JÜLICH
FORSCHUNGSZENTRUM

GPUs: Platform, Programming, Pitfalls

GridKa School 2016: Data Science on Modern Architectures

Andreas Herten, Forschungszentrum Jülich, 1 September 2016

About, Outline

JÜLICH
FORSCHUNGSZENTRUM

Andreas Herten

- Physics in
  — Aachen (Dipl. at CMS)
  — Jülich/Bochum (Dr. at PANDA)
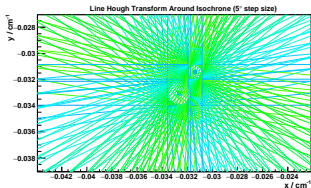
- Since then: NVIDIA Application Lab
  Optimizing scientific applications for/on
  GPUs

JÜLICH
APPLICATION LAB
NVIDIA

Member of the Helmholtz Association

Status Quo
GPU all around

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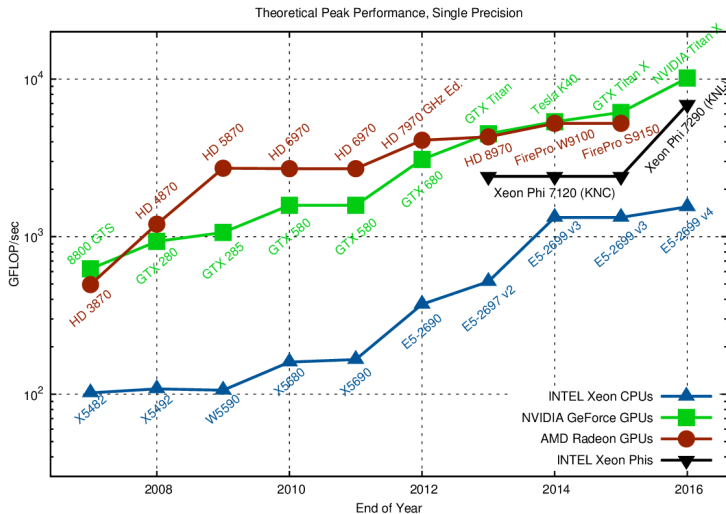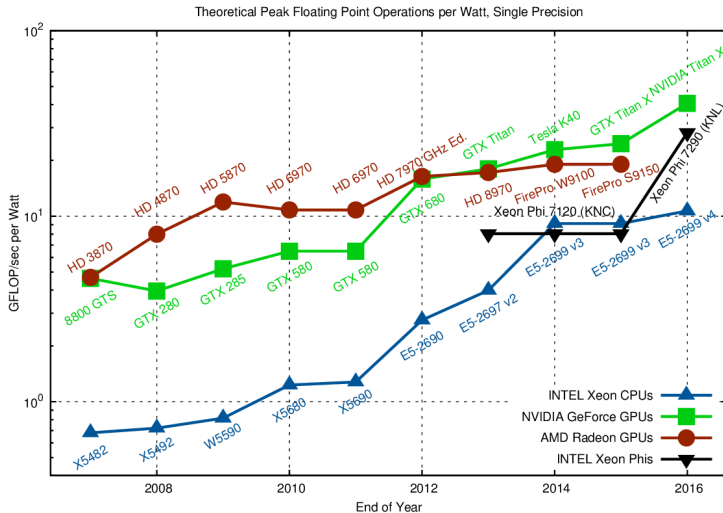
- 1999: General computations with shaders of *graphics hardware*
- 2001: NVIDIA GeForce 3 with programmable shaders [1]; 2003: DirectX 9 at ATI
- 2016: Top 500: $1/10$ with GPUs, Green 500: 70 $\%$ of top 50 with GPUs

Theoretical Peak Performance, Single Precision

Member of the Helmholtz Association

Theoretical Peak Floating Point Operations per Watt, Single Precision

Member of the Helmholtz Association

But why?!

Let's find out!

Platform

Member of the Helmholtz Association

CPU vs. GPU
A matter of specialties

Transporting one

Transporting many

CPU vs. GPU
Chip
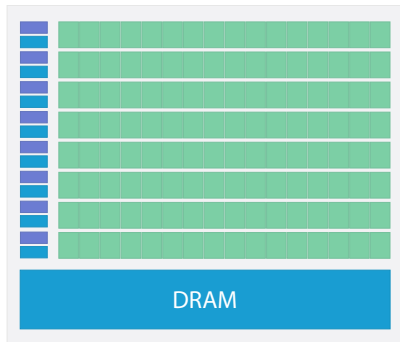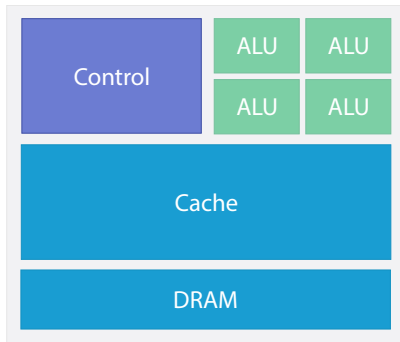
JÜLICH
FORSCHUNGSZENTRUM

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

JÜLICH
FORSCHUNGSZENTRUM

Aim: Hide Latency
Everything else follows

SIMT
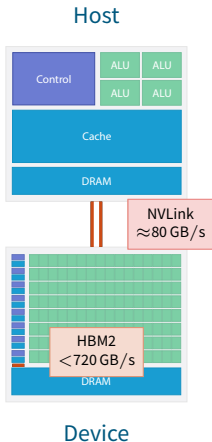
Asynchronicity

Memory

Member of the Helmholtz Association

Memory
GPU memory ain't no CPU memory

Host

- 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

Control | ALU ALU / ALU ALU

Cache

DRAM

NVLink ≈80 GB/s

HBM2 <720 GB/s

DRAM

Device

Aim: Hide Latency
Everything else follows

SIMT

Asynchronicity

Memory

Member of the Helmholtz Association

Async
Following different streams

- Problem: Memory transfer is comparably slow
  Solution: Do something else in meantime (**computation**)!
- $\rightarrow$ 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
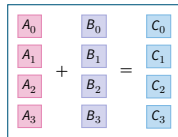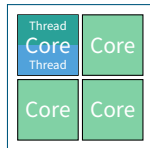*Everything else follows*

SIMT

Asynchronicity

Memory

Member of the Helmholtz Association

SIMT
Of threads and warps

Vector

- 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*)
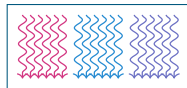 — Fast switching of threads (large register file)
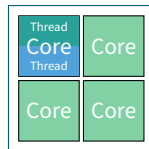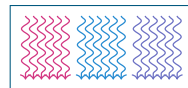 — Branching   if

SMT

SIMT

Member of the Helmholtz Association

SIMT
Of threads and warps

Pascal GP100

Vector

$$A_0 + B_0 = C_0$$
$$A_1 + B_1 = C_1$$
$$A_2 + B_2 = C_2$$
$$A_3 + B_3 = C_3$$

SMT

SIMT

SIMT
Of threads and warps

Multiprocessor

Vector

$$A_0 + B_0 = C_0$$
$$A_1 + B_1 = C_1$$
$$A_2 + B_2 = C_2$$
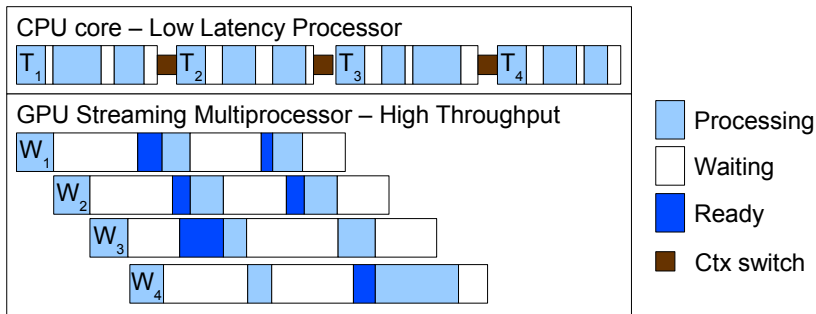$$A_3 + B_3 = C_3$$

SMT
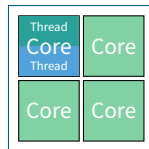
SIMT
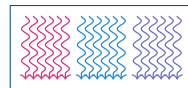
Latency Hiding
GPU's ultimate feature

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

CPU core – Low Latency Processor

T_1 T_2 T_3 T_4

GPU Streaming Multiprocessor – High Throughput

W_1

W_2

W_3

W_4

■ Processing
□ Waiting
■ Ready
■ Ctx switch

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

Member of the Helmholtz Association

Programming

Member of the Helmholtz Association

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];
}

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

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

Member of the Helmholtz Association

Programming GPUs is easy: Just don't!

Use applications & libraries!
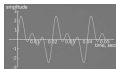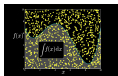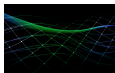
cuBLAS
Parallel algebra

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

$\rightarrow$ 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();
```
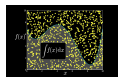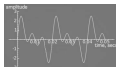
Programming GPUs is easy: Just don't!

Use applications & libraries!

cuSPARSE

cuBLAS

cuDNN

OpenCV

Thrust

{🔥} ARRAYFIRE

Numba

theano

cuFFT

cuRAND

CUDA Math

Member of the Helmholtz Association

Thrust
Iterators! Iterators everywhere!

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

$\rightarrow$ `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

JÜLICH
FORSCHUNGSZENTRUM

- 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

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

OpenACC
Code example

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

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

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

OpenACC
Code example

JÜLICH
FORSCHUNGSZENTRUM

```c
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…

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`**
    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

CUDA C/C++
Warp the kernel, it's a thread.

- Methods to exploit parallelism:

  — Threads $\rightarrow$ Block

  — Blocks $\rightarrow$ Grid

  — All in 3D
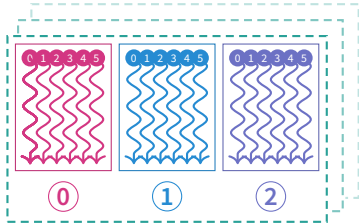
- Execution unit: **kernel**
  — Function executing in parallel on device

    `__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

  $\Rightarrow$ **SAXPY!**

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();
```

Programming

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.

nvprof
Command that line

Usage: `nvprof` `./app`

nvprof
Command that line

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

Visual Profiler
Your new favorite tool
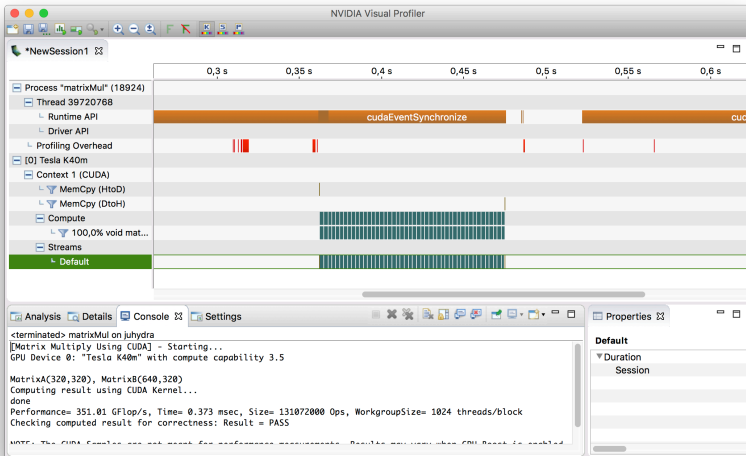
Pitfalls

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

Omitted
There's so much more!

What I did on talk about

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

Member of the Helmholtz Association

Summary & Conclusion

- 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

Member of the Helmholtz Association

Pascal Performance

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 mm²	601 mm²	610 mm²
Manufacturing Process	28-nm	28-nm	16-nm FinFET

Figure: Tesla P100 performance characteristics in comparison [5]

Glossary I

API A programmatic interface to software by well-defined functions. Short for application programming interface. 37, 52

ATI Canada-based GPUs manufacturing company; bought by AMD in 2006. 3, 52

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 29, 37–39, 48, 52, 53

GCC The GNU Compiler Collection, the collection of open source compilers, among other for C and Fortran. 37, 52

Member of the Helmholtz Association

JÜLICH
FORSCHUNGSZENTRUM

LLVM An open Source compiler infrastructure, providing, among others, Clang for C. 37, 52

NVIDIA US technology company creating GPUs. 2, 3, 37, 41, 50, 52

OpenACC Directive-based programming, primarily for many-core machines. 33–35, 48, 52

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 37, 41, 52

OpenMP Directive-based programming, primarily for multi-threaded machines. 33, 52

Glossary III

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 23, 38, 39, 52

CPU Central Processing Unit. 9, 10, 13, 33, 37, 52, 53

GPU Graphics Processing Unit. 2, 9–14, 16, 24–26, 28, 32, 33, 37, 41, 46, 48, 52, 53

SIMD Single Instruction, Multiple Data. 17–19, 52

SIMT Single Instruction, Multiple Threads. 11, 12, 14, 16–19, 52

SM Streaming Multiprocessor. 17–19, 52

SMT Simultaneous Multithreading. 17–19, 52

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.

[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/.

References II

[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/.

Member of the Helmholtz Association