GPU ACCELERATORS AT JSC
OF THREADS AND KERNELS

23 November 2018 | Andreas Herten | 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
  OpenACC/OpenMP
  CUDA C/C++
  Performance Analysis
  Advanced Topics

Using GPUs on JURECA & JUWELS
  Compiling
  Resource Allocation
JUWELS – Jülich’s New Large System

- 2500 nodes with Intel Xeon CPUs ($2 \times 24$ 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

Theoretical Peak Performance, Double Precision

End of Year

10^2

10^3

10^4

2008
2010
2012
2014
2016

INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis

Member of the Helmholtz Association

23 November 2018
Slide 7/41
Status Quo Across Architectures

Memory Bandwidth

Theoretical Peak Memory Bandwidth Comparison

- INTEL Xeon CPUs
- NVIDIA Tesla GPUs
- AMD Radeon GPUs
- INTEL Xeon Phis

Graphic: Rupp
CPU vs. GPU

A matter of specialties

Transporting one

Transporting many
CPU vs. GPU

Chip

Control

ALU

ALU

ALU

ALU

Cache

DRAM

DRAM
GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory
Aim: Hide Latency

*Everything else follows*

SIMT

Asynchronicity

Memory
Memory

GPU memory ain't no CPU memory

- 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
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

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

\[ \text{SIMT} = \text{SIMD} \oplus \text{SMT} \]

- **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

![Diagram of SIMT and SMT concepts](image)
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

Tesla V100

Graphics: Nvidia Corporation

Member of the Helmholtz Association
23 November 2018
Slide 16/41
SIMT

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

Vector

Addition

Member of the Helmholtz Association 23 November 2018 Slide 16/41
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**

- \( T_1 \)
- \( T_2 \)
- \( T_3 \)
- \( T_4 \)

**GPU Streaming Multiprocessor: High Throughput**

- \( W_1 \)
- \( W_2 \)
- \( W_3 \)
- \( W_4 \)

Legend:
- Blue: Thread/Warp
- Green: Processing
- Orange: Context Switch
- Purple: Ready
- Light Blue: Waiting
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

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

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

saxpy(n, a, x, y);
```
Programming GPUs
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
- OpenCV
- ArrayFire
- Thrust
- Numba
- theano

Member of the Helmholtz Association

23 November 2018

Slide 22/41
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

```c
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

```c
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
OpenACC/OpenMP
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
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);
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…

- 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

```c
__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];
}
```

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

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.19%</td>
<td>262.43ms</td>
<td>301</td>
<td>871.86us</td>
<td>863.88us</td>
<td>882.44us</td>
<td>void matrixMulCUDA&lt;int=32&gt;(float*, float*, float*, int, int)</td>
</tr>
<tr>
<td>0.58%</td>
<td>1.5428ms</td>
<td>2</td>
<td>771.39us</td>
<td>764.65us</td>
<td>778.12us</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>0.23%</td>
<td>599.40us</td>
<td>1</td>
<td>599.40us</td>
<td>599.40us</td>
<td>599.40us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
</tbody>
</table>

==37064== API calls:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>61.26%</td>
<td>258.38ms</td>
<td>1</td>
<td>258.38ms</td>
<td>258.38ms</td>
<td>258.38ms</td>
<td>cudaEventSynchronize</td>
</tr>
<tr>
<td>35.68%</td>
<td>150.49ms</td>
<td>3</td>
<td>50.164ms</td>
<td>914.97us</td>
<td>148.65ms</td>
<td>cudaMalloc</td>
</tr>
<tr>
<td>0.73%</td>
<td>3.0774ms</td>
<td>3</td>
<td>1.0258ms</td>
<td>1.0097ms</td>
<td>1.0565ms</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>0.62%</td>
<td>2.6287ms</td>
<td>4</td>
<td>657.17us</td>
<td>655.12us</td>
<td>660.56us</td>
<td>cuDeviceTotalMem</td>
</tr>
<tr>
<td>0.56%</td>
<td>2.3408ms</td>
<td>301</td>
<td>7.7760us</td>
<td>7.3810us</td>
<td>53.103us</td>
<td>cudaLaunch</td>
</tr>
<tr>
<td>0.48%</td>
<td>2.0111ms</td>
<td>364</td>
<td>5.5250us</td>
<td>235ns</td>
<td>201.63us</td>
<td>cudaDeviceGetAttribute</td>
</tr>
<tr>
<td>0.21%</td>
<td>872.52us</td>
<td>1</td>
<td>872.52us</td>
<td>872.52us</td>
<td>872.52us</td>
<td>cudaDeviceSynchronize</td>
</tr>
<tr>
<td>0.15%</td>
<td>612.20us</td>
<td>1505</td>
<td>406ns</td>
<td>361ns</td>
<td>1.1970us</td>
<td>cudaSetupArgument</td>
</tr>
<tr>
<td>0.12%</td>
<td>499.01us</td>
<td>3</td>
<td>166.34us</td>
<td>140.45us</td>
<td>216.16us</td>
<td>cudaFree</td>
</tr>
</tbody>
</table>
Visual Profiler

NVIDIA Visual Profiler

*NewSession1*

- Process "matrixMul" (18924)
- Thread 39720768
- Runtime API
- Driver API
- Profiling Overhead
- [0] Tesla K40m
- Context 1 (CUDA)
  - MemCpy (HtoD)
  - MemCpy (DtoH)
  - Compute
    - 100,0% void math...
  - Streams
- Default

Analysis | Details | Console | Settings

<terminated> matrixMul on jujhydra

Matrix Multiply Using CUDA | Starting...
GPU Device 0: "Tesla K40m" with compute capability 3.5
Matrix A(320,320), Matrix B(640,320)
Computing result using CUDA Kernel... done
Performance: 351.01 GFlop/s, Time: 0.373 msec, Size= 13107200 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

Note: The CUDA Profiler can be used for performance measurement. Results may vary when CPU Profiler is used.
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/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, \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  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
API  A programmatic interface to software by well-defined functions. Short for application programming interface. 40

CUDA  Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 39, 40, 41, 42, 49, 52, 56

JSC  Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 2, 52, 55

JURECA  A multi-purpose supercomputer with 1800 nodes at JSC. 2, 4, 48, 50

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, 48, 50
Glossary II

MPI The Message Passing Interface, a API definition for multi-node computing. 47, 49

NVIDIA US technology company creating GPUs. 3, 4, 5, 40, 44, 52, 55, 56, 57

NVLink NVIDIA’s communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 5, 56

OpenACC Directive-based programming, primarily for many-core machines. 2, 35, 36, 37, 38, 49, 52

OpenCL The Open Computing Language. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 40, 44

OpenMP Directive-based programming, primarily for multi-threaded machines. 2, 35, 36

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

Pascal  GPU architecture from NVIDIA (announced 2016). 56

POWER  CPU architecture from IBM, earlier: PowerPC. See also POWER8. 57

POWER8  Version 8 of IBM’s POWER processor, available also under the OpenPOWER Foundation. 5, 57

SAXPY  Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 28, 42

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

CPU  Central Processing Unit. 3, 4, 5, 10, 11, 14, 15, 16, 17, 18, 22, 23, 24, 28, 40, 56, 57
Glossary IV

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, 32, 35, 36, 39, 40, 42, 43, 44, 47, 48, 50, 51, 52, 55, 56, 57

HBP  Human Brain Project. 55

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

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


