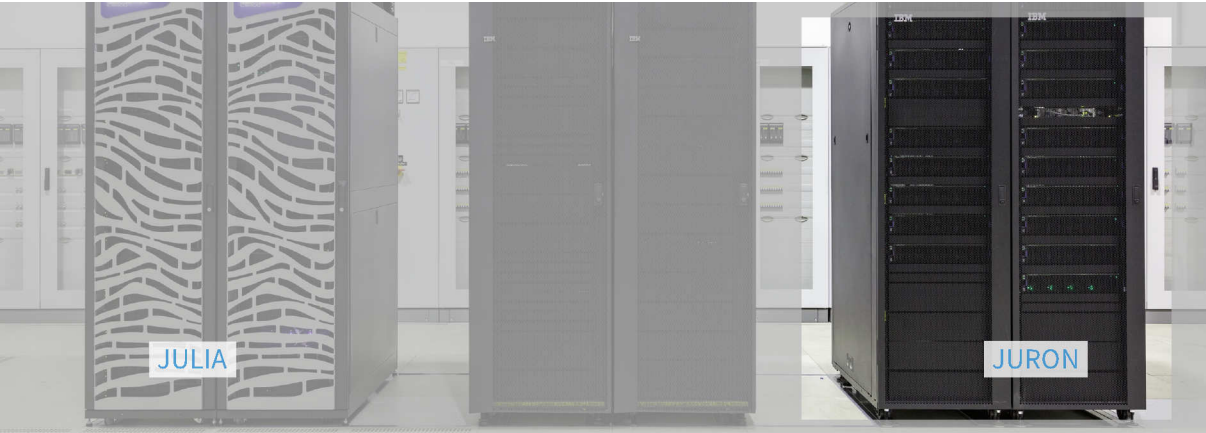
JUWELS – Jülich's New Scalable System

- 2500 nodes with Intel Xeon CPUs (2×24 cores)
- 48 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 *Prototype*

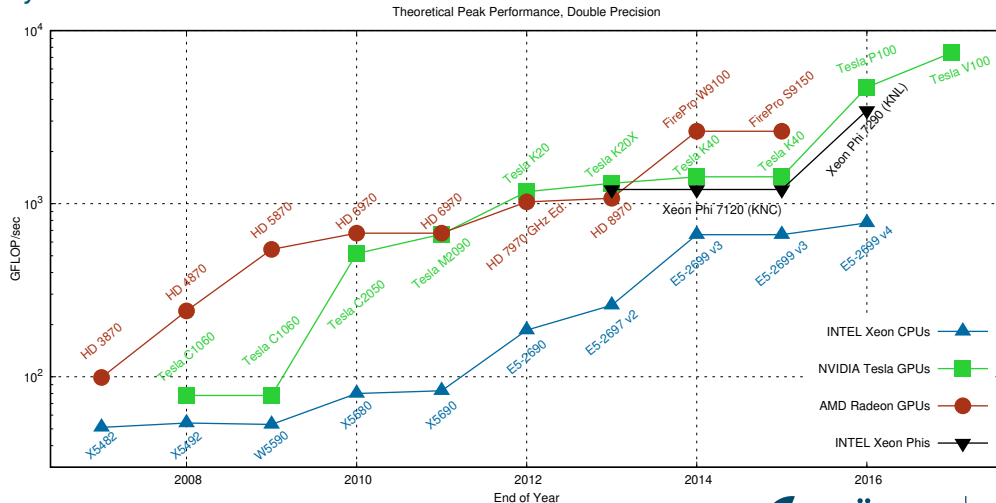
- 18 nodes with IBM POWER8NVL CPUs (2×10 cores)
- Per Node: 4 NVIDIA Tesla P100 cards (16 GB HBM2 memory), connected via NVLink
- GPU: 0.38 PFLOP/s peak performance

GPU Architecture

Why?

Status Quo Across Architectures

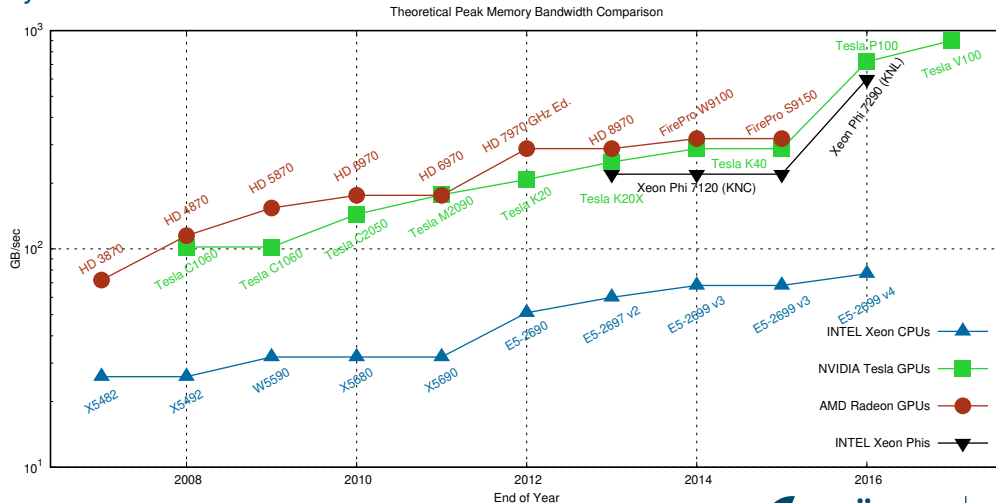
Memory Bandwidth



Graphic: Rupp [2]

Status Quo Across Architectures

Memory Bandwidth



Graphic: Rupp [2]

CPU vs. GPU

A matter of specialties



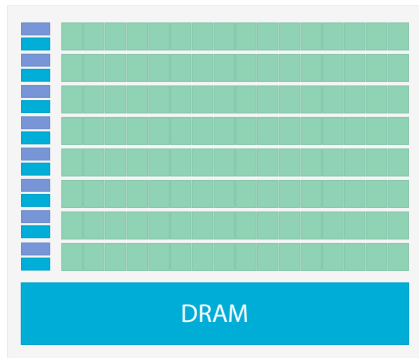
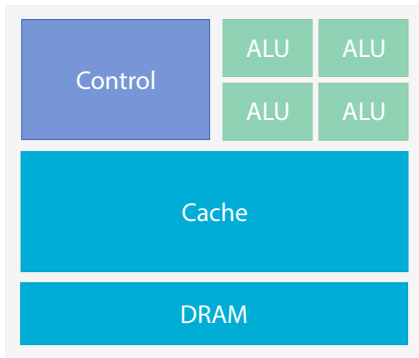
Transporting one



Transporting many

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

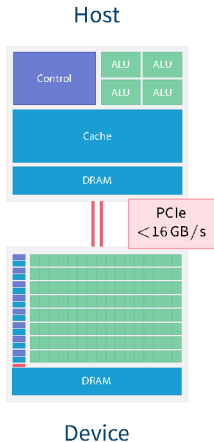
Memory

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...?)



Memory

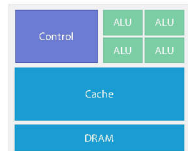
GPU memory ain't no CPU memory

- GPU: accelerator / extension card
- Separate device from CPU
- **Separate memory, but UVA and UM**
- Memory transfers need special consideration!
Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance...?)
- P100: 16 GB RAM, 720 GB/s; V100: 16 (32) GB RAM, 900 GB/s

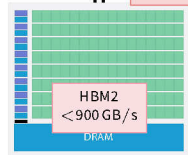
Unified Memory



Host



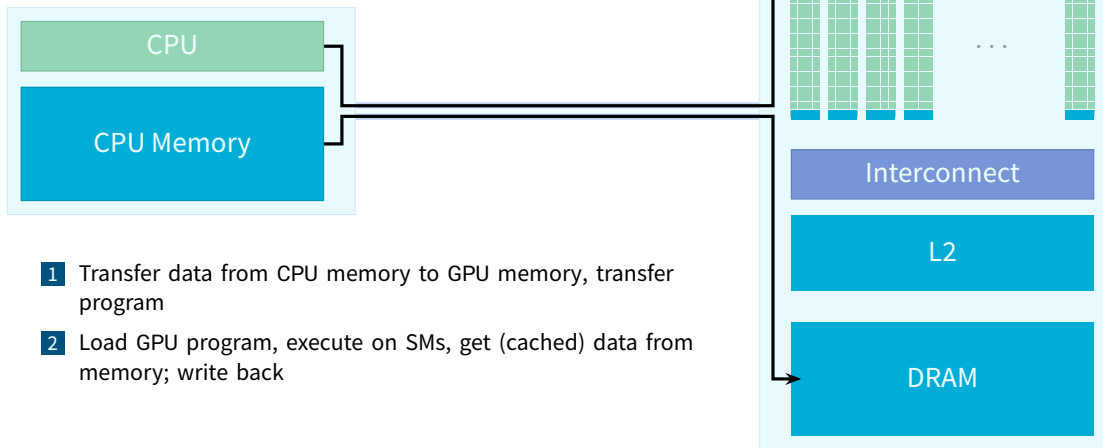
NVLink
≈ 80 GB/s



Device

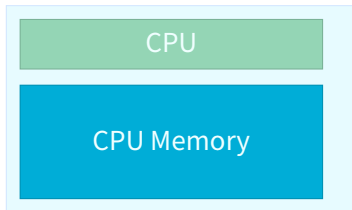
Processing Flow

CPU → GPU → CPU

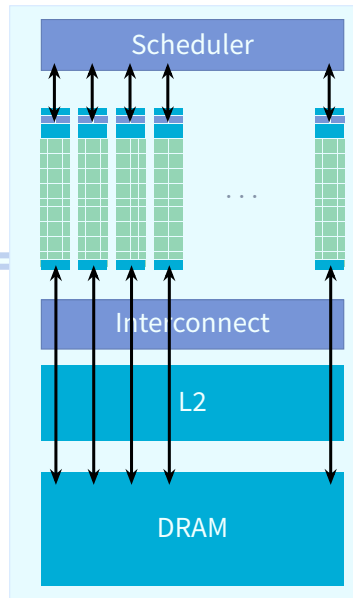


Processing Flow

CPU → GPU → CPU

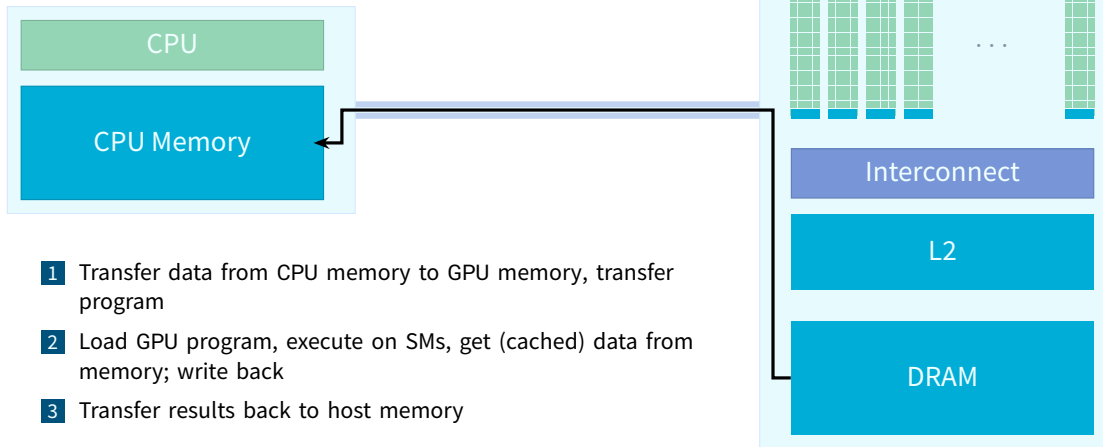


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back



Processing Flow

CPU → GPU → CPU



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

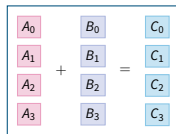
Memory

SIMT

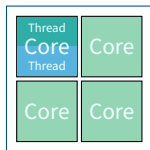
Of threads and warps

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

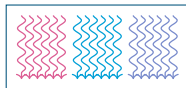
Vector



SMT



SIMT



SIMT

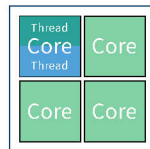
Of t



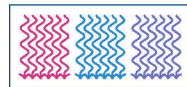
Vector

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

SMT



SIMT



Graphics: volta-pictures

SIMT

of 1

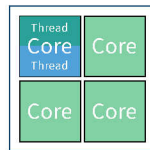
Multiprocessor



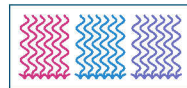
Vector

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

SMT



SIMT



Graphics: volta-pictures

SIMT

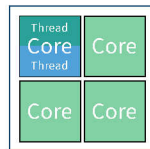
Of t



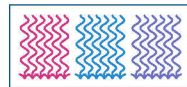
Vector

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

SMT



SIMT

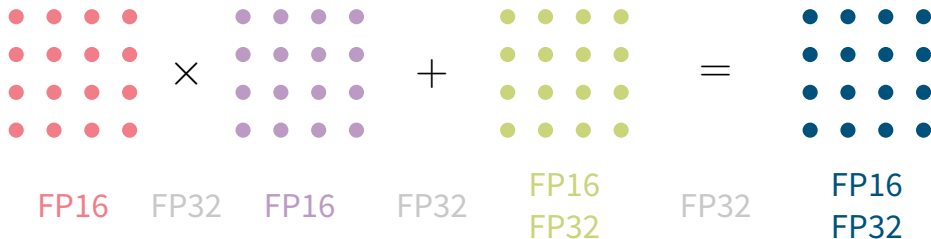


Graphics: volta-pictures

New: Tensor Cores

New in Volta

- 8 Tensor Cores per Streaming Multiprocessor (SM) (640 total for V100)
 - Performance: 125 TFLOP/s (half precision)
 - Calculate $\mathbf{A} \times \mathbf{B} + \mathbf{C} = \mathbf{D}$ (4×4 matrices; $\mathbf{A}, \mathbf{B}$: half precision)
- 64 floating-point FMA operations per clock (mixed precision)



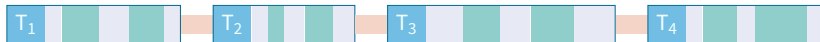
Low Latency vs. High Throughput

Maybe GPU's ultimate feature

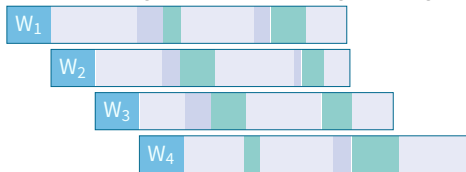
CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency



GPU Streaming Multiprocessor: High Throughput



- Thread/Warp
- Processing
- Context Switch
- Ready
- Waiting

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



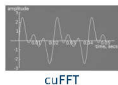
Libraries

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

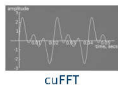
```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

Finalize

Libraries

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

Use applications & libraries



Numba



theano

Thrust

Iterators! Iterators everywhere! 

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- 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 with lambdas

```
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;

using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);

x = d_x;
```

Thrust

Code example with lambdas

```
#include <thrust/for_each.h>
#include <thrust/execution_policy.h>
constexpr int gGpuThreshold = 10000;
void saxpy(float *x, float *y, float a, int N) {
    auto r = thrust::counting_iterator<int>(0);

    auto lambda = [=] __host__ __device__ (int i) {
        y[i] = a * x[i] + y[i];};

    if(N > gGpuThreshold)
        thrust::for_each(thrust::device, r, r+N, lambda);
    else
        thrust::for_each(thrust::host, r, r+N, lambda);}
```

Source

Programming GPUs

Directives

GPU Programming with Directives

Keepin' you portable

- Annotate usual source code by directives

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

- Also: Generalized API functions

```
acc_copy();
```

- 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

- Compilers support limited
- Raw power hidden
- Somewhat harder to debug

GPU Programming with Directives

The power of... two.

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

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

OpenACC Similar to OpenMP, but more specifically for GPUs
Less *prescriptive*, more *descriptive*

OpenACC

Code example

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

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

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

OpenACC

Code example

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

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

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

Programming GPUs

Languages

Programming GPU Directly

Finally...

- Two solutions:

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
- Different compilers available

CUDA NVIDIA's GPU platform 2007

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

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



CUDA's Parallel Model

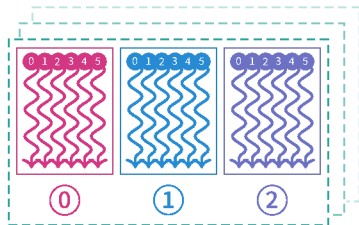
In software: Threads, Blocks

- 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];  
}
```

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

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

Call kernel
2 blocks, each 5 threads

```
cudaDeviceSynchronize();
```

Wait for
kernel to finish

Programming GPUs

Abstraction Libraries/DSL

Abstraction Libraries & DSLs

- Libraries with ready-programmed abstractions; partly compiler/transpiler necessary
- Have different backends to choose from for targeted accelerator
- Between Thrust, OpenACC, and CUDA
- Examples: **Kokkos**, **Alpaka**, **Futhark**, **HIP**, **C++AMP**, ...

An Alternative: Kokkos

From Sandia National Laboratories

- C++ library for *performance* portability
- Data-parallel patterns, architecture-aware memory layouts, ...

```
Kokkos::View<double*> x("X", length);  
Kokkos::View<double*> y("Y", length);  
double a = 2.0;  
  
// Fill x, y  
  
Kokkos::parallel_for(length, KOKKOS_LAMBDA (const int& i) {  
    x(i) = a*x(i) + y(i);  
});
```

→ <https://github.com/kokkos/kokkos/>

Programming GPUs

Tools

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

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

nvprof

Command that line

Usage: nvprof ./app

```
$ 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
```

nvprof

Command that line

With metrics: `nvprof --metrics flop_sp_efficiency ./app`

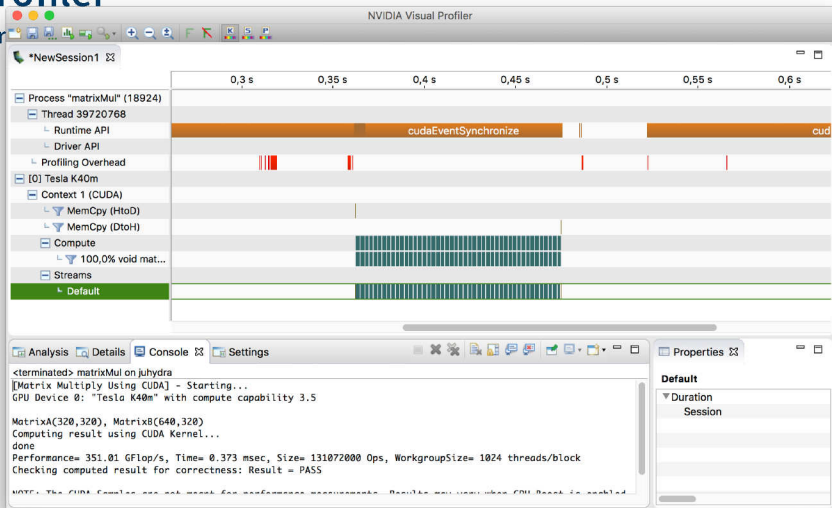
```
$ nvprof --metrics flop_sp_efficiency ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
[Matrix Multiply Using CUDA] - Starting...
==37122== NVPROF is profiling process 37122, command: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
GPU Device 0: "Tesla P100-SXM2-16GB" with compute capability 6.0

MatrixA(1024,1024), MatrixB(1024,1024)
Computing result using CUDA Kernel...
==37122== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
done122== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (0 of 2)...
Performance= 26.61 GFlop/s, Time= 80.697 msec, Size= 2147483648 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
==37122== Profiling application: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
==37122== Profiling result:
==37122== Metric result:
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla P100-SXM2-16GB (0)"					
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)					
301	flop_sp_efficiency	FLOP Efficiency(Peak Single)	22.96%	23.40%	23.15%

Visual Profiler

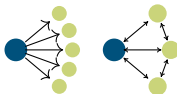
Your new favor



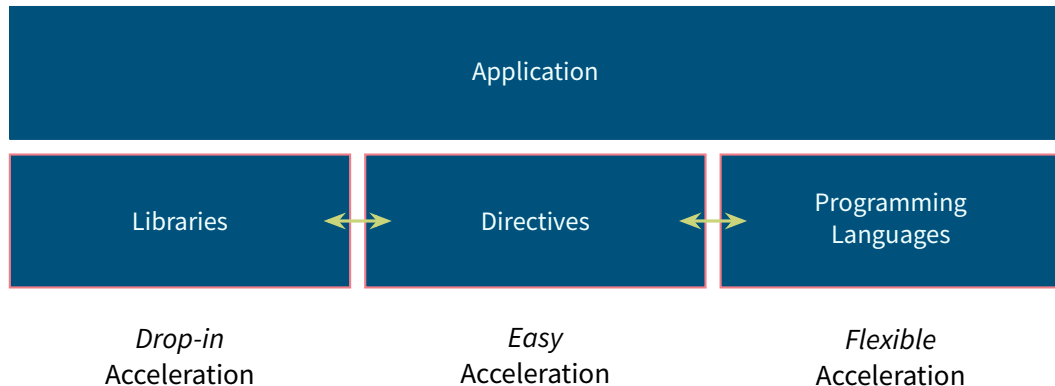
Advanced Topics

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**



Summary of Acceleration Possibilities



Using GPUs on JURECA & JUWELS

Compiling

CUDA

- Module: `module load CUDA/9.2.88`
- 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/18.7-GCC-7.3.0`
- Compile: `pgc++ -acc -ta=tesla file.cpp`

MPI

- Module: `module load MVAPICH2/2.3-GDR` (also needed: `GCC/7.3.0`)
Enabled for CUDA (*CUDA-aware*); no need to copy data to host before transfer

Running

- Dedicated GPU partitions

JUWELS

`--partition=gpus` 48 nodes (Job limits: <1 d)

JURECA

`--partition=gpus` 70 nodes (Job limits: <1 d, ≤ 32 nodes)

`--partition=develgpus` 4 nodes (Job limits: <2 h, ≤ 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 1 - 3 April 2019

OpenACC Course 28 - 29 October 2019

- Generally: see [online documentation](#) and sc@fz-juelich.de
- Further consultation via our lab: NVIDIA Application Lab in Jülich
- Interested in JURON? [Get access!](#)

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

APPENDIX

Appendix

Glossary

References

Glossary I

API A programmatic interface to software by well-defined functions. Short for application programming interface. [41](#), [46](#)

CUDA Computing platform for [GPUs](#) from NVIDIA. Provides, among others, CUDA C/C++. [37](#), [46](#), [47](#), [48](#), [50](#), [60](#), [63](#), [68](#)

DSL A Domain-Specific Language is a specialization of a more general language to a specific domain. [2](#), [49](#), [50](#)

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. [2](#), [63](#), [66](#)

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. [2](#), [4](#), [59](#), [61](#)

Glossary II

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](#), [59](#), [61](#)

MPI The Message Passing Interface, a API definition for multi-node computing. [57](#), [60](#)

NVIDIA US technology company creating [GPUs](#). [3](#), [4](#), [5](#), [46](#), [53](#), [63](#), [66](#), [67](#), [68](#), [69](#)

NVLink **NVIDIA**'s communication protocol connecting [CPU](#) ↔ [GPU](#) and [GPU](#) ↔ [GPU](#) with high bandwidth. [5](#), [68](#)

OpenACC Directive-based programming, primarily for many-core machines. [42](#), [43](#), [44](#), [50](#), [60](#), [63](#)

Glossary III

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (**CPU**, **GPU**, DSP, FPGA). The alternative to **CUDA**. 46, 53

OpenMP Directive-based programming, primarily for multi-threaded machines. 42

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). 68

POWER **CPU** architecture from IBM, earlier: PowerPC. See also POWER8. 68

POWER8 Version 8 of IBM's **POWER**processor, available also under the OpenPOWER Foundation. 5, 68

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 30, 48

Glossary IV

- Tesla** The GPU product line for general purpose computing computing of NVIDIA. 3, 4, 5
- Thrust** A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. 37
- Volta** GPU architecture from NVIDIA (announced 2017). 26
- CPU** Central Processing Unit. 3, 4, 5, 10, 11, 14, 15, 16, 17, 18, 22, 23, 24, 25, 30, 42, 46, 67, 68
- 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, 31, 32, 33, 36, 40, 41, 42, 45, 46, 48, 49, 52, 53, 57, 59, 61, 62, 63, 66, 67, 68, 69

Glossary V

HBP Human Brain Project. 67

SIMD Single Instruction, Multiple Data. 22, 23, 24, 25

SIMT Single Instruction, Multiple Threads. 12, 13, 19, 21, 22, 23, 24, 25

SM Streaming Multiprocessor. 22, 23, 24, 25, 26

SMT Simultaneous Multithreading. 22, 23, 24, 25

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 31, 32, 36).

References: Images, Graphics I

- [1] Alexandre Debiève. *Bowels of computer*. Freely available at Unsplash. URL: <https://unsplash.com/photos/F07JIlwj0tU>.
- [3] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (page 10).
- [4] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (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>.