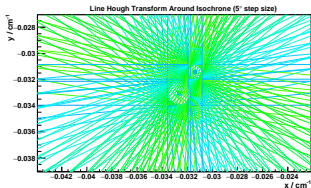


GPUs: Platform, Programming, Pitfalls

GridKa School 2016: Data Science on Modern Architectures

Andreas Herten

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

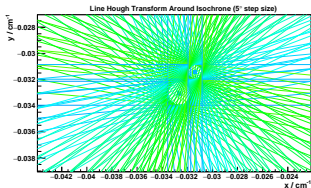


- Since then: NVIDIA Application Lab
Optimizing scientific applications for/on
GPUs

JÜLICH
APPLICATION LAB
NVIDIA

Andreas Herten

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



- Since then: NVIDIA Application Lab
Optimizing scientific applications for/on GPUs



Motivation

Platform

Hardware

Features

Programming

Libraries

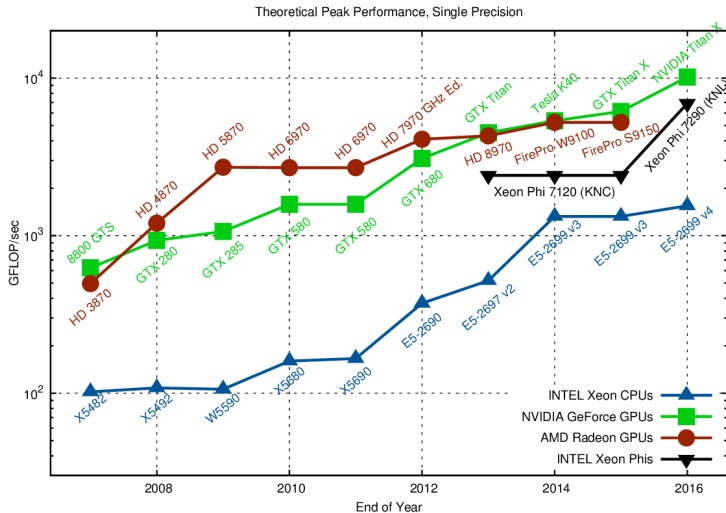
Directives

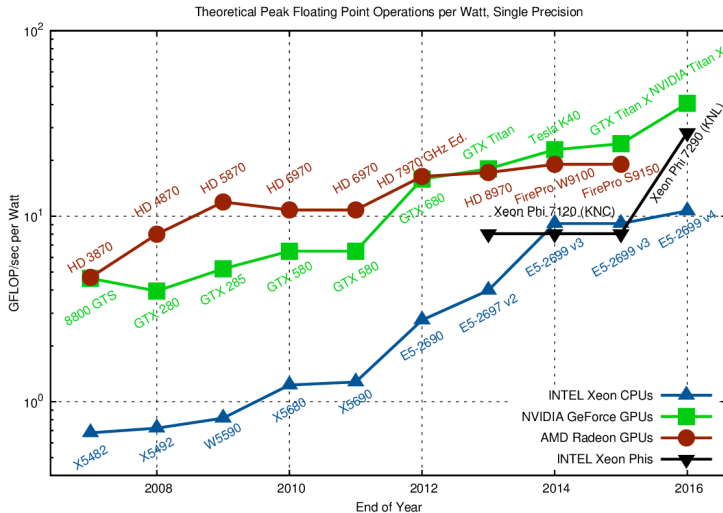
Languages

Tools

Pitfalls

- 1999: General computations with shaders of *graphics hardware*
- 2001: NVIDIA GeForce 3 with programmable shaders [1]; 2003: DirectX 9 at ATI
- 2016: Top 500: $\frac{1}{10}$ with GPUs, Green 500: 70 % of top 50 with GPUs









But why?!



But why?!

Let's find out!

Platform

CPU vs. GPU

A matter of specialties



CPU vs. GPU

A matter of specialties



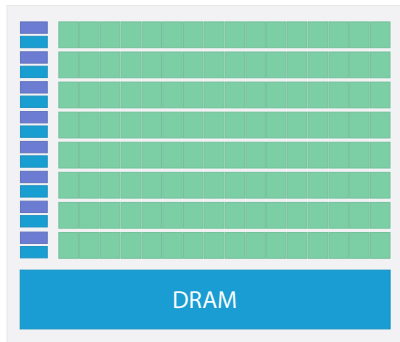
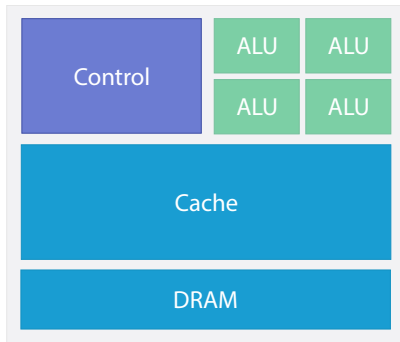
Transporting one



Transporting many

CPU vs. GPU

Chip



Aim: Hide Latency
Everything else follows

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency
Everything else follows

SIMT

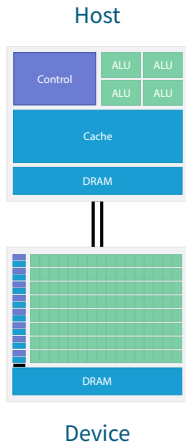
Asynchronicity

Memory

Memory

GPU memory ain't no CPU memory

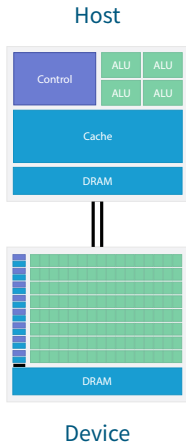
- GPU: accelerator / extension card
- Separate device from CPU



Memory

GPU memory ain't no CPU memory

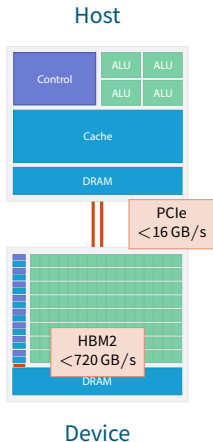
- GPU: accelerator / extension card
- Separate device from CPU
Separate memory, but UVA



Memory

GPU memory ain't no CPU memory

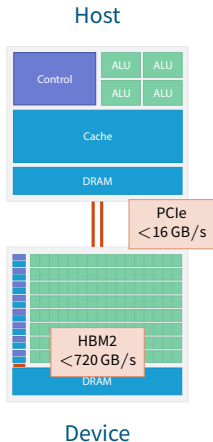
- GPU: accelerator / extension card
- Separate device from CPU
Separate memory, but UVA



Memory

GPU memory ain't no CPU memory

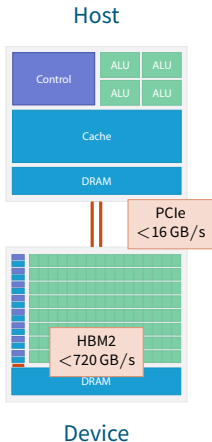
- GPU: accelerator / extension card
- Separate device from CPU
Separate memory, but UVA
- Memory transfers need special consideration!
Do as little as possible!



Memory

GPU memory ain't no CPU memory

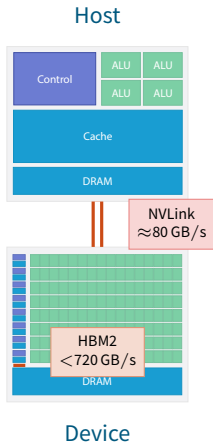
- GPU: accelerator / extension card
- Separate device from CPU
- Separate memory, but UVA and UM**
- Memory transfers need special consideration!
Do as little as possible!
 - Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance...?)



Memory

GPU memory ain't no CPU memory

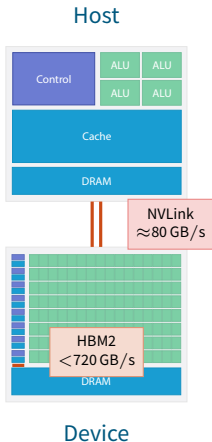
- GPU: accelerator / extension card
- Separate device from CPU
- **Separate memory, but UVA and UM**
- Memory transfers need special consideration!
Do as little as possible!
- Formerly: Explicitly copy data to/from GPU
Now: Done automatically (performance...?)



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...?)
 - Values for P100: 16 GB RAM, 720 GB/s



Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

- 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

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

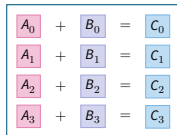
Memory

SIMT

Of threads and warps

Scalar

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

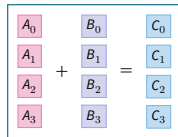


SIMT

Of threads and warps

Vector

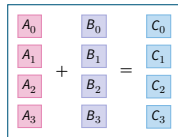
- CPU:
 - Single Instruction, Multiple Data (SIMD)



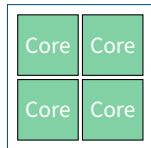
SIMT

Of threads and warps

Vector



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

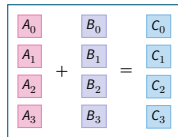


SIMT

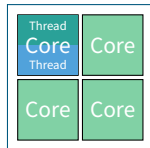
Of threads and warps

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

Vector



SMT

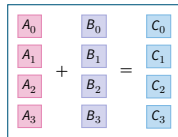


SIMT

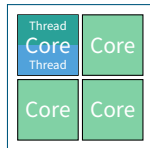
Of threads and warps

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

Vector



SMT

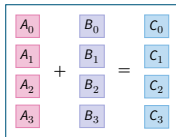


SIMT

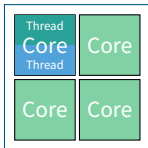
Of threads and warps

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

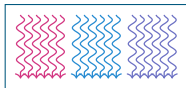
Vector



SMT




SIMT

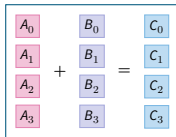


SIMT

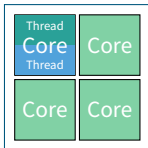
Of threads and warps

- 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 

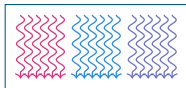
Vector



SMT



SIMT



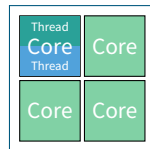
SIMT

Of threads and warps

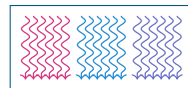
Vector

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

SMT



SIMT



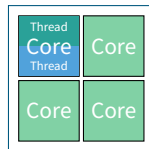
SIMT

Of threads and warps

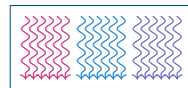
Vector

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

SMT



SIMT



SIMT

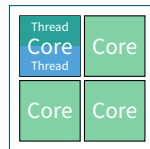
Of threads and warps



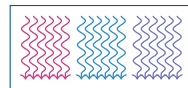
Vector

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

SMT



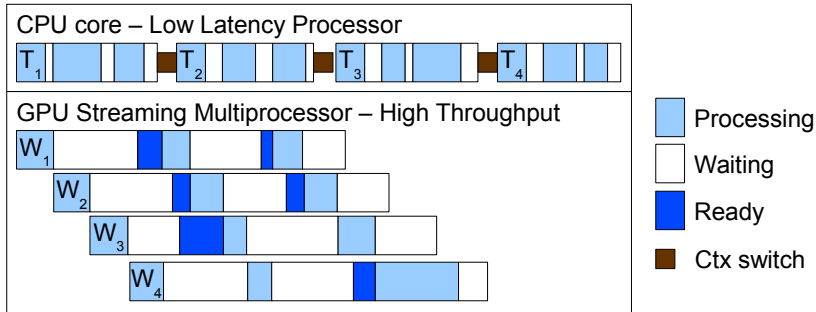
SIMT



Latency Hiding

GPU's ultimate feature

- CPU minimizes latency within each thread
- GPU hides latency with computations from other thread groups



CPU vs. GPU

Low latency vs. high throughput



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

Preface: CPU

A simple CPU program!

SAXPY: $\vec{y} = a\vec{x} + \vec{y}$, with single precision

Part of LAPACK BLAS Level 1

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

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

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

Programming GPUs is easy: Just don't!

Programming GPUs is easy: Just don't!

Use applications & libraries!

Programming GPUs is easy: Just don't!

Use applications & libraries!



Libraries

The truth is out there!

Programming GPUs is easy: Just don't!

Use applications & libraries!



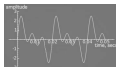
cuBLAS



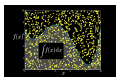
cuSPARSE



cuDNN



cuFFT



cuRAND



CUDA Math



{} ARRAYFIRE

Numba

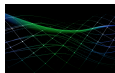
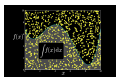
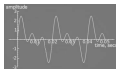
theano

Libraries

The truth is out there!

Programming GPUs is easy: Just don't!

Use applications & libraries!



Numba

theano

- GPU-parallel BLAS (all 152 routines)
- Single, double, complex data types
- Constant competition with Intel's MKL
- Multi-GPU support

→ <https://developer.nvidia.com/cublas>
<http://docs.nvidia.com/cuda/cublas>

cuBLAS

Code example

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

cublasInit();

float * d_x, * d_y;
cudaMalloc((void **)&d_x, n * sizeof(x[0]));
cudaMalloc((void **)&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);

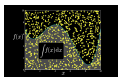
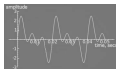
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

Libraries

The truth is out there!

Programming GPUs is easy: Just don't!

Use applications & libraries!



Numba

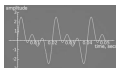
theano

Libraries

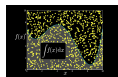
The truth is out there!

Programming GPUs is easy: Just don't!

Use applications & libraries!



cuFFT



cuRAND



CUDA Math



Numba

theano

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- Based on iterators
- Data parallel primitives (`scan()`, `sort()`, `reduce()`, ...)
- Fully compatible with plain CUDA C (comes with **CUDA** Toolkit)

→ <http://thrust.github.io/>
<http://docs.nvidia.com/cuda/thrust/>

Thrust

Code example

```
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;
```

Programming

Directives

GPU Programming with Directives

Keepin' you portable

- Annotate usual source code by directives

```
#pragma acc loop
```

```
for (int i = 0; i < 1; i++) {};
```

GPU Programming with Directives

Keepin' you portable

- Annotate usual source code by directives

```
#pragma acc loop
```

```
for (int i = 0; i < 1; i++) {};
```

- Also: Generalized functions

```
acc_copy();
```

- Compiler interprets directives, creates according instructions

- Annotate usual source code by directives

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

- Also: Generalized 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

- Only few compilers
- Not all the raw power
available
- Harder to debug
- Easy to program wrong

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

```
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);
```

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

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

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

Programming 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...

- 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`
GCC/LLVM solutions on way (slowly)
- 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`
GCC/LLVM solutions on way (slowly)
- 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`
GCC/LLVM solutions on way (slowly)
- Also: CUDA Fortran

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

- Methods to exploit parallelism:

- Methods to exploit parallelism:

- Thread



- Methods to exploit parallelism:

- Threads



- Methods to exploit parallelism:

— Threads → Block



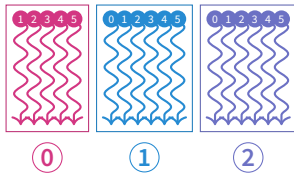
- Methods to exploit parallelism:

- Threads \rightarrow Block
- Block



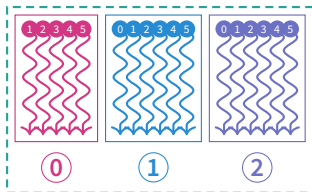
- Methods to exploit parallelism:

- Threads \rightarrow Block
- Blocks



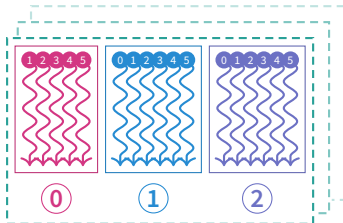
- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid



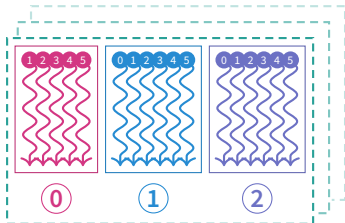
- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- All in 3D



- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- All in 3D



- Execution unit: **kernel**

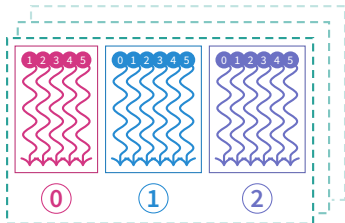
- Function executing in parallel on device

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

- Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...
- Execution order non-deterministic!
- Only threads in one warp (32 threads of block) can communicate reliably/quickly

- Methods to exploit parallelism:

- Threads → Block
- Blocks → Grid
- All in 3D



- Execution unit: **kernel**

- Function executing in parallel on device

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

- Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...
- Execution order non-deterministic!
- Only threads in one warp (32 threads of block) can communicate reliably/quickly

⇒ **SAXPY!**

```
__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();
```

Programming

Tools

- NVIDIA

- `cuda-gdb` GDB-like command line utility for debugging
 - `cuda-memcheck` Like Valgrind's memcheck, for checking errors in memory accesses
 - `Nsight` IDE for GPU developing, based on Eclipse (Linux, OS X) or Visual Studio (Windows)
 - `nvprof` Command line profiler, including detailed performance counters
 - `Visual Profiler` Timeline profiling and annotated performance experiments
- OpenCL: `CodeXL` (Open Source, GPUOpen/AMD) – debugging, profiling.

nvprof

Command that line

Usage: nvprof ./app

```

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==27580== Profiling application: ./matrixMul
==27580== Profiling result:
Time(%)    Time    Calls    Avg      Min      Max    Name
99.82%    111.33ms    301    369.85us    363.97us    375.62us    void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
0.11%     124.58us    2      62.289us    43.393us    81.185us    [CUDA memcpy HtoD]
0.07%     80.736us    1      80.736us    80.736us    80.736us    [CUDA memcpy DtoH]

==27580== API calls:
Time(%)    Time    Calls    Avg      Min      Max    Name
50.18%    348.27ms    3    116.09ms    241.59us    347.79ms    cudaMalloc
32.66%    226.68ms    1    226.68ms    226.68ms    226.68ms    cudaDeviceReset
15.47%    107.40ms    1    107.40ms    107.40ms    107.40ms    cudaEventSynchronize
0.52%     3.5853ms    301    11.911us    11.045us    34.486us    cudaLaunch
0.36%     2.4915ms    332    7.5040us    196ns     277.14us    cuDeviceGetAttribute
0.24%     1.6478ms    4    411.96us    294.19us    539.73us    cuDeviceTotalMem
0.19%     1.3333ms    3    444.43us    181.15us    813.00us    cudaMemcpy
0.12%     802.85us    3    267.62us    249.19us    299.41us    cudaFree
0.09%     604.10us    1    604.10us    604.10us    604.10us    cudaGetDeviceProperties
0.07%     451.30us    1505    299ns     266ns     6.0860us    cudaSetupArgument
0.05%     362.32us    1    362.32us    362.32us    362.32us    cudaDeviceSynchronize
0.03%     242.14us    4    60.534us    56.884us    69.764us    cuDeviceGetName
0.02%     127.99us    301    425ns     384ns     2.4580us    cudaConfigureCall
0.00%     10.920us    2    5.4600us    4.2100us    6.7100us    cudaEventRecord
0.00%     10.613us    1    10.613us    10.613us    10.613us    cudaGetDevice
0.00%     9.4980us    8    1.1870us    246ns     4.2760us    cuDeviceGet
0.00%     5.7490us    2    2.8740us    1.1700us    4.5790us    cudaEventCreate
0.00%     5.4630us    1    5.4630us    5.4630us    5.4630us    cudaEventElapsedTime
0.00%     3.2900us    2    1.6450us    1.2160us    2.0740us    cuDeviceGetCount
  
```


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

```

Slides — aherten@JUHYDRA: ~/cudaSamples/NVIDIA_CUDA-7.5_Samples/bin/x86_64/linux/release — ..linux/release — ssh juhydra
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==27425== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Performance= 3.26 GFlop/s, Time= 40.205 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

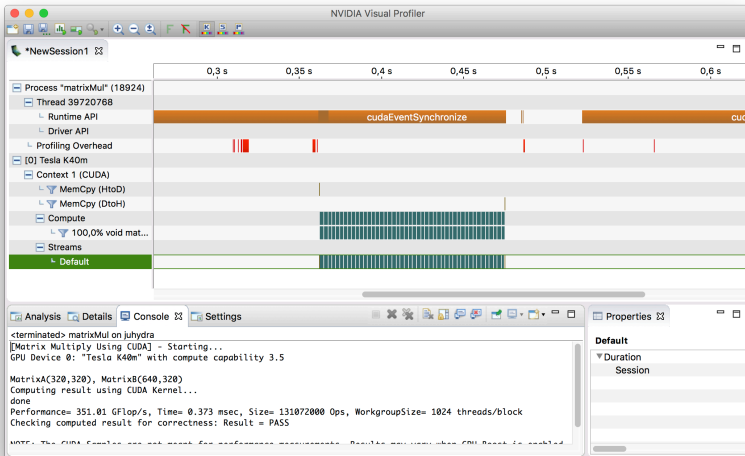
NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==27425== Profiling application: ./matrixMul
==27425== Profiling result:
==27425== Metric result:
Invocations      Metric Name      Metric Description      Min      Max      Avg
Device "Tesla K40m (0)"
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
301              flop_sp_efficiency      FLOP Efficiency(Peak Single)      7.88%      8.19%      8.07%

# aherten @ JUHYDRA in ~/cudaSamples/NVIDIA_CUDA-7.5_Samples/bin/x86_64/linux/release [20:58:41]
$ nvprof --metrics flop_sp_efficiency ./matrixMul

```

Visual Profiler

Your new favorite tool



Pitfalls

Pitfalls; Caveats; Tipps

There are mistakes to be made, opportunities to be missed

- Try to use a library if possible; let others do the hard work
- Profile! Don't trust your gut!
- Gradually improve and specialize when porting and optimizing
- Expose enough parallelism! The GPU wants to be fed
- Express data locality
- Study your data transfers, can you reduce it?
- Unified Memory is a good start, but explicit transfers might be fast
- Use specialized memory: constant memory, shared memory! Pinned host memory is sometimes a very easy performance booster
- Overlap computation and transfer
- Does your code really need double precision? Is single precision sufficient? Or, maybe, even half precision?
- The number of threads and blocks is a tunable parameter; 128 is a good start

What I did on talk about

- Atomic operations
- Shared memory
- Pinned memory
- How debugging works
- Overlapping streams
- Cross-compilation for heterogeneous systems
- ...

- 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
- Felice will surely give you more details in today's GPU session!

- 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
- Felice will surely give you more details in today's GPU session!

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

Appendix

Further Reading & Links

Pascal Performances

Glossary

References

Further Reading & Links

More!

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

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs ¹	5040	6840	10600
Peak FP64 GFLOPs ¹	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²
Manufacturing Process	28-nm	28-nm	16-nm FinFET

Figure: Tesla P100 performance characteristics in comparison [5]

API A programmatic interface to software by well-defined functions. Short for application programming interface. 55–59, 82

ATI Canada-based GPUs manufacturing company; bought by AMD in 2006. 4, 82

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 49, 55–70, 77, 78, 82

GCC The GNU Compiler Collection, the collection of open source compilers, among other for C and Fortran. 55–59, 82

- LLVM** An open Source compiler infrastructure, providing, among others, Clang for C. 55–59, 82
- NVIDIA** US technology company creating GPUs. 2–4, 55–59, 71, 80, 82
- OpenACC** Directive-based programming, primarily for many-core machines. 52–54, 77, 78, 82
- OpenCL** The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 55–59, 71, 82
- OpenMP** Directive-based programming, primarily for multi-threaded machines. 52, 82

- SAXPY** Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 39, 60–70, 82
- CPU** Central Processing Unit. 11, 12, 15–17, 52, 55–59, 82
- GPU** Graphics Processing Unit. 2, 3, 11–19, 21, 22, 32, 33, 35, 36, 40–45, 47, 48, 51, 52, 55–59, 71, 75, 77, 78, 82
- SIMD** Single Instruction, Multiple Data. 23–31, 82
- SIMT** Single Instruction, Multiple Threads. 13, 14, 18, 19, 21–33, 35, 36, 82
- SM** Streaming Multiprocessor. 23–31, 82
- SMT** Simultaneous Multithreading. 23–31, 82

- [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>.
- [2] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>.
- [3] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/>.
- [4] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/>.

- [5] NVIDIA Corporation. *Pictures: Pascal Blockdiagram, Pascal Multiprocessor*. Pascal Architecture Whitepaper. URL: <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.
- [6] Wes Breazell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/>.