GPU ACCELERATORS AT JSC
OF THREADS AND KERNELS

21 May 2019 | 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 × 24 cores)
- 46 + 10 nodes with 4 NVIDIA Tesla V100 cards
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance (Top500: #26)
**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 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
JURON – A Human Brain Project *Pilot System*

- 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

Performance

Theoretical Peak Performance, Double Precision

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

Member of the Helmholtz Association
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
CPU vs. GPU

A matter of specialties

Transporting one

Transporting many
CPU vs. GPU

Chip
GPU Architecture

Overview

Aim: Hide Latency

*Everything else follows*
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

- GPU: accelerator / extension card
  → Separate device from CPU

Separate memory, but UVA!
Memory transfers needs special consideration!
Do as little as possible!
Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance…?)

P100: 16GBRAM, 720GB/s;
V100: 16(32)GBRAM, 900GB/s
Memory

GPU memory ain’t no CPU memory

- GPU: accelerator / extension card
- Separate device from CPU
- Separate memory, but UVA

Unified Virtual Addressing

Memory transfers need special consideration!
Do as little as possible!

Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance…?)

P100: 16GB RAM, 720GB/s
V100: 16(32)GB RAM, 900GB/s
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…?)

P100: 16 GB RAM, 720 GB/s
V100: 16 (32) GB RAM, 900 GB/s
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!*

---

Member of the Helmholtz Association
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…?)

Host

Device

Unified Memory
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…?)

**Host**

- Control
- ALU
- ALU
- ALU
- Cache
- DRAM
- NVLink ≈ 80 GB/s

**Device**

- HBM2 < 900 GB/s
- DRAM
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 → GPU → CPU

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

Member of the Helmholtz Association 21 May 2019 Slide 12/41
Processing Flow

CPU → GPU → CPU

1. Transfer data from CPU memory to GPU memory

Scheduler

Interconnect

L2

DRAM
**Processing Flow**

**CPU → GPU → CPU**

1. Transfer data from CPU memory to GPU memory, transfer program

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

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
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
GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory
SIMT

SIMT = SIMD ⊕ SMT

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

Scalar

\[
\begin{align*}
A_0 + B_0 &= C_0 \\
A_1 + B_1 &= C_1 \\
A_2 + B_2 &= C_2 \\
A_3 + B_3 &= C_3
\end{align*}
\]
SIMT

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

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

```
\[
\begin{array}{cccc}
A_0 & B_0 & C_0 \\
A_1 & B_1 & C_1 \\
A_2 & B_2 & C_2 \\
A_3 & B_3 & C_3 \\
\end{array}
\]
```

+ [Vector]

---

Member of the Helmholtz Association  21 May 2019  Slide 16/41
CPU:
- Single Instruction, Multiple Data (SIMD)
- Simultaneous Multithreading (SMT)
SIMT

SIMT = SIMD + SMT

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

SIMT = SIMD ⊕ SMT

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

SIMT = SIMD ⊕ SMT

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

**SIMT = SIMD ⊕ 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 $\text{if } \ldots$
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, an warp)

Fast switching of threads (large register file)

Branching

Tesla V100

Graphics: Nvidia Corporation

Vector

A₀ + B₀ = C₀
A₁ + B₁ = C₁
A₂ + B₂ = C₂
A₃ + B₃ = C₃

SIMT

Thread
Core
Thread
Core

Member of the Helmholtz Association
21 May 2019
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 if

Tesla V100 Graphics: Nvidia Corporation

Vector

\[ \begin{align*}
A_0 & + B_0 = C_0 \\
A_1 & + B_1 = C_1 \\
A_2 & + B_2 = C_2 \\
A_3 & + B_3 = C_3
\end{align*} \]

SIMT

Member of the Helmholtz Association
21 May 2019
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

Tesla V100 Multiprocessor Graphics: Nvidia Corporation

Vector

\[ \begin{align*}
A_0 &+ B_0 = C_0 \\
A_1 &+ B_1 = C_1 \\
A_2 &+ B_2 = C_2 \\
A_3 &+ B_3 = C_3
\end{align*} \]

SIMT

Member of the Helmholtz Association 21 May 2019 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
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**

[Diagram showing CPU core with threads T1 to T4, highlighting latency and processing stages.]
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**

---

Member of the Helmholtz Association 21 May 2019 Slide 17/41
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

Member of the Helmholtz Association  
21 May 2019  
Slide 18/41
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];
}

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!
Libraries

Programming GPUs is easy: Just don’t!

Use applications & libraries
Libraries

Programming GPUs is easy: Just don’t!

Use applications & libraries
Programming GPUs is easy: **Just don’t!**

**Use applications & libraries**

- cuBLAS
- cuSPARSE
- cuDNN
- cuFFT
- cuRAND
- CUDA Math
- OpenCV
- ArrayFire
- Thrust
- Numba
- theano
Libraries

Programming GPUs is easy: Just don’t!

Use applications & libraries

- cuBLAS
- cuSPARSE
- cuDNN
- cuFFT
- cuRAND
- CUDA Math
- OpenCV
- ArrayFire
- Thrust
- Numba
- theano
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);
```
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
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
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
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

Member of the Helmholtz Association
21 May 2019
Slide 24|41
Programming GPUs

OpenACC/OpenMP
GPU Programming with Directives

Keepin’ you portable

- Annotate serial source code by directives

```c
#pragma acc loop
for (int i = 0; i < 1; i++) {}
```
GPU Programming with Directives

Keepin’ you portable

- Annotate serial source code by directives
  
  ```c
  #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
  
  ```c
  #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:
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: CUDAFortran

Choose what flavor you like, what colleagues/collaboration is using.

Hardest: Come up with parallelized algorithm.
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
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
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:
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

Parallel function:

```c
__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 or order non-deterministic!
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 $\rightarrow$ Block
  - Block

Parallel function: __global__

```c
kernel(int a, float *b) {}```

Access own ID by global variables `threadIdx.x`, `blockIdx.y`, …

Execution entity: threads

Lightweight fast switching!

10000 threads execute simultaneously or in non-deterministic order!
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

Parallel function: kernel

```c
__global__ kernel (int a, float *b) { }
```

Access own ID by global variables `threadIdx.x`, `blockIdx.y`, …

Execution entity: threads

Lightweight fast switching!

1000 threads execute simultaneously or in non-deterministic order.
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

```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();
```
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();
```
__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

```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
tab int a = 42;
tab int n = 10;
tab 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

```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();
```
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();
```
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();
```
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.
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

```
$ 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 memcpyDtoH]

==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   cudaMemcpyGetAttribute
0.21%    872.52us    1   872.52us   872.52us   872.52us   cudaMemcpy
0.15%   612.20us    1505   406ns   361ns   1.1970us   cudaMemcpySynchronize
0.12%   499.01us    3   166.34us   140.45us   216.16us   cudaFree
```
Visual Profiler

The NVIDIA Visual Profiler is a tool used to profile CUDA applications. It provides detailed information about the execution of the application, including timing, memory usage, and other performance metrics. This helps developers identify bottlenecks and optimize their code for better performance.

In the image, the profiler is showing a session for the process "matrixMult" with a duration of 0.3 seconds. The timeline is divided into intervals and shows various CUDA events such as "cudaEventSynchronize" and "cudaMemcpy". The session also includes analysis details and properties, such as default settings for duration and session.

The example shown is a Matrix Multiplication using CUDA. The GPU device is "Tesla K40m" with compute capability 3.5. The performance is stated as 351.01 GFlop/s, time of 0.373 msec, size of 531072000 Ops, and workgroup size of 1824 threads/block. The result is marked as PASS for correctness.
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 MVAPICH2/2.3.1-GDR (also needed: GCC/8.3.0)
  Enabled for CUDA (CUDA-aware); no need to copy data to host before transfer
Running

- Dedicated GPU partitions
  
  **JUWELS**
  
  --partition=gpus  46 nodes (Job limits: <1 d)
  --partition=develgpus  10 nodes (Job limits: <2 h, ≤ 2 nodes)

  **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

```bash
#!/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!
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 2020
  - OpenACC Course  28 - 29 October 2019
- 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 2020
  - 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; contact me!
GPUs provide highly-parallel computing power
We have many devices installed at JSC, ready to be used!
Training courses by JSC
  CUDA Course  April 2020
  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; contact me!
Interested in JURON? Get access!
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 2020
  - 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
Glossary I

**API** A programmatic interface to software by well-defined functions. Short for application programming interface. 72, 73, 74, 75, 76

**CUDA** Computing platform for **GPUs** from NVIDIA. Provides, among others, CUDA C/C++. 2, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 100, 103, 104, 105, 106, 107, 111

**JSC** Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 2, 103, 104, 105, 106, 107, 110

**JURECA** A multi-purpose supercomputer with 1800 nodes at JSC. 2, 5, 99, 101

**JURON** One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. 6, 7

**JUWELS** Jülich’s new supercomputer, the successor of JUQUEEN. 2, 3, 4, 99, 101
Glossary II

**MPI**  The Message Passing Interface, a API definition for multi-node computing. 98, 100

**NVIDIA**  US technology company creating GPUs. 3, 4, 5, 6, 7, 72, 73, 74, 75, 76, 94, 95, 103, 104, 105, 106, 107, 110, 111, 112, 113

**NVLink**  NVIDIA’s communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 6, 7, 112

**OpenACC**  Directive-based programming, primarily for many-core machines. 2, 65, 66, 67, 68, 69, 70, 100, 103, 104, 105, 106, 107

**OpenCL**  The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 72, 73, 74, 75, 76, 94, 95
OpenMP  Directive-based programming, primarily for multi-threaded machines.  2, 65, 66, 67, 68

P100  A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast HBM2 memory.  6, 7

Pascal  GPU architecture from NVIDIA (announced 2016).  112

POWER  CPU architecture from IBM, earlier: PowerPC. See also POWER8.  112

POWER8  Version 8 of IBM’s POWER processor, available also under the OpenPOWER Foundation.  6, 7, 112

SAXPY  Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset.  50, 86, 87, 88, 89, 90, 91, 92
Glossary IV

**Tesla**  The GPU product line for general purpose computing computing of NVIDIA. 3, 4, 5, 6, 7

**CPU**  Central Processing Unit. 3, 4, 5, 6, 7, 12, 13, 14, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 50, 72, 73, 74, 75, 76, 111, 112

**GPU**  Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 49, 51, 52, 53, 54, 55, 56, 57, 65, 66, 67, 68, 71, 72, 73, 74, 75, 76, 90, 91, 92, 93, 94, 95, 98, 99, 101, 102, 103, 104, 105, 106, 107, 110, 111, 112, 113

**HBP**  Human Brain Project. 110

**SIMD**  Single Instruction, Multiple Data. 35, 36, 37, 38, 39, 40, 41, 42, 43, 44
**Glossary V**

**SIMT**  Single Instruction, Multiple Threads. 15, 16, 17, 30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44

**SM**  Streaming Multiprocessor. 35, 36, 37, 38, 39, 40, 41, 42, 43, 44

**SMT**  Simultaneous Multithreading. 35, 36, 37, 38, 39, 40, 41, 42, 43, 44

References: Images, Graphics I


