JÜLICH
FORSCHUNGSZENTRUM

GPU Programming 101

GridKa School 2017: `make science && run`

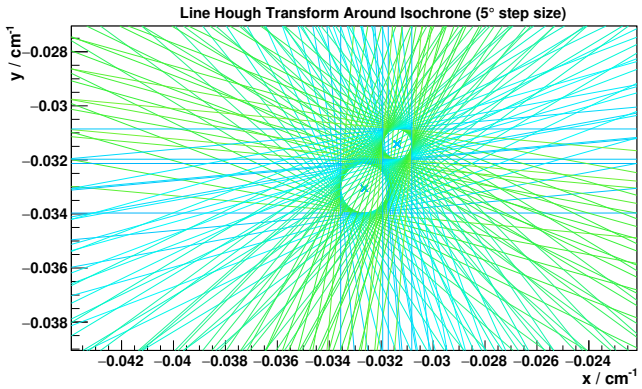
Andreas Herten, Forschungszentrum Jülich, 31 August 2017

About, Outline

JÜLICH FORSCHUNGSZENTRUM

Andreas Herten

- Physics in
  - Aachen (Dipl. at CMS)
  - Jülich/Bochum (Dr. at PANDA)



- Since then: NVIDIA Application Lab
  Optimizing scientific applications for/on GPUs at Jülich Supercomputing Centre

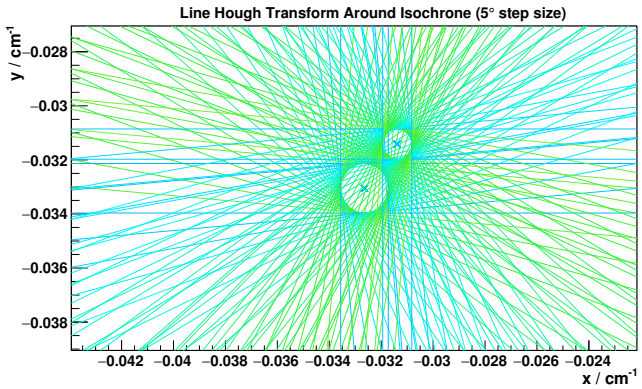
JÜLICH APPLICATION LAB NVIDIA

About, Outline

JÜLICH FORSCHUNGSZENTRUM

Andreas Herten

- Physics in
  - Aachen (Dipl. at CMS)
  - Jülich/Bochum (Dr. at PANDA)



- Since then: NVIDIA Application Lab
  Optimizing scientific applications for/on GPUs at Jülich Supercomputing Centre

JÜLICH APPLICATION LAB NVIDIA

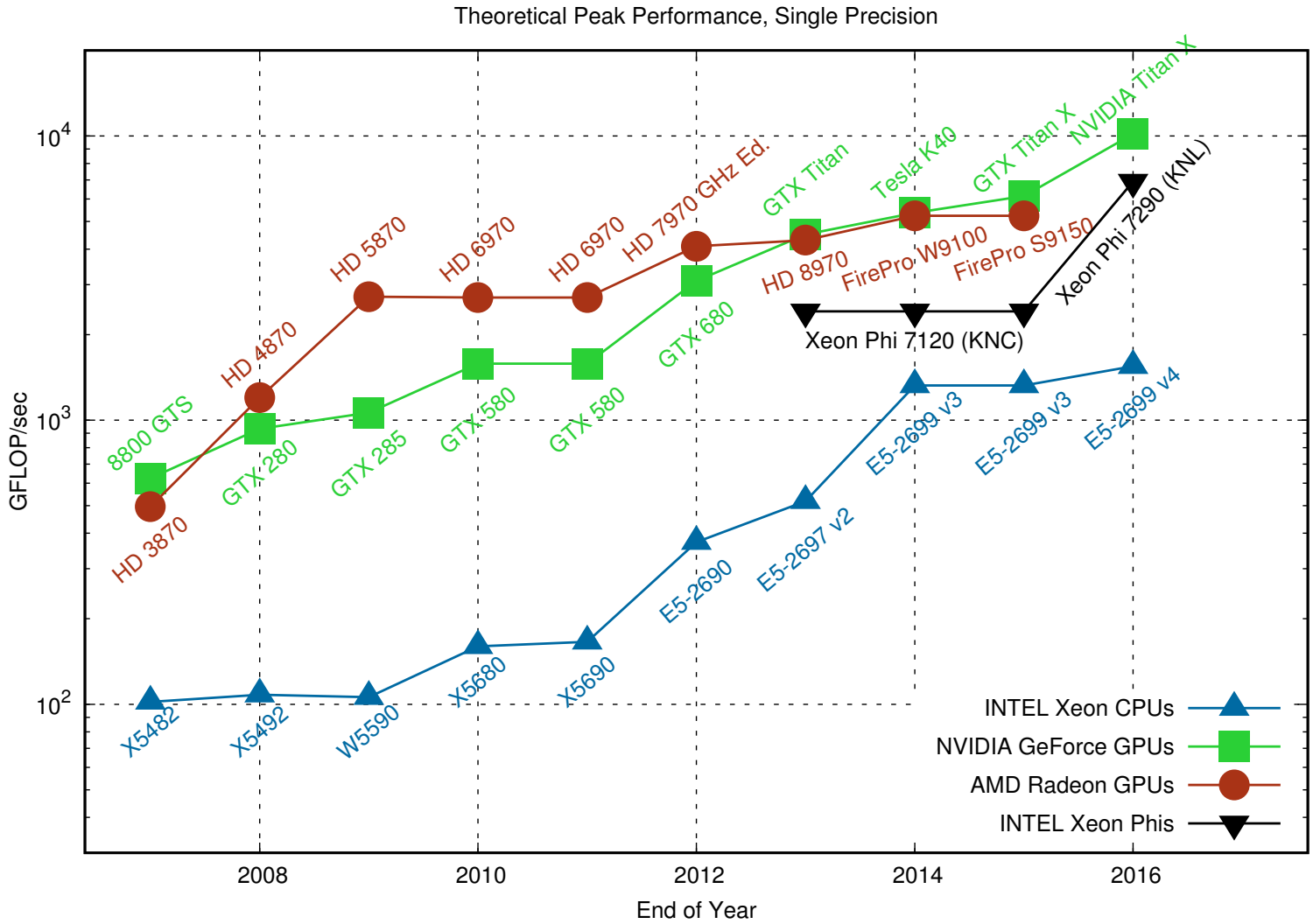
Status Quo

GPU all around

- 1999: General computations with shaders of *graphics hardware*
- 2001: NVIDIA GeForce 3 with programmable shaders [1]; 2003: DirectX 9 at ATI
- 2007: CUDA
- 2017: Top 500: 15 % with GPUs, Green 500: 9 of 10 of top 10 with GPUs

Status Quo

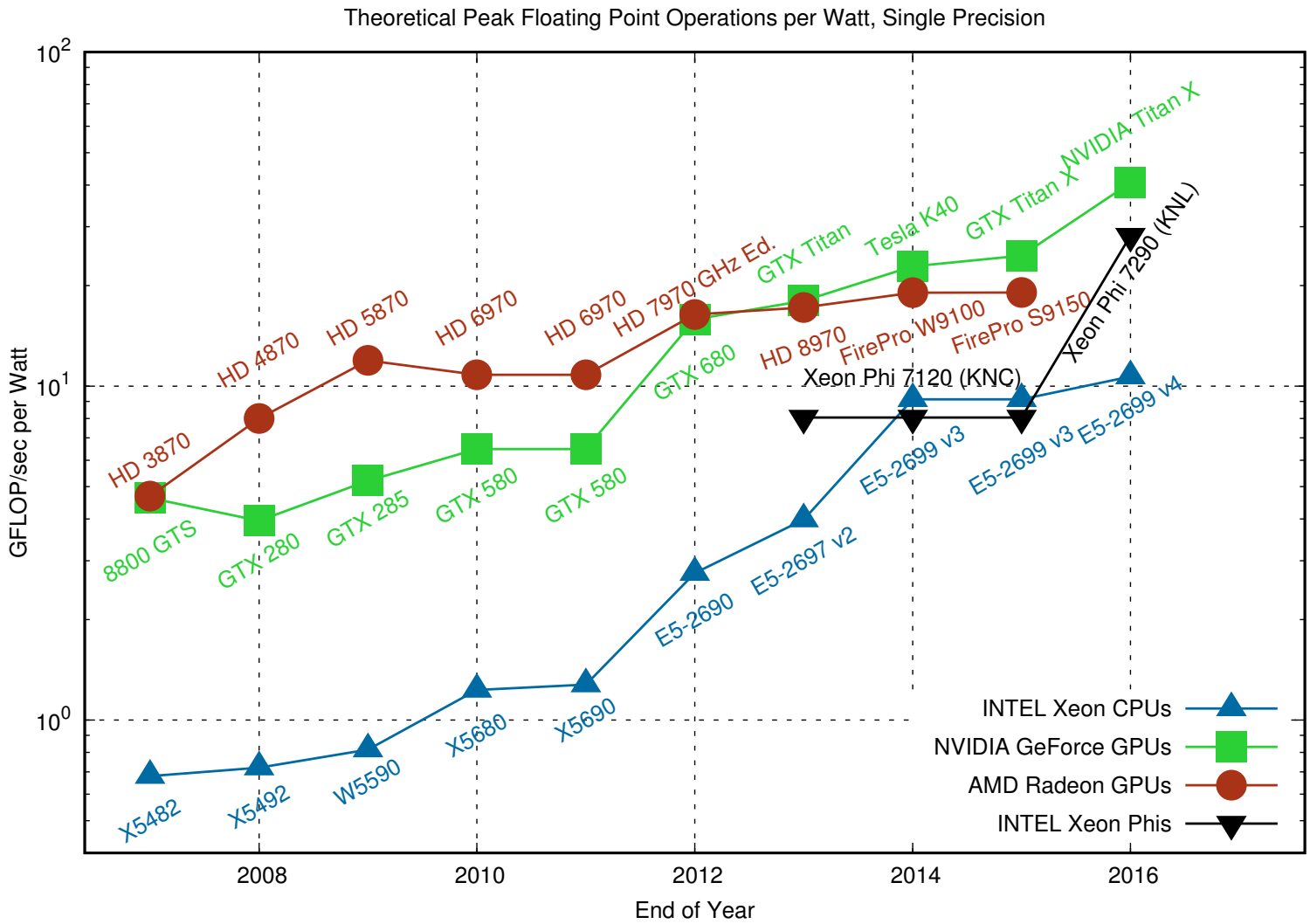
Peak performance single precision



Graphic: Rupp [2]

Status Quo

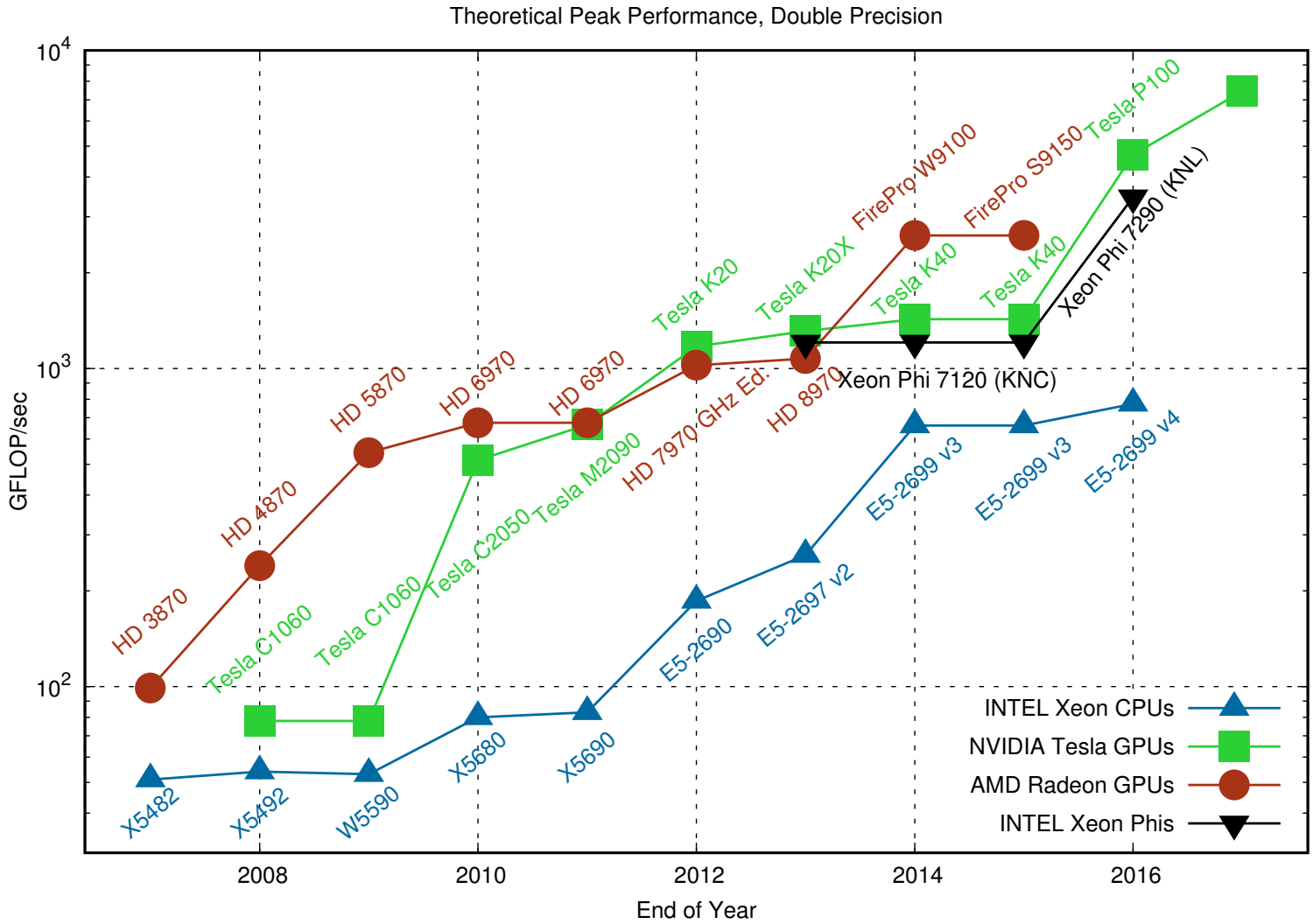
Performance per Watt



Graphic: Rupp [2]

Status Quo

Peak performance double precision



Graphic: Rupp [2]


JÜLICH
JURECA

But why?!

But why?!

Let's find out!

JÜLICH
FORSCHUNGSZENTRUM

Platform

CPU vs. GPU

A matter of specialties

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

CPU vs. GPU

A matter of specialties

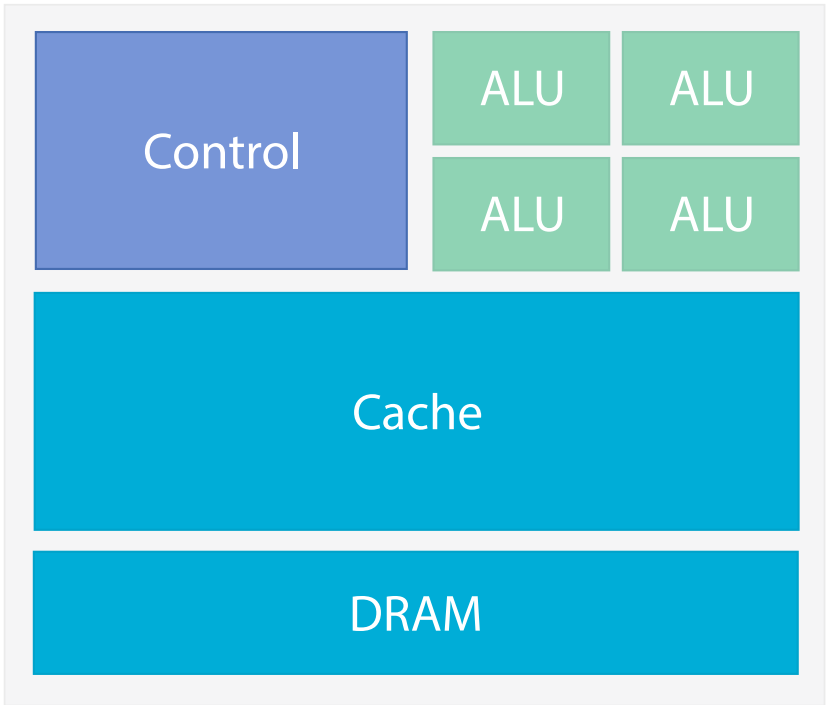
Transporting one

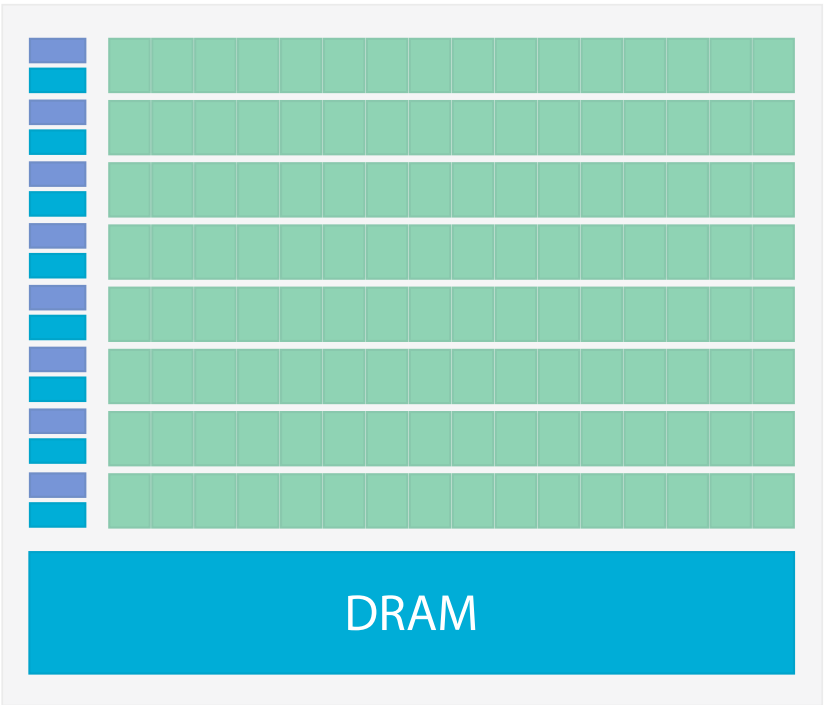
Transporting many

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

CPU vs. GPU

Chip





JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

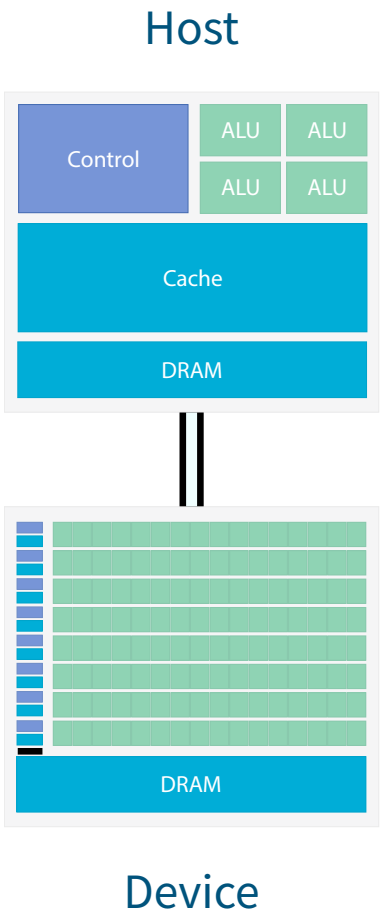
Memory

Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU

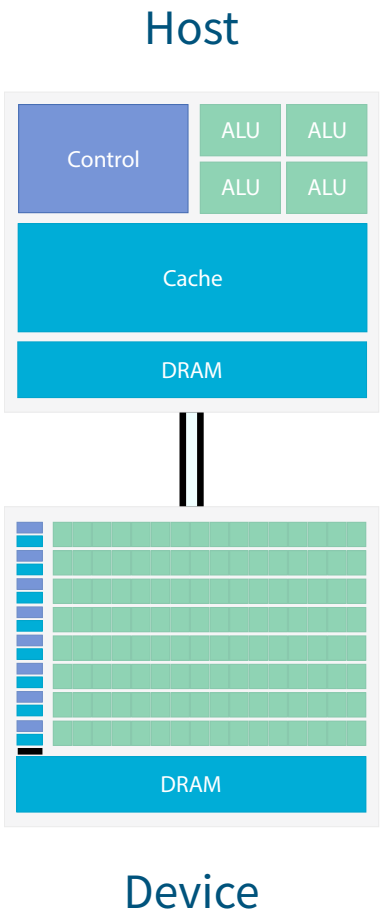


Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU
Separate memory, but UVA

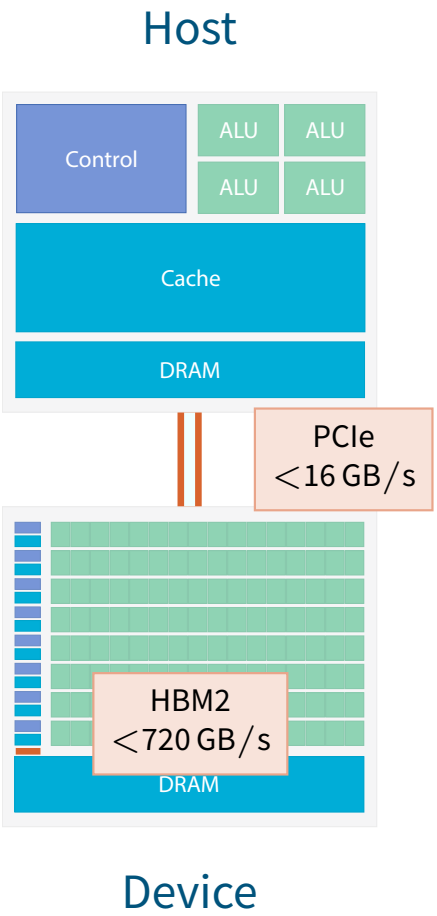


Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU
Separate memory, but UVA



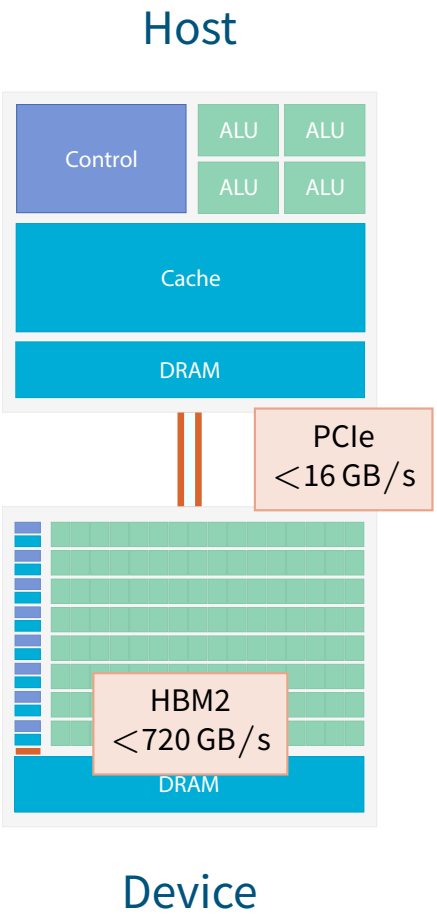
Memory

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!



Memory

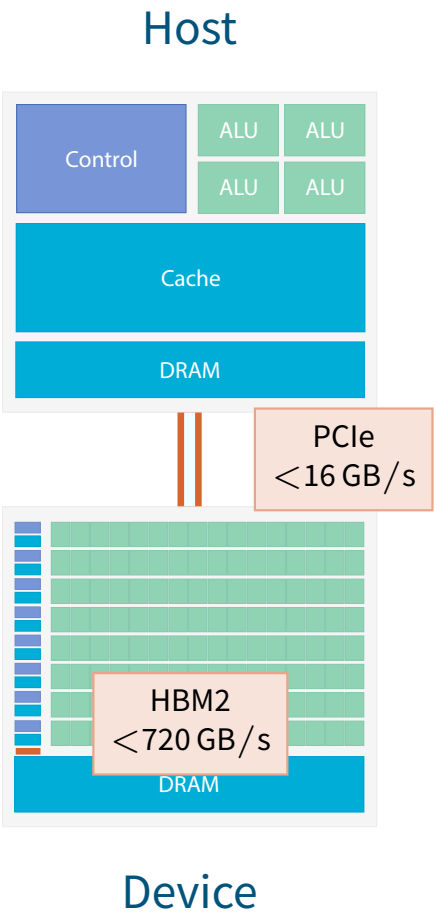
GPU memory ain't no CPU memory

JÜLICH FORSCHUNGSZENTRUM

- 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…?)



Memory

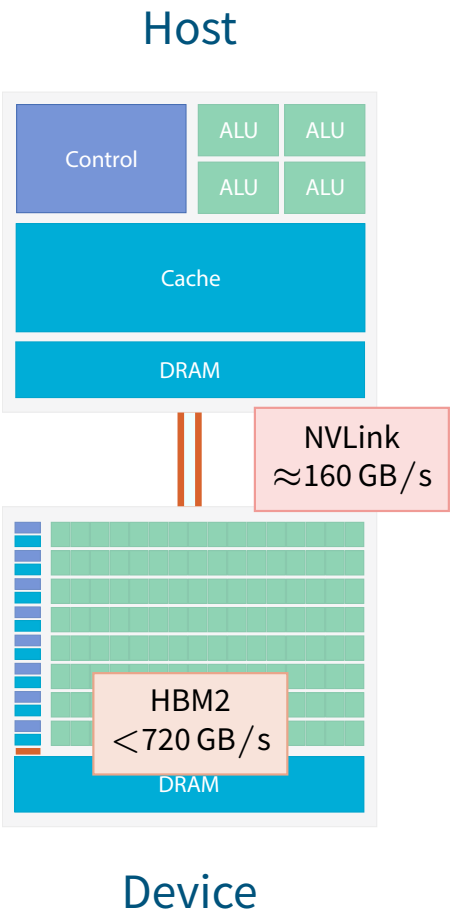
GPU memory ain't no CPU memory

JÜLICH FORSCHUNGSZENTRUM

- 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…?)



Memory

GPU memory ain't no CPU memory

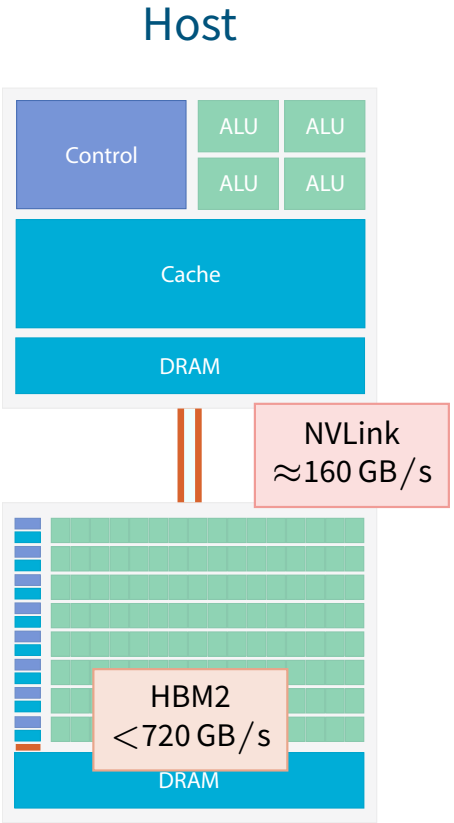
- GPU: accelerator / extension card

→ 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…?)
- Example values

P100

16 GB RAM, 720 GB/s



Memory

GPU memory ain't no CPU memory

JÜLICH FORSCHUNGSZENTRUM

- GPU: accelerator / extension card

→ Separate device from CPU
Separate memory, but UVA and UM

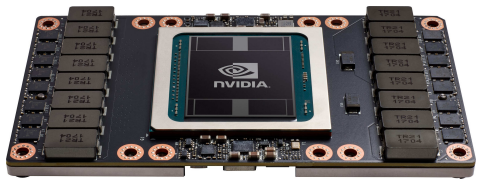
- Memory transfers need special consideration!
Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance…?)
- Example values

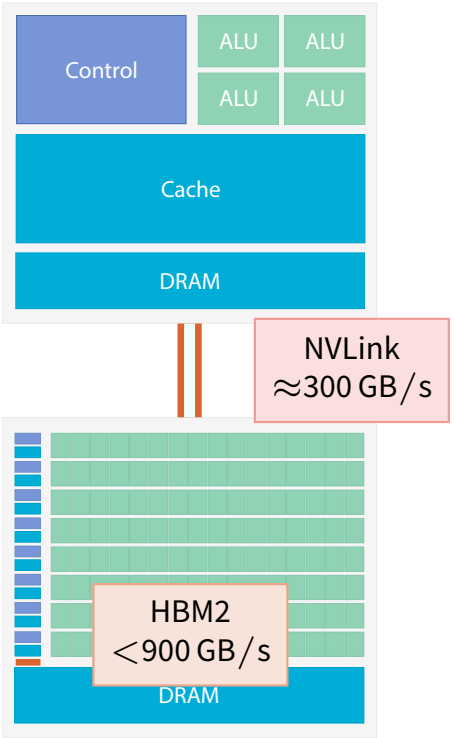
P100

16 GB RAM, 720 GB/s

V100

16 GB RAM, 900 GB/s





JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Async

Following different streams

JÜLICH FORSCHUNGSZENTRUM

- 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
- Also: Fast switching of contexts to keep GPU busy.

JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

JÜLICH FORSCHUNGSZENTRUM

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

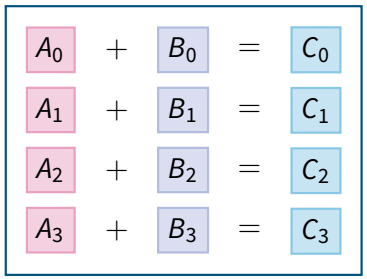
Memory

SIMT

Of threads and warps

JÜLICH FORSCHUNGSZENTRUM

Scalar



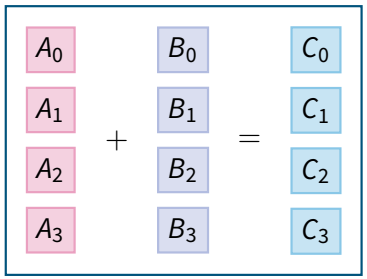
- CPU:
  - Single Instruction, Multiple Data (SIMD)

SIMT

Of threads and warps

JÜLICH FORSCHUNGSZENTRUM

Vector



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

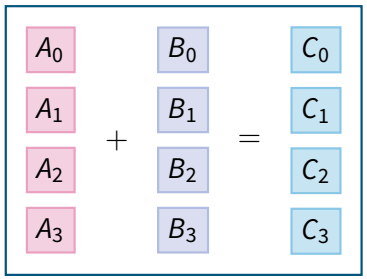
SIMT

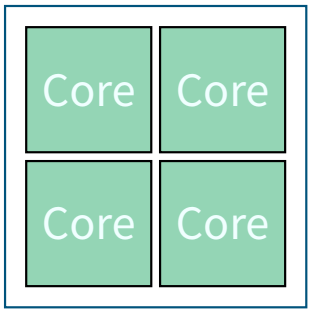
Of threads and warps

JÜLICH FORSCHUNGSZENTRUM

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

Vector





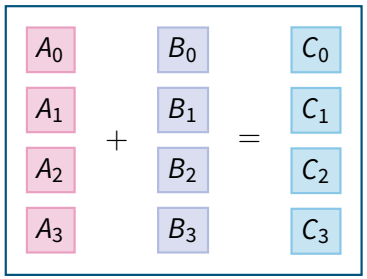
SIMT

Of threads and warps

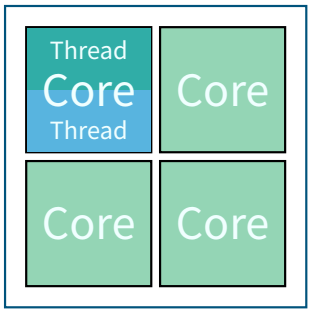
JÜLICH FORSCHUNGSZENTRUM

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

Vector



SMT



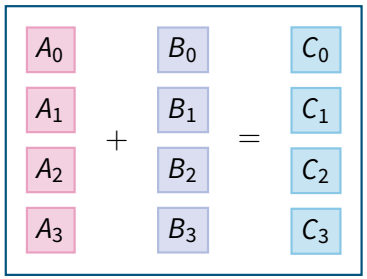
SIMT

Of threads and warps

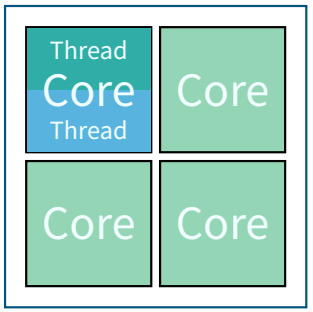
JÜLICH FORSCHUNGSZENTRUM

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

Vector



SMT



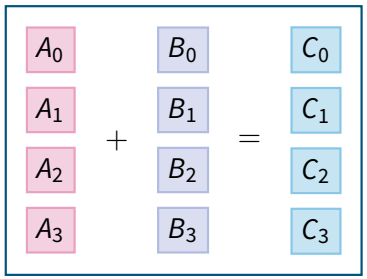
SIMT

Of threads and warps

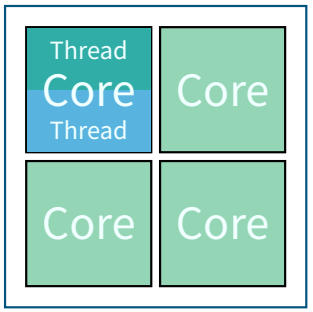
JÜLICH FORSCHUNGSZENTRUM

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

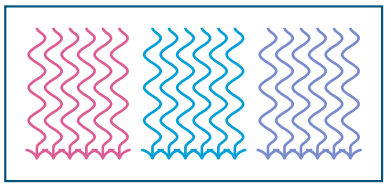
Vector



SMT



SIMT



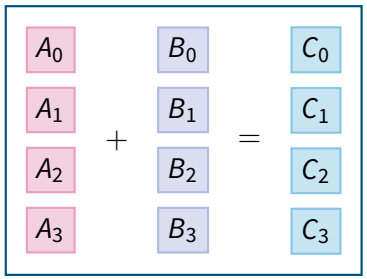
SIMT

Of threads and warps

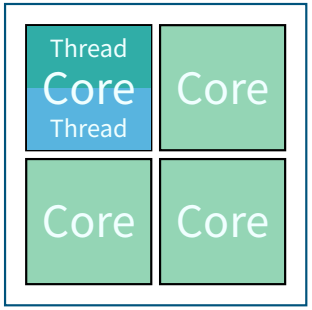
JÜLICH FORSCHUNGSZENTRUM

- 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 

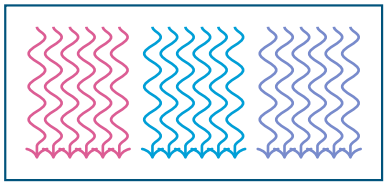
Vector



SMT



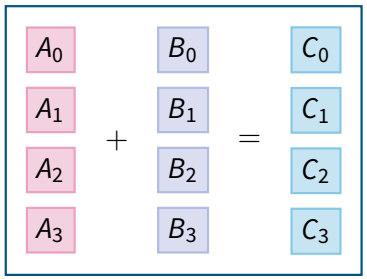
SIMT



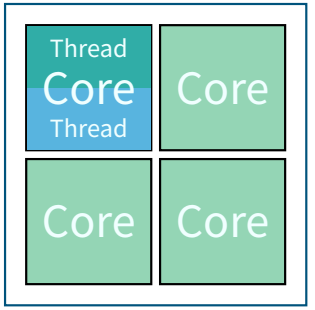
SIMT

Of threads and warps

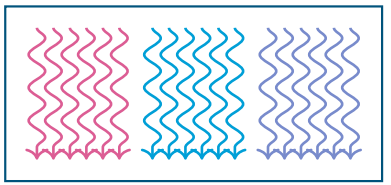
Vector



SMT



SIMT

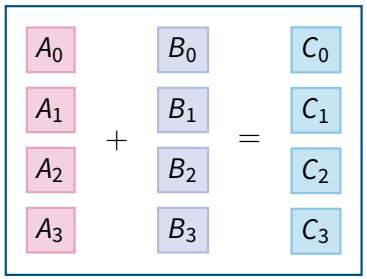


Graphics: Nvidia Corporation [5]

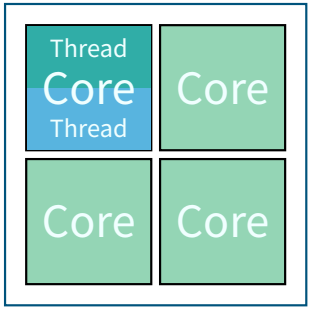
SIMT

Of threads and warps

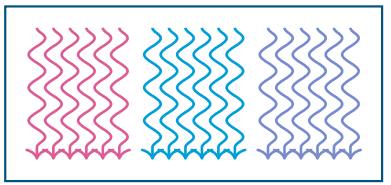
Vector



SMT



SIMT



Graphics: Nvidia Corporation [5]

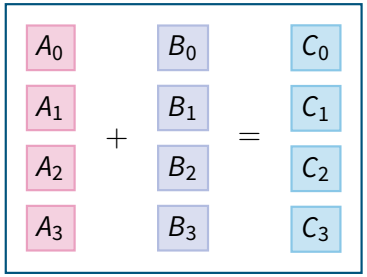
SIMT

Of threc

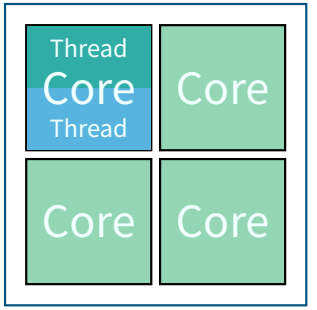
Multiprocessor

JÜLICH FORSCHUNGSZENTRUM

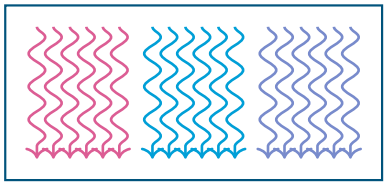
Vector



SMT



SIMT

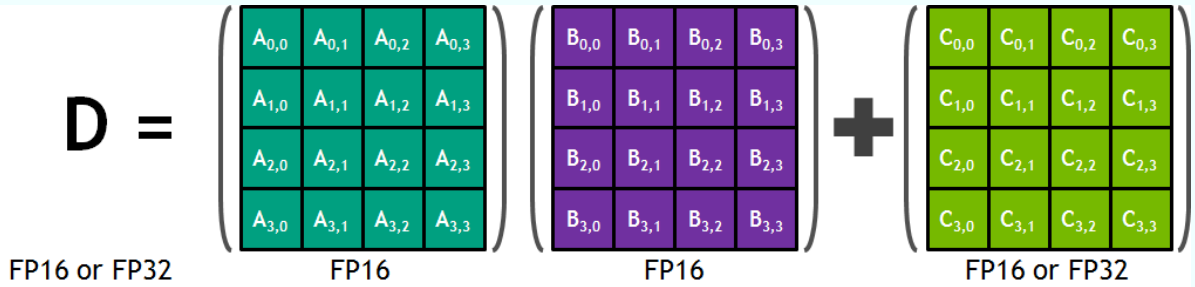


Graphics: Nvidia Corporation [5]

SIMT

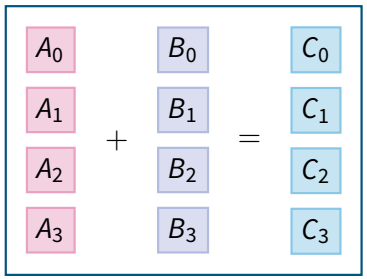
Of threc

Tensor Cores

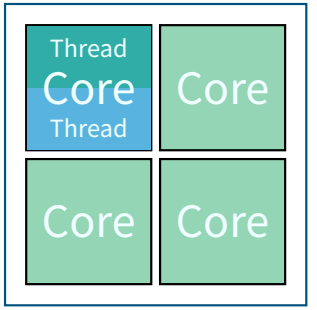


120 PFLOP/s for Deep Learning

Vector

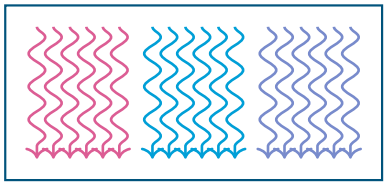


SMT



Graphics: Nvidia Corporation [5]

SIMT



Low Latency vs. High Throughput

Maybe GPU's ultimate feature

JÜLICH FORSCHUNGSZENTRUM

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

JÜLICH FORSCHUNGSZENTRUM

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency

Thread/Warp
Processing
Context Switch
Ready
Waiting

Low Latency vs. High Throughput

Maybe GPU's ultimate feature

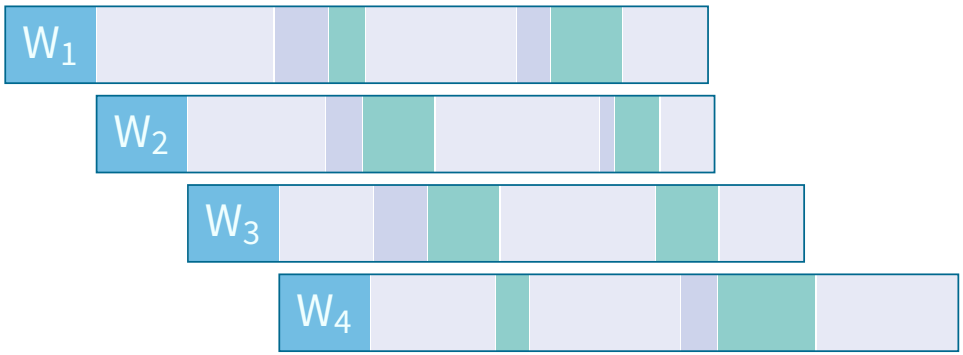
JÜLICH FORSCHUNGSZENTRUM

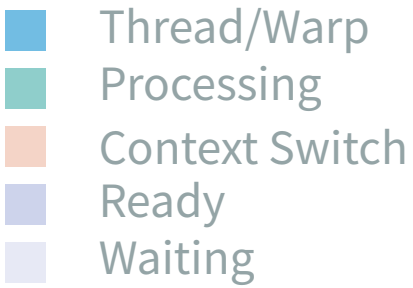
CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

CPU Core: Low Latency

GPU Streaming Multiprocessor: High Throughput

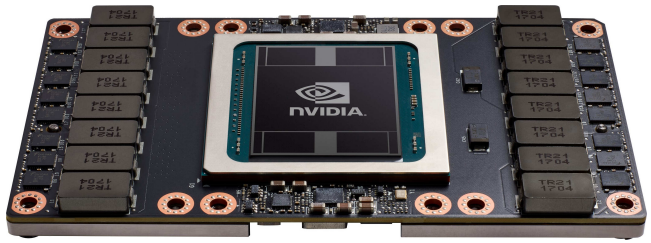




CPU vs. GPU

Let's summarize this!

JÜLICH FORSCHUNGSZENTRUM



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

JÜLICH
FORSCHUNGSZENTRUM

Programming GPUs

Preface: CPU

A simple CPU program as reference!

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];
}

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

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

Libraries

The truth is out there!

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

Libraries

The truth is out there!

JÜLICH FORSCHUNGSZENTRUM

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

Use applications & libraries!

Libraries

The truth is out there!

JÜLICH FORSCHUNGSZENTRUM

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

Use applications & libraries!

Wizard: Breazell [6]

Libraries

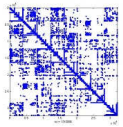
The truth is out there!

JÜLICH FORSCHUNGSZENTRUM

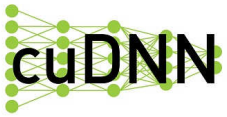
Programming GPUs is easy: **Just don't!**

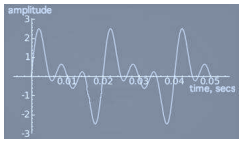
Use applications & libraries!

cuBLAS

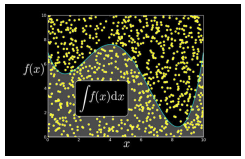


cuSPARSE

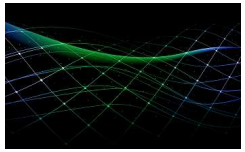




cuFFT



cuRAND



CUDA Math

Numba

t heano

Wizard: Breazell [6]

Member of the Helmholtz Association

Libraries

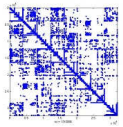
The truth is out there!

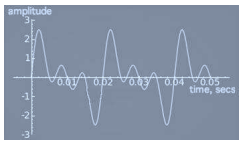
JÜLICH FORSCHUNGSZENTRUM

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

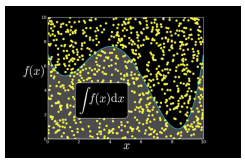
Use applications & libraries!

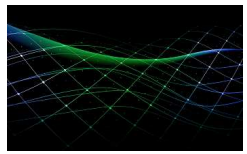

cuBLAS


cuSPARSE



cuFFT


cuRAND


CUDA Math

Numba

t heano

Wizard: Breazell [6]

cuBLAS

Parallel algebra

JÜLICH
FORSCHUNGSZENTRUM

- 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

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

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

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

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

cuBLAS

Code example

JÜLICH FORSCHUNGSZENTRUM

```
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

Libraries

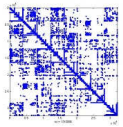
The truth is out there!

JÜLICH FORSCHUNGSZENTRUM

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

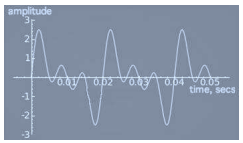
Use applications & libraries!


cuBLAS

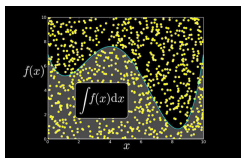

cuSPARSE

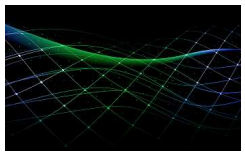
OpenCV

cuDNN



cuFFT


cuRAND


CUDA Math

Thrust

Numba

t heano

Wizard: Breazell [6]

Libraries

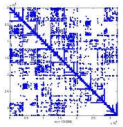
The truth is out there!

JÜLICH FORSCHUNGSZENTRUM

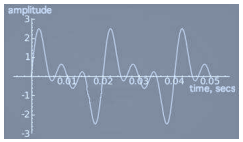
Programming GPUs is easy: **Just don't!**

Use applications & libraries!

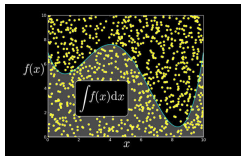

cuBLAS


cuSPARSE

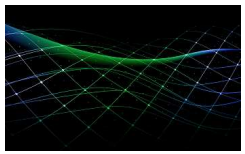
Numba



cuFFT



cuRAND


CUDA Math

t heano

Wizard: Breazell [6]

Thrust

Iterators! Iterators everywhere! 🚀

JÜLICH FORSCHUNGSZENTRUM

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- 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

JÜLICH FORSCHUNGSZENTRUM

```
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

JÜLICH FORSCHUNGSZENTRUM

```
#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

JÜLICH
FORSCHUNGSZENTRUM

Programming GPUs

Directives

GPU Programming with Directives

Keepin' you portable

JÜLICH FORSCHUNGSZENTRUM

- Annotate usual source code by directives

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

GPU Programming with Directives

Keepin' you portable

JÜLICH FORSCHUNGSZENTRUM

- 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

GPU Programming with Directives

Keepin' you portable

JÜLICH FORSCHUNGSZENTRUM

- 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.

JÜLICH FORSCHUNGSZENTRUM

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
Might eventually be re-merged into OpenMP standard

OpenACC

Code example

JÜLICH FORSCHUNGSZENTRUM

```
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

JÜLICH FORSCHUNGSZENTRUM

```
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);
```

OpenACC

Code example

JÜLICH FORSCHUNGSZENTRUM

```
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);
```

GPU tutorial this afternoon!

JÜLICH
FORSCHUNGSZENTRUM

Programming GPUs

Languages

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

Programming GPU Directly

Finally…

JÜLICH FORSCHUNGSZENTRUM

- 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…

JÜLICH FORSCHUNGSZENTRUM

- 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…

JÜLICH FORSCHUNGSZENTRUM

- 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 Threading Model

Warp the kernel, it's a thread!

JÜLICH FORSCHUNGSZENTRUM

- Methods to exploit parallelism:

CUDA Threading Model

Warp the kernel, it's a thread!

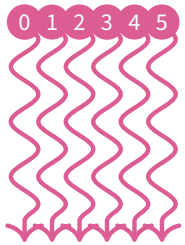
- Methods to exploit parallelism:
  - Thread

CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

  — Threads



CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

— Threads → Block

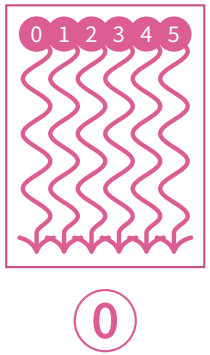
CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:

— Threads → Block

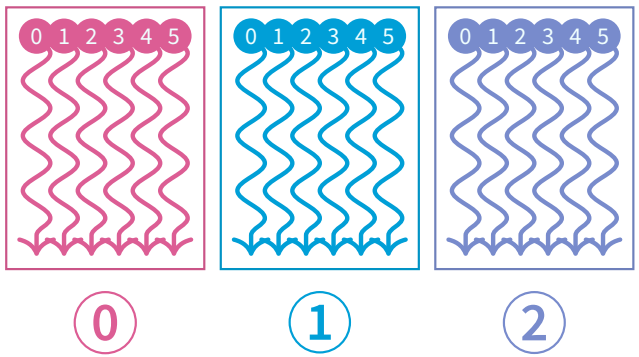
— Block



CUDA Threading Model

Warp the kernel, it's a thread!

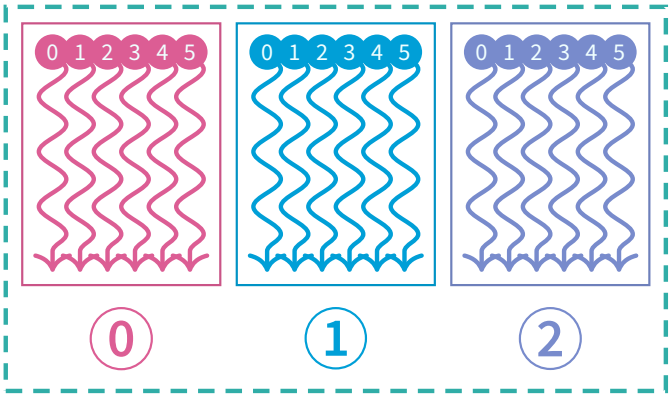
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks



CUDA Threading Model

Warp the kernel, it's a thread!

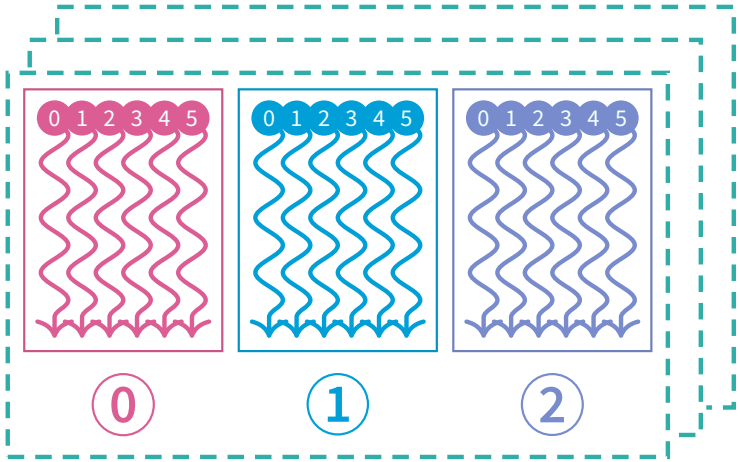
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid



CUDA Threading Model

Warp the kernel, it's a thread!

- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D

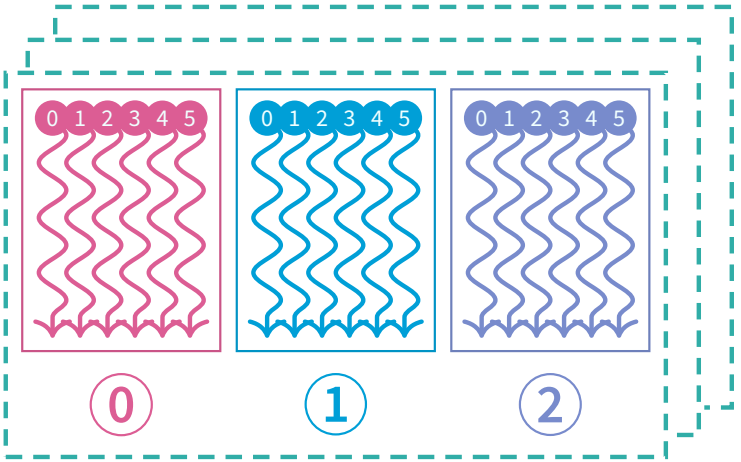


CUDA Threading Model

Warp the kernel, it's a thread!

JÜLICH FORSCHUNGSZENTRUM

- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



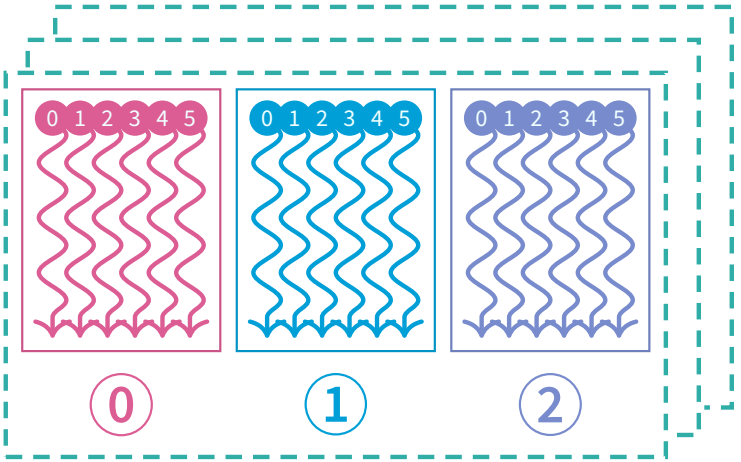
- Parallel function: **kernel**
  - `__global__ kernel(int a, float * b) { }`
  - Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...
- Execution entity: **threads**
  - Lightweight → fast switchting!
  - 1000s threads execute simultaneously → order non-deterministic!

CUDA Threading Model

Warp the kernel, it's a thread!

JÜLICH FORSCHUNGSZENTRUM

- 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 switchting!
  - 1000s threads execute simultaneously → order non-deterministic!

⇒ **SAXPY!**

CUDA SAXPY

With runtime-managed data transfers

JÜLICH FORSCHUNGSZENTRUM

```
__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

JÜLICH FORSCHUNGSZENTRUM

Specify kernel

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

With runtime-managed data transfers

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

With runtime-managed data transfers

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

With runtime-managed data transfers

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

CUDA SAXPY

With runtime-managed data transfers

JÜLICH FORSCHUNGSZENTRUM

```
__global__ void saxpy_cuda(int n, float a, float * x, fl
  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

JÜLICH
FORSCHUNGSZENTRUM

Programming GPUs

Abstraction Libraries/DSL

Abstraction Libraries & DSLs

JÜLICH FORSCHUNGSZENTRUM

- 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, …

Abstraction Libraries & DSLs

JÜLICH FORSCHUNGSZENTRUM

- 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

JÜLICH FORSCHUNGSZENTRUM

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

→ https://github.com/kokkos/kokkos/

```
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);
});
```

JÜLICH
FORSCHUNGSZENTRUM

Programming GPUs

Tools

GPU Tools

The helpful helpers helping helpless (and others)

JÜLICH FORSCHUNGSZENTRUM

- 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)

JÜLICH FORSCHUNGSZENTRUM

- 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

JÜLICH FORSCHUNGSZENTRUM

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*, i
  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
  0.11%  477.69us         1  477.69us  477.69us  477.69us  cudaGetDeviceProperties
  0.04%  179.27us         4  44.817us  40.744us  53.504us  cuDeviceGetName
  0.03%  136.20us       301     452ns     401ns  2.4000us  cudaConfigureCall
  0.00%  9.0850us         2  4.5420us  3.4760us  5.6090us  cudaEventRecord
  0.00%  8.7210us         1  8.7210us  8.7210us  8.7210us  cudaGetDevice
```

nvprof

Command that line

JÜLICH FORSCHUNGSZENTRUM

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

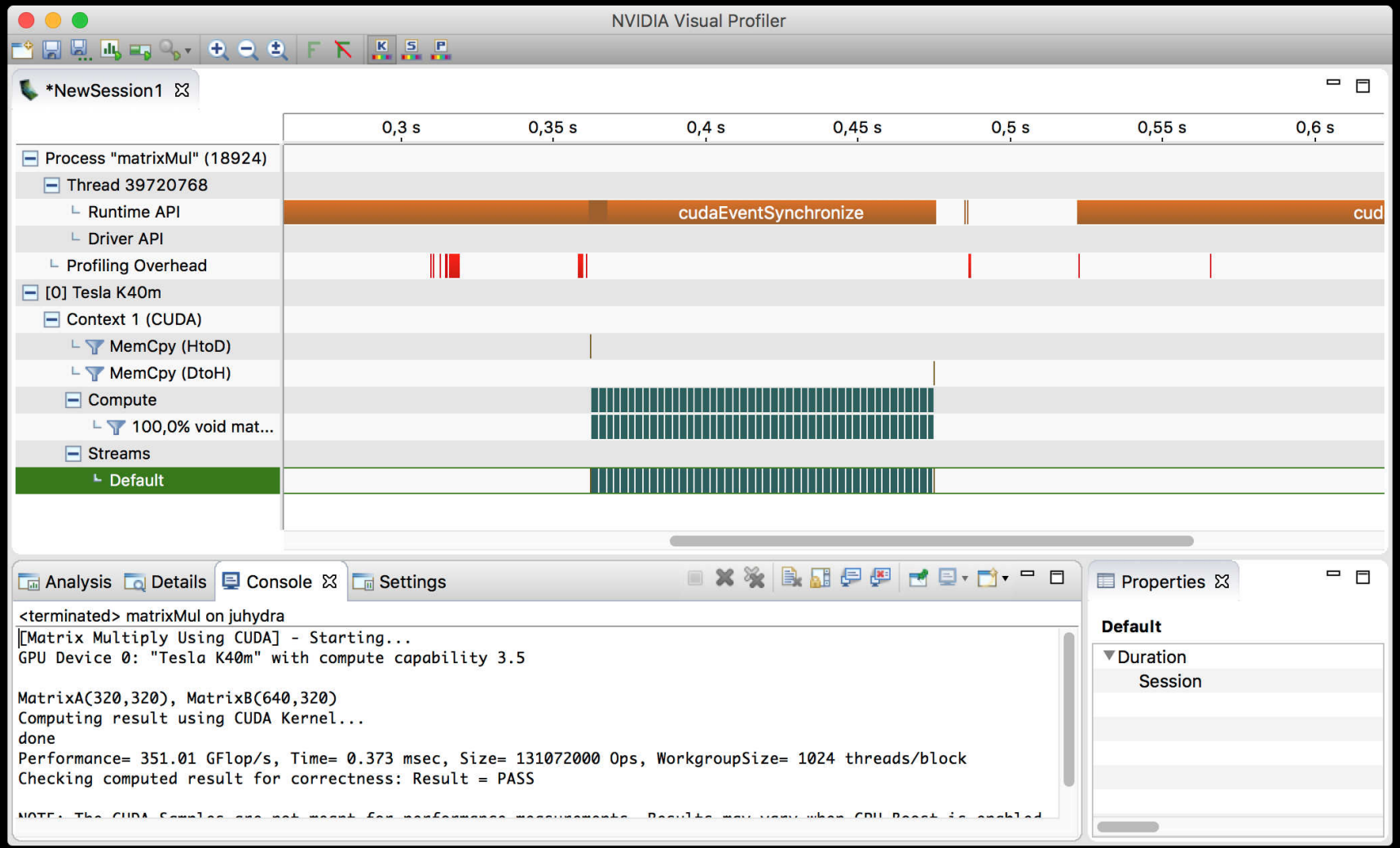
```
$ nvprof --metrics flop_sp_efficiency ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
[Matrix Multiply Using CUDA] - Starting...
==37122== NVPROF is profiling process 37122, command: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
GPU Device 0: "Tesla P100-SXM2-16GB" with compute capability 6.0

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)...
==37122== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
...
==37122== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Performance= 26.61 GFlop/s, Time= 80.697 msec, Size= 2147483648 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==37122== Profiling application: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
==37122== Profiling result:
==37122== Metric result:
Invocations                               Metric Name                        Metric Description         Min
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%
```

Visual Profiler

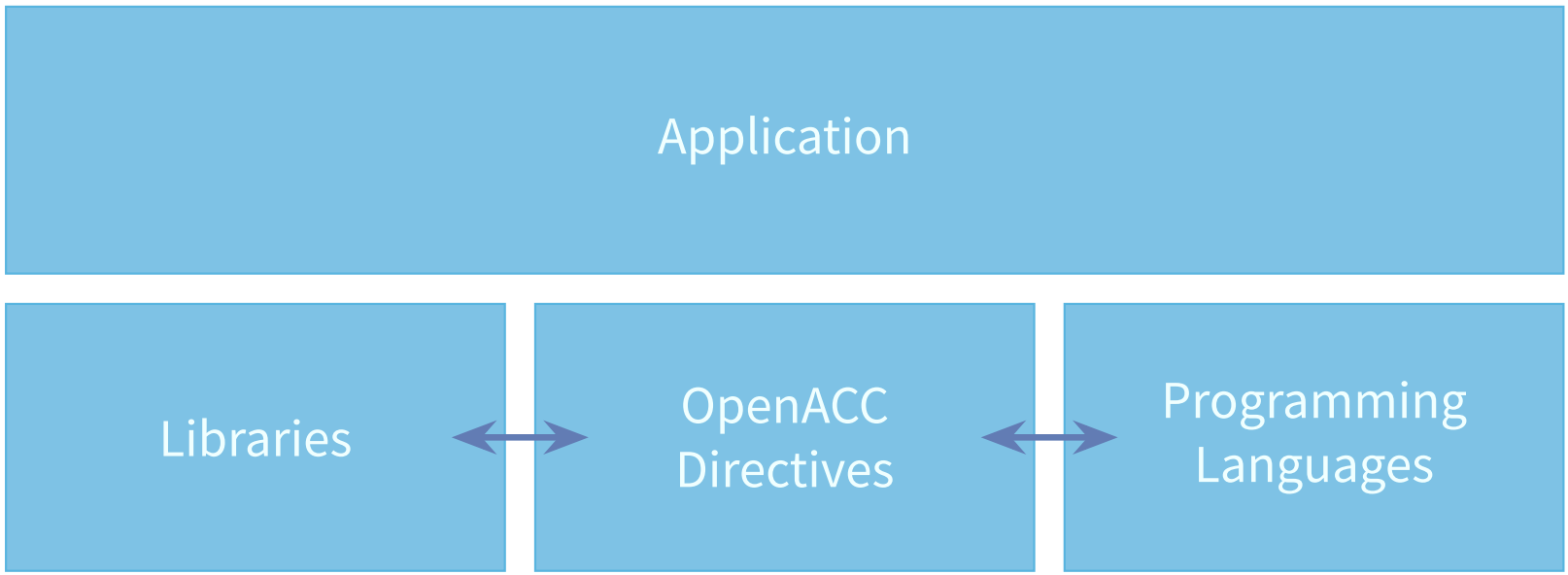
Your new favorite tool



JÜLICH
FORSCHUNGSZENTRUM

Conclusions

Summary of Acceleration Possibilities

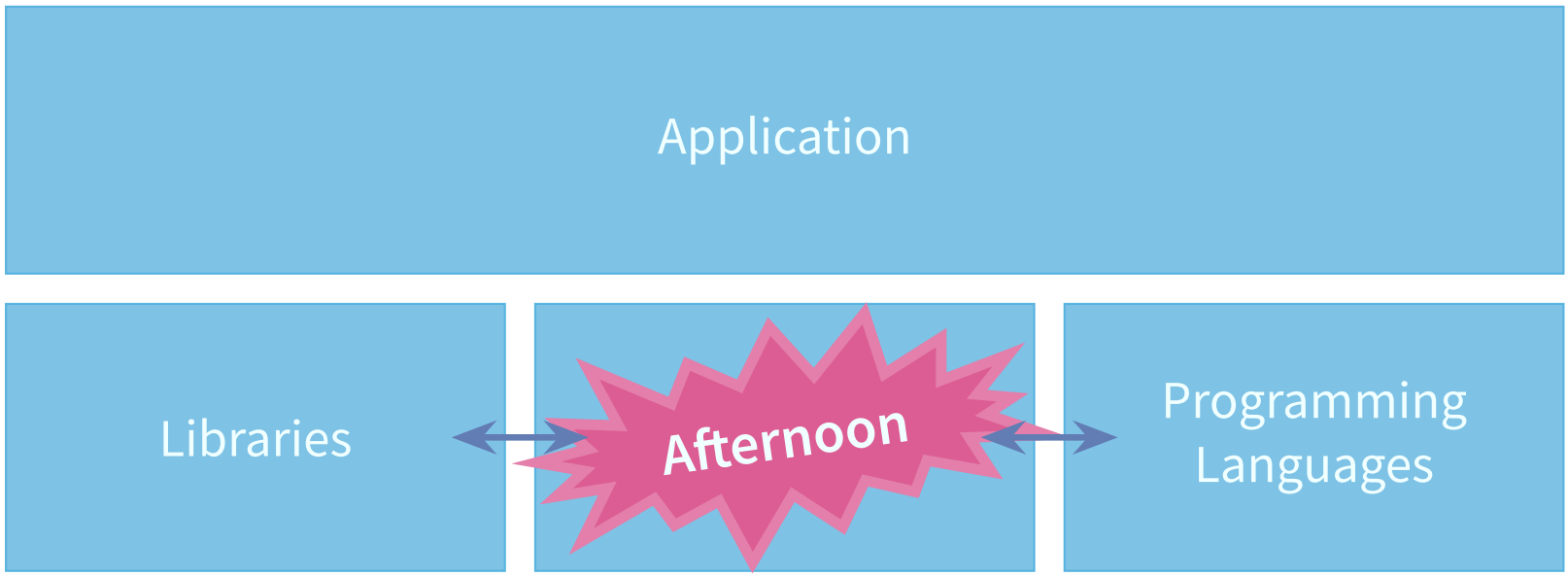


Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Summary of Acceleration Possibilities



Drop-in Acceleration

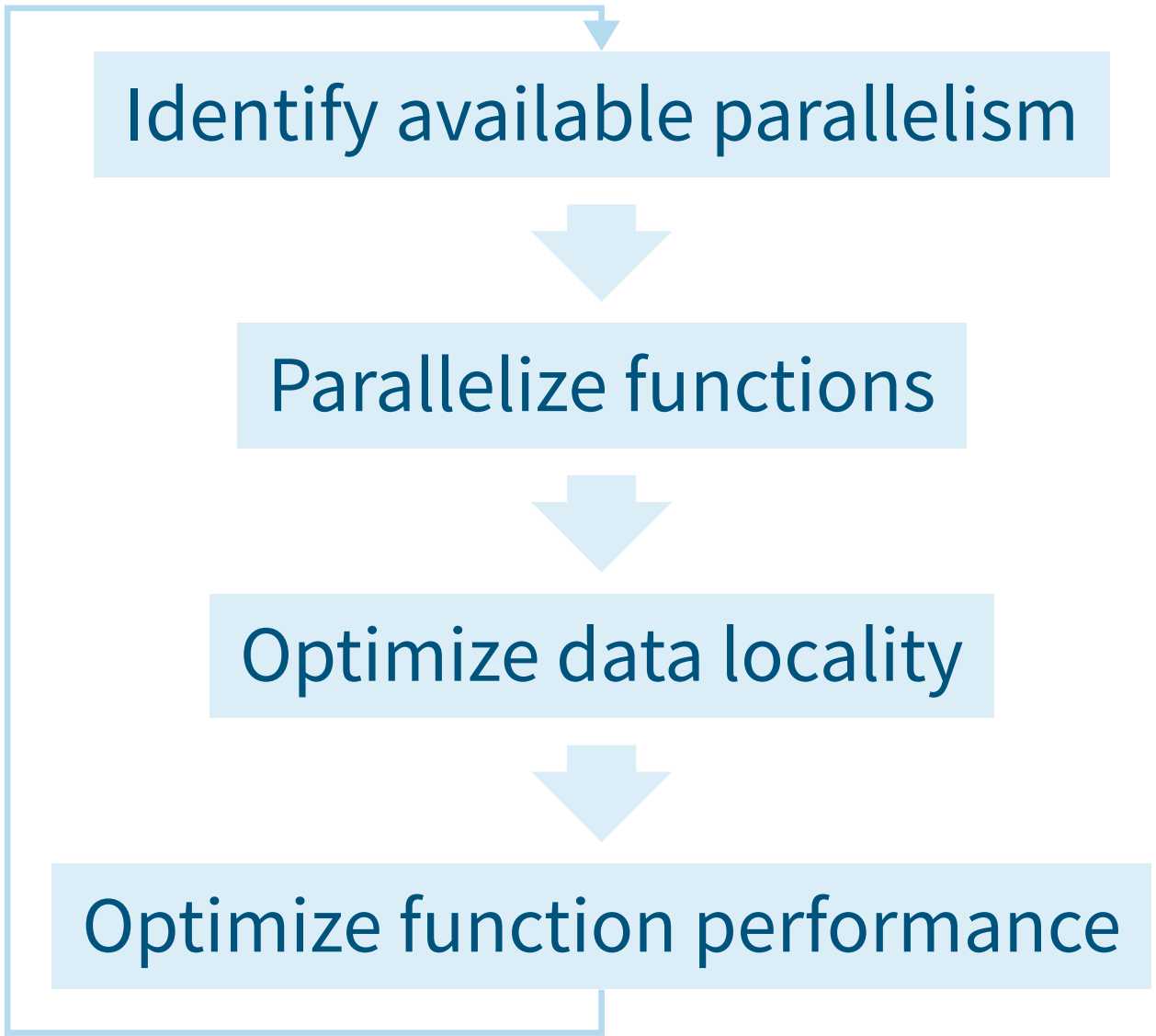
Easy Acceleration

Flexible Acceleration

The Performance Cookbook

For fashionable modern programmers

JÜLICH FORSCHUNGSZENTRUM



Omitted

There's so much more!

JÜLICH FORSCHUNGSZENTRUM

What I did not talk about

- Atomic  operations
- Shared memory
- Pinned memory
- Managed memory
- Debugging
- Overlapping streams
- Multi-GPU programming (intra-node; MPI)
- Cooperative threads
- Half precision FP16
- …

Summary & Conclusion

JÜLICH FORSCHUNGSZENTRUM

- GPUs can improve your performance many-fold
- For a fitting, parallelizable application
- Libraries are easiest
- Direct programming (plain CUDA) is most powerful
- OpenACC is somewhere in between (and portable)
- There are many tools helping the programmer

→ See it in action this afternoon at **OpenACC tutorial**

Summary & Conclusion

JÜLICH FORSCHUNGSZENTRUM

- GPUs can improve your performance many-fold
- For a fitting, parallelizable application
- Libraries are easiest
- Direct programming (plain CUDA) is most powerful
- OpenACC is somewhere in between (and portable)
- There are many tools helping the programmer

→ See it in action this afternoon at **OpenACC tutorial**

Thank you for your attention!

a.herten@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Appendix

Further Reading & Links

More!

JÜLICH FORSCHUNGSZENTRUM

- A discussion of SIMD, SIMT, SMT by Y. Kreinin.
- NVIDIA's documentation: `docs.nvidia.com`
- NVIDIA's Parallel For All blog

Volta Performance

9-@ &*>+3/2	9-@ CDD0	9-@ E D0	9-@ &100	9-@ V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	1B2	128	64	64
FP32 Cores / GPU	2880	30C2	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	B60	B6	1CB2	2560
TeDsor Cores / SM	EF	EF	EF	8
TeDsor Cores / GPU	EF	EF	EF	640
GPU GoostClocH	810/8C5 MI J	1114 MI J	1480 MI J	1462 MI J
PeaHFP32 TFKL PS[1]	5	6M	10M	15
PeaHFP64 TFKL PS[1]	1MC	M1	5M	CM
PeaHTeDsor TFKL PS[1]	EF	EF	EF	120
TextNre UDDs	240	1B2	224	320
MeP orQRDterSace	384TUDGVVW	384TUDGVVW	40B6TUDI GM2	40B6TUDI GM2
MeP orQSOe	Up to 12 GG	Up to 24 GG	16 GG	16 GG
K2 CacXe SOe	1536 KG	30C2 KG	40B6 KG	6144 KG
SXareY MeP orQSOe / SM	16 KG/32 KG/48 KG	B6 KG	64 KG	CoDS2NraUe Np to B6 KG
WeZGter Fle SOe / SM	256 KG	256 KG	256 KG	256KG
WeZGter Fle SOe / GPU	3840 KG	6144 KG	14336 KG	20480 KG
TVP	235 [atts	250 [atts	300 [atts	300 [atts
TraDsGtors	CM UDGD	8 UDGD	15M UDGD	21M UDGD
GPU VGe SOe	551 PP\	601 PP\	610 PP\	815 PP\
MaDNSactNrDZ Process	28 DP	28 DP	16 DP FDF] T^	12 DP FFE

[1] PeaHTFKL PS rates are UaseY oDGPU GoostClocH

Figure: Tesla V100 performance characteristics in comparison [5]

JÜLICH
FORSCHUNGSZENTRUM

Appendix

Glossary I

JÜLICH FORSCHUNGSZENTRUM

API A programmatic interface to software by well-defined functions. Short for application programming interface. 67, 68, 69, 75, 76, 77, 78, 79, 118

ATI Canada-based GPUs manufacturing company; bought by AMD in 2006. 4, 118

CPI Cycles per Instructions; a metric to determine efficiency of an architecture or program. 118

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 4, 63, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 98, 99, 112, 113, 118

Glossary II

JÜLICH FORSCHUNGSZENTRUM

DSL A Domain-Specific Language is a specialization of a more general language to a specific domain. 2, 3, 97, 98, 99, 118

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. 118

IPC Instructions per Cycle; a metric to determine efficiency of an architecture or program. 118

LLVM An open Source compiler infrastructure, providing, among others, Clang for C. 118

MPI The Message Passing Interface, a API definition for multi-node computing. 111, 118

Glossary III

JÜLICH FORSCHUNGSZENTRUM

NVIDIA US technology company creating GPUs. 2, 3, 4, 75, 76, 77, 78, 79, 102, 103, 115, 118

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with 80 GB/s. PCI-Express: 16 GB/s. 118

OpenACC Directive-based programming, primarily for many-core machines. 70, 71, 72, 73, 98, 99, 112, 113, 118

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 75, 76, 77, 78, 79, 102, 103, 118

OpenGL The *Open Graphics Library*, an API for rendering graphics across different hardware architectures. 118

Glossary IV

JÜLICH FORSCHUNGSZENTRUM

OpenMP Directive-based programming, primarily for multi-threaded machines. 70, 118

P100 A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast *HBM2* memory. 118

PAPI The Performance API, a C/C++ API for querying performance counters. 118

Pascal GPU architecture from NVIDIA (announced 2016). 118

perf Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. 118

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 118

Glossary V

POWER8 CPU architecture from IBM, available also under the OpenPOWER Foundation. 118

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 47, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 118

Score-P Collection of tools for instrumenting and subsequently scoring applications to gain insight into the program's performance. 118

Tesla The GPU product line for general purpose computing computing of NVIDIA. 118

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 63, 118

References I

[1] Chris McClanahan. “History and evolution of gpu architecture”. In: *A Survey Paper* (2010). URL: http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf (page 4).

[2] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (pages 5–7).

[3] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (pages 12, 13).

[4] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (pages 12, 13).

References II

[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 38–41, 116).

[6] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 48–52, 61, 62).