GPU: PLATFORM AND PROGRAMMING

GEORGIAN-GERMAN SCIENCE BRIDGE QUALISTARTUP

12 September 2019 | Andreas Herten | Forschungszentrum Jülich ***Handout Version***

Member of the Helmholtz Association

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

About, Outline

Jülich Supercomputing Centre

- Operation of supercomputers
- Application support
- Research
- Me: *All things GPU*

Slides: `http://bit.ly/ggsb-gpu`

Topics

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Status Quo

A short but parallel story

1999 Graphics computation pipeline implemented in dedicated *graphics hardware*
Computations using OpenGL graphics library [2]
»GPU« coined by NVIDIA [3]

2001 NVIDIA GeForce 3 with *programmable* shaders (instead of fixed pipeline) and floating-point support; 2003: DirectX 9 at ATI

2007 CUDA

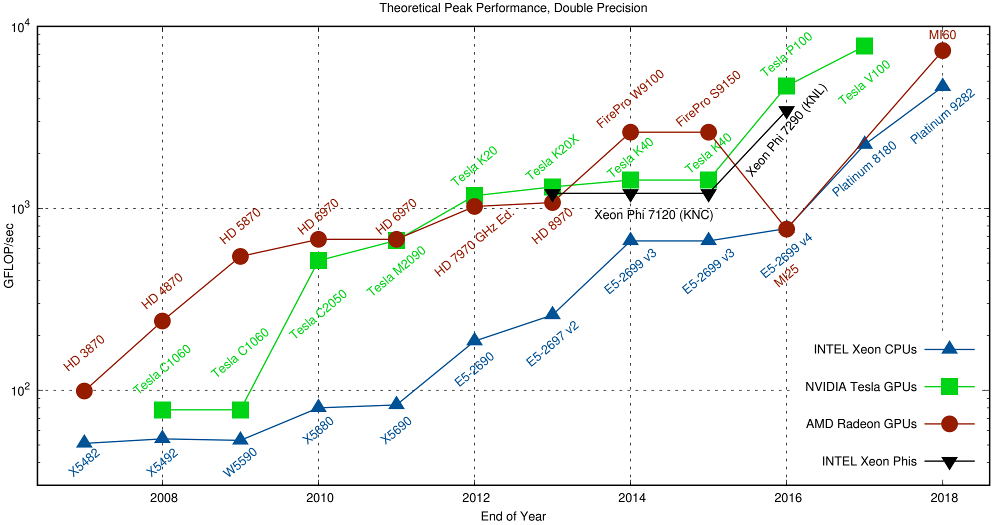
2009 OpenCL

2019 Top 500: 25 % with NVIDIA GPUs (#1, #2) [4], Green 500: 8 of top 10 with GPUs [5]

2021 Aurora: First (?) US exascale supercomputer based on Intel GPUs
Frontier: First (?) US *more-than*-exascale supercomputer based on AMD GPUs

Status Quo

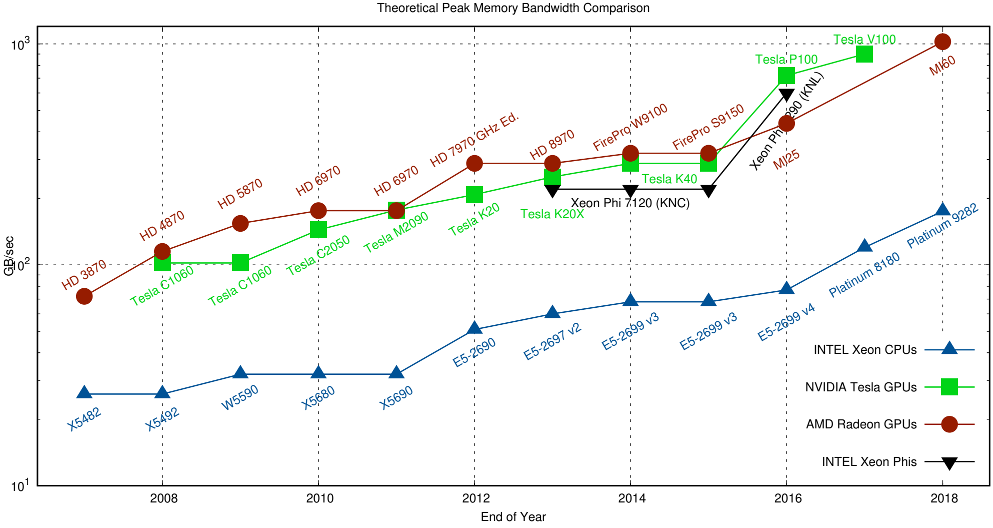
A short but parallel story



Graphic: Rupp [6]

Status Quo

Peak performance double precision



Graphic: Rupp [6]

Status Quo

Peak memory bandwidth

JURECA – Jülich's Multi-Purpose Supercomputer

- 1872 nodes with Intel Xeon E5 CPUs (2 $\times$ 12 cores)
- 75 nodes with 2 NVIDIA Tesla K80 cards (look like 4 GPUs)
- JURECA Booster: 1640 nodes with Intel Xeon Phi *Knights Landing*
- 1.8 (CPU) + 0.44 (GPU) + 5 (KNL) PFLOP/s peak performance
- Mellanox EDR InfiniBand

Next:
Booster!

JUWELS – Jülich’s New Scalable System

- 2500 nodes with Intel Xeon CPUs (2×24 cores)
- 48 nodes with 4 NVIDIA Tesla V100 cards
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance

Platform

CPU vs. GPU

A matter of specialties

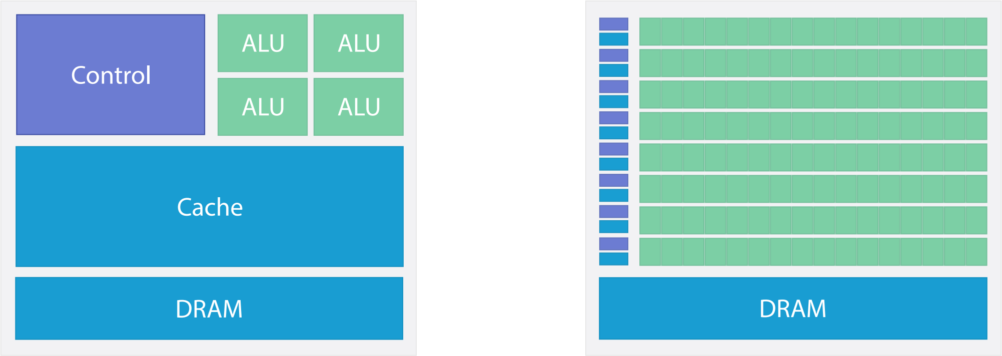
Transporting one

Transporting many

Graphics: Lee [7] and Shearings Holidays [8]

CPU vs. GPU

Chip



GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Memory

GPU memory ain't no CPU memory

Unified Virtual Addressing

- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA

- Memory transfers need special consideration!

Do as little as possible!

- Formerly: Explicitly copy data to/from GPU

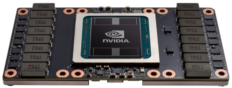
Now: Done automatically (performance…?)

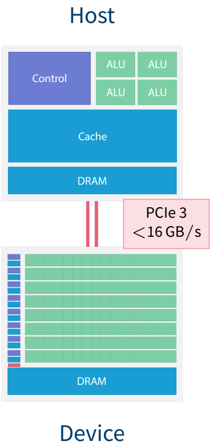
P100

16 GB RAM, 720 GB/s

V100

32 GB RAM, 900 GB/s





Memory

GPU memory ain't no CPU memory

Unified Memory

- GPU: accelerator / extension card

→ Separate device from CPU

Separate memory, but UVA and UM

- Memory transfers need special consideration!

  Do as little as possible!

- Formerly: Explicitly copy data to/from GPU

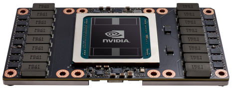
  Now: Done automatically (performance…?)

P100

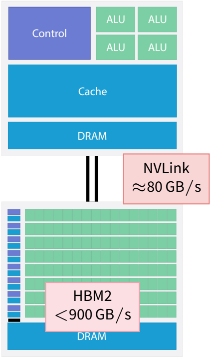
16 GB RAM, 720 GB/s

V100

32 GB RAM, 900 GB/s



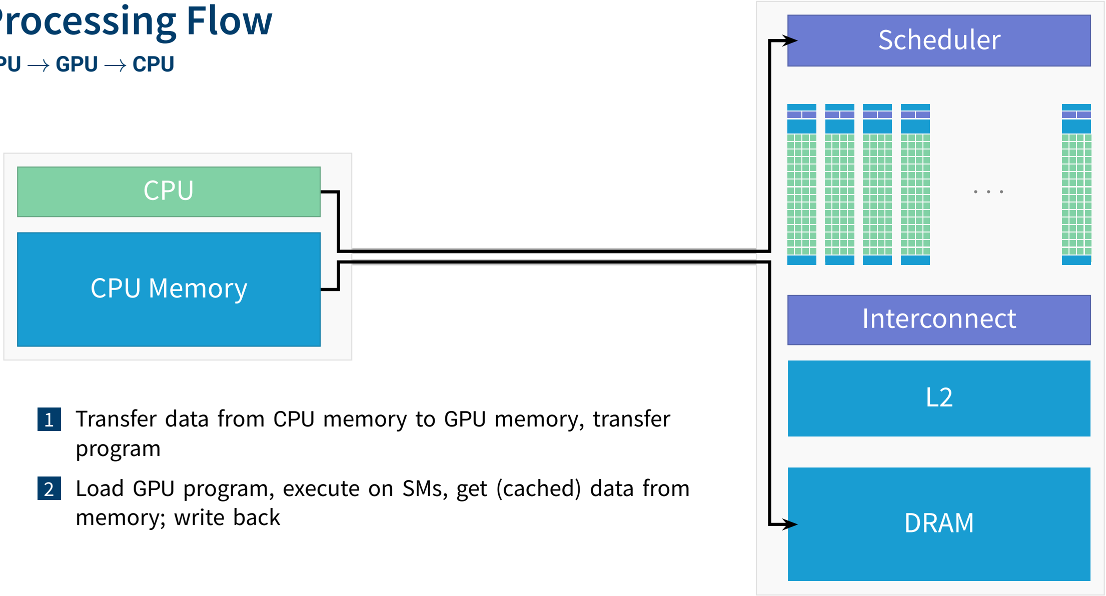
Host



Device

Processing Flow

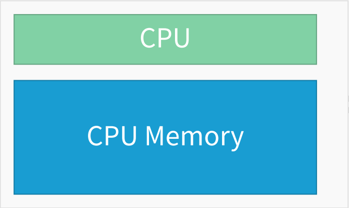
CPU → GPU → CPU

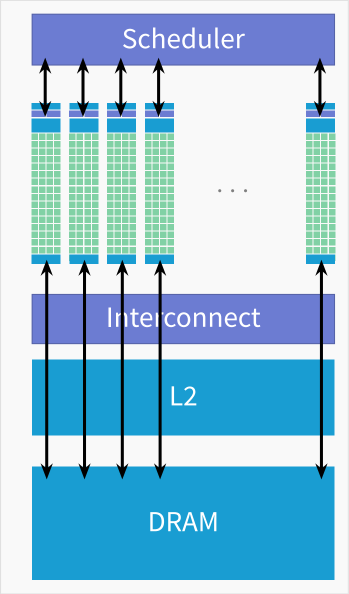


1. Transfer data from CPU memory to GPU memory, transfer program
2. Load GPU program, execute on SMs, get (cached) data from memory; write back

Processing Flow

CPU → GPU → CPU

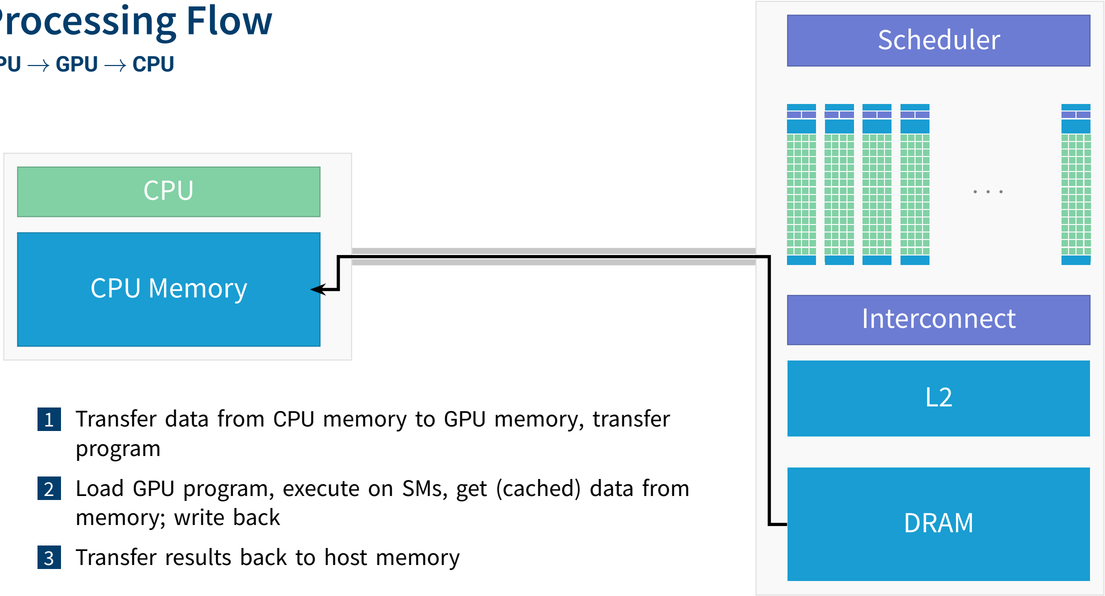




1. Transfer data from CPU memory to GPU memory, transfer program
2. Load GPU program, execute on SMs, get (cached) data from memory; write back

Processing Flow

CPU → GPU → CPU



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

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Async

Following different streams

- Problem: Memory transfer is comparably slow
  Solution: Do something else in meantime (**computation**)!

→ Overlap tasks

- Copy and compute engines run separately (*streams*)

- GPU needs to be fed: Schedule many computations
- CPU can do other work while GPU computes; synchronization
- Also: Fast switching of contexts to keep GPU busy (*KGB*)

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

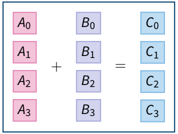
Memory

SIMT

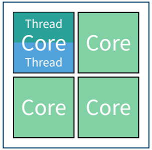
Of threads and warps

- CPU:
    - Single Instruction, Multiple Data (SIMD)
    - Simultaneous Multithreading (SMT)
- GPU: Single Instruction, Multiple Threads (SIMT)
    - CPU core $\approxeq$ GPU multiprocessor (SM)
    - Working unit: set of threads (32, a *warp*)
    - Fast switching of threads (large register file) $\rightarrow$ **hide latency**
    - Branching if

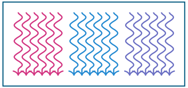
Vector



SMT

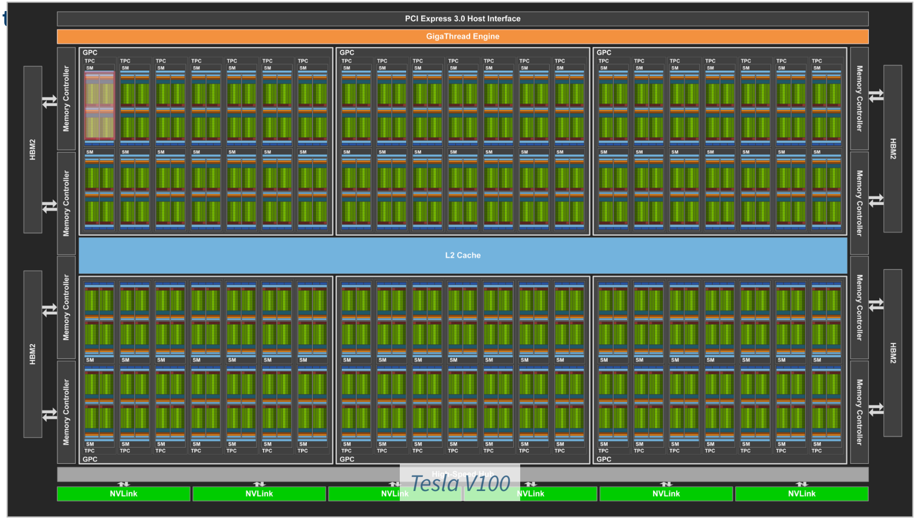


SIMT

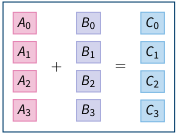


SIMT

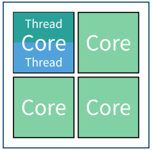
Of t



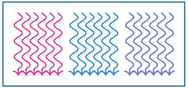
Vector



SMT



SIMT

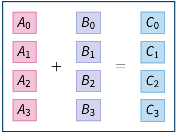


Graphics: Nvidia Corporation [9]

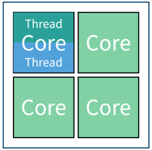
SIMT

Of t

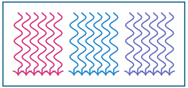
Vector



SMT



SIMT



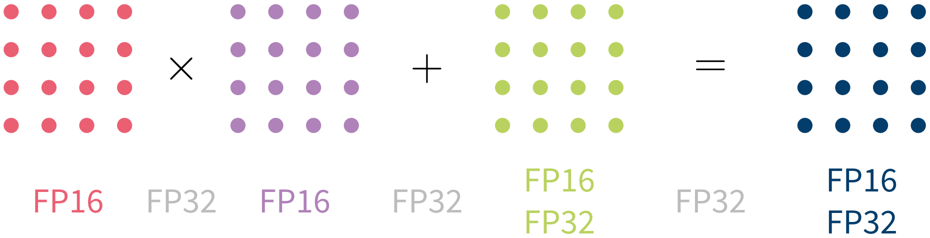
Graphics: Nvidia Corporation [9]

New: Tensor Cores

New in Volta

- 8 Tensor Cores per Streaming Multiprocessor (SM) (640 total for V100)
- Performance: 125 TFLOP/s (half precision)
- Calculate **A** $\times$ **B** $+$ **C** $=$ **D** (4×4 matrices; **A**, **B**: half precision)

$\rightarrow$ 64 floating-point FMA operations per clock (mixed precision)



GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

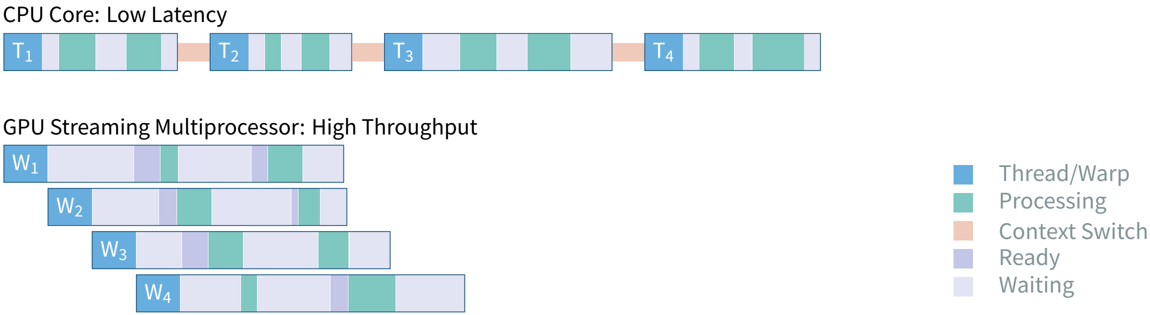
Memory

Low Latency vs. High Throughput

Maybe GPU's ultimate feature

CPU Minimizes latency within each thread

GPU Hides latency with computations from other thread warps

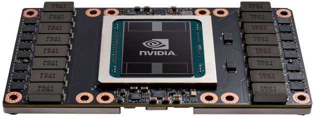


CPU vs. GPU

Let's summarize this!

Optimized for **low latency**

- + Large main memory
- + Fast clock rate
- + Large caches
- + Branch prediction
- + Powerful ALU
- – Relatively low memory bandwidth
- – Cache misses costly
- – Low performance per watt



Optimized for **high throughput**

- + High bandwidth main memory
- + Latency tolerant (parallelism)
- + More compute resources
- + High performance per watt
- – Limited memory capacity
- – Low per-thread performance
- – Extension card

GPU Device Comparison

Feature	NVIDIA	AMD
	HPC-grade	
Name	Tesla V100 (Volta)	Radeon Instinct MI60 (*Vega*)
Performance / TFLOP/s	$14.8_{\text{FP32}}, 7.5_{\text{FP64}}$	$14.7_{\text{FP32}}, 7.4_{\text{FP64}}$
Memory Capacity / GB	32	32
Memory Bandwidth / TB/s	0.9	1
	Workstation-grade	
Name	Quadro GV100 (Volta)	Radeon Pro Vega II (*Vega*)
Performance	$14.8_{\text{FP32}}, 7.4_{\text{FP64}}$	$14.2_{\text{FP32}}, 0.9_{\text{FP64}}$
Memory Capacity	32	32
Memory Bandwidth	0.9	1
	Consumer-grade	
Name	GeForce RTX 2080 Super (*Turing*)	Radeon RX 5700 XT (*Navi*)
Performance	$11_{\text{FP32}}, 0.3_{\text{FP64}}$	10_{FP32}, ?
Memory Capacity	8	8
Memory Bandwidth	0.5	0.4

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

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

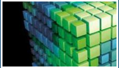
Use applications & libraries

Wizard: Breazell [10]

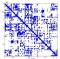
Libraries

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

Use applications & libraries



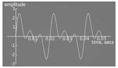
cuBLAS *rocBLAS*



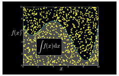
cuSPARSE
rocSPARSE

cuDNN

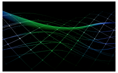
cuDNN *rocDNN*



cuFFT *rocFFT*



cuRAND *rocRAND*



CUDA Math

OpenCV

Thrust

ArrayFire

Numba

theano

Wizard: Breazell [10]

BLAS on GPU

Parallel algebra

cuBLAS

- GPU-parallel BLAS (all 152 routines) by NVIDIA
- 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`

rocBLAS

- AMD BLAS implementation

→ `https://github.com/ROCmSoftwarePlatform/rocBLAS`
`https://rocblas.readthedocs.io/en/latest/`

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

Allocate GPU memory

Copy data to GPU

Call BLAS routine

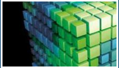
Copy result to host

Finalize

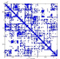
Libraries

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

Use applications & libraries



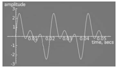
cuBLAS *rocBLAS*



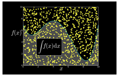
cuSPARSE
rocSPARSE

cuDNN

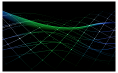
cuDNN *rocDNN*



cuFFT *rocFFT*



cuRAND *rocRAND*



CUDA Math

OpenCV

Thrust

ArrayFire

Numba

theano

Wizard: Breazell [10]

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)
- Great with `[](){}` lambdas!

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

- AMD backend available: `https://github.com/ROCmSoftwarePlatform/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;
```

Thrust

Code example with lambdas

```
#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

Programming GPUs

About GPU Programming

Parallelism

Libraries are not enough?

You think you want to write your own GPU code?

Primer on Parallel Scaling

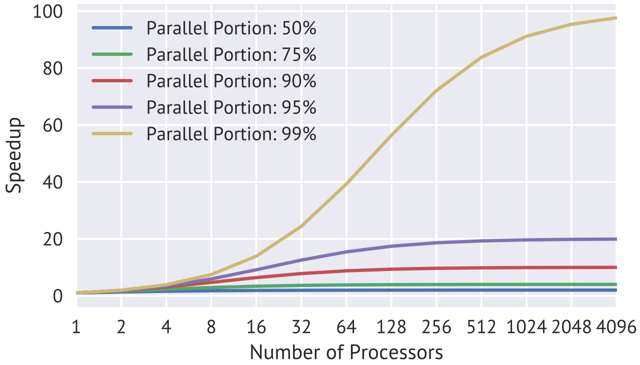
Amdahl's Law

Possible maximum speedup for N parallel processors

Total Time $t = t_{\mathbf{serial}} + t_{\mathbf{parallel}}$

N Processors $t(N) = t_s + t_p/N$

Speedup $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p/N}$



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Parallelism

Parallel programming is not easy!

Things to consider:

- Is my application **computationally intensive** *enough*?
- What are the levels of **parallelism**?
- How much **data** needs to be **transferred**?
- Is the **gain** worth the pain?

Possibilities

Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*…

- OpenACC, OpenMP
- Thrust, Kokkos, SYCL
- PyCUDA, Cupy, Numba

Other alternatives (for completeness)

- CUDA Fortran
- HIP
- OpenCL

Programming GPUs

Directives

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 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.

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

```c
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);
```

See JSC OpenACC course in October!

Programming GPUs

Languages, finally

Programming GPU Directly

Finally…

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

CUDA **NVIDIA's GPU platform** *2007*

- **Platform: Drivers, programming language (CUDA C/C++), API, compiler, tools, …**
- **Only NVIDIA GPUs**
- **Compilation with `nvcc` (free, but not open)**
  `clang` has CUDA support, but CUDA needed for last step
- **Also: CUDA Fortran**

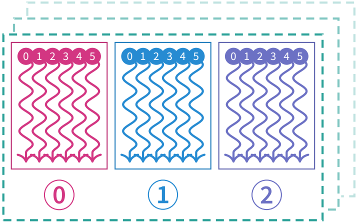
HIP AMD's new unified programming model for AMD (via ROCm) and NVIDIA GPUs *2016+*

- 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!

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



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

⇒ **SAXPY!**

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

CUDA SAXPY

With runtime-managed data transfers

```
__global__ void saxpy(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();
```

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

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device
- All threads execute same code; but can take different paths in program flow (some penalty)
- Info about thread: local, global IDs

```
int currentThreadId = threadIdx.x;
float x = input[currentThreadId];
output[currentThreadId] = x*x;
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```c
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
void scale(float scale, float * in, float * out, int N) {
    for (
        int i = 0;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove for | Add global

Replace i by threadIdx.x | … including block configuration

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
          ;
          i++
    )
        if (i < N)
            out[i] = scale * in[i];
}
```


JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace i by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = 0;



        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Summary

- C function with explicit loop

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

- CUDA kernel with implicit loop

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N)
        out[i] = scale * in[i];
}
```

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!
- Example:

```
int nThreads = 32;
scale<<<N/nThreads, nThreads>>>(23, in, out, N)
```

- Possibility for too many threads; include termination condition into kernel!
- Actual full kernel launch definition

```
kernel<<<dim3 gD, dim bD, size_t shared, cudaStream_t stream>>>(...)
```

Grid Sizes

- Block and grid sizes are hardware-dependent
- JUWELS: Tesla V100

Block
- $\vec{N}_{\text{Thread}} \leq (1024^{(x)}, 1024^{(y)}, 64^{(z)})$
- $\prod_{i=x,y,z} \vec{N}^{(i)}_{\text{Thread}} \leq 1024$

Grid
- $\vec{N}_{\text{Blocks}} \leq (2147483647^{(x)}, 65535^{(y)}, 65535^{(z)}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

- Find out yourself: `deviceQuery` example from CUDA Samples
- Workflow: Chose 128 or 256 as block dim; calculate grid dim from problem size

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16);
int gx = (Nx % blockDim.x == 0) ? Nx / blockDim.x : Nx / blockDim.x + 1;
int gy = (Ny % blockDim.y == 0) ? Ny / blockDim.y : Ny / blockDim.y + 1;
dim3 gridDim(gx, gy);
kernel<<<gridDim, blockDim>>>();
```

HIP SAXPY

From CUDA to HIP

```
#include <cuda_runtime.h>
__global__ void saxpy(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();
```

Works on AMD and NVIDIA GPUs!

HIP SAXPY

From CUDA to HIP

```
#include <hip/hip_runtime.h>
__global__ void saxpy(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;a
int n = 10;
float x[n], y[n];
// fill x, y
hipMallocManaged(&x, n * sizeof(float));
hipMallocManaged(&y, n * sizeof(float));

hipLaunchKernelGGL(saxpy, 2, 5, 0, 0, n, a, x, y);

hipDeviceSynchronize();
```

Works on AMD and NVIDIA GPUs!

Programming GPUs

Abstraction Libraries/DSL

Abstraction Libraries & DSLs

- Libraries with ready-programmed abstractions; partly compiler/transpiler necessary
- Have different backends to choose from for targeted accelerator
- Between Thrust, OpenACC, and CUDA
- Examples: **SYCL**, **Kokkos**, Alpaka, Futhark, C++AMP, …

An Alternative: Kokkos

From Sandia National Laboratories

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

```
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);
});
```

→ https://github.com/kokkos/kokkos/

Another Alternative: SYCL

- Extension of/upon OpenCL
- With buffers, queues, accessors, lambdas, …
- Part of programming model for Aurora's Intel GPUs

→ khronos.org/sycl/

```
class mySaxpy;

std::vector<double> h_x(length), h_y(length);
// Fill x, y
cl::sycl::buffer<double, 1> d_x(h_x), d_y(h_y);

cl::sycl::queue queue;

queue.submit([&] (cl::sycl::handler& cgh) {
    auto x_acc = d_x.get_access<cl::sycl::access::mode::read>(cgh);
    auto y_acc = d_y.get_access<cl::sycl::access::mode::read>(cgh);

    cgh.parallel_for<class mySaxpy>(length,
        [=] (cl::sycl::id<1> idx) {
            y_acc[idx] = a * x_acc[idx] + y_acc[idx];
    });
});
```


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Programming GPUs

Tools

GPU Tools

The helpful helpers helping helpless (and others)

- NVIDIA

`cuda-gdb`	GDB-like command line utility for debugging
`cuda-memcheck`	Like Valgrind's `memcheck`, for checking errors in memory accesses
Nsight	IDE for GPU developing, based on Eclipse (Linux, OS X) or Visual Studio (Windows)
`nvprof`	**Command line profiler, including detailed performance counters**
Visual Profiler	**Timeline profiling and annotated performance experiments**
New	Nsight Systems (timeline), Nsight Compute (kernel analysis)

- OpenCL/HIP:

CodeXL	Debugging, profiling.
ROCmGDB	AMD's GDB symbolic debugger
RadeonComputeProfiler	Profiler for OpenCL and ROCm

nvprof

Command that line

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*, int, int)
  0.58%  1.5428ms         2  771.39us  764.65us  778.12us  [CUDA memcpy HtoD]
  0.23%  599.40us         1  599.40us  599.40us  599.40us  [CUDA 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
```

nvprof

Command that line

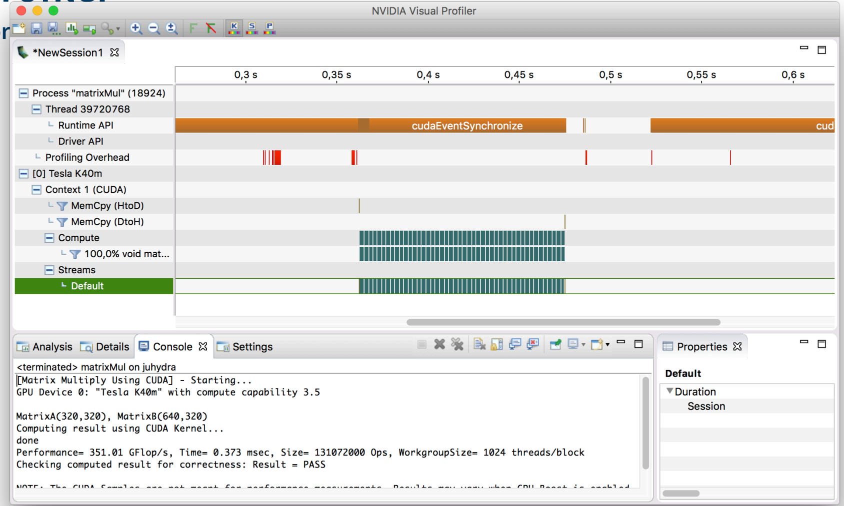
With metrics: `nvprof --metrics flop_sp_efficiency ./app`

```
$ 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)...
Performance= 26.61 GFlop/s, Time= 80.697 msec, Size= 2147483648 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
==37122== Profiling application: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
==37122== Profiling result:
==37122== Metric result:
Invocations                          Metric Name                        Metric Description         Min         Max         Avg
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%      23.40%      23.15%
```

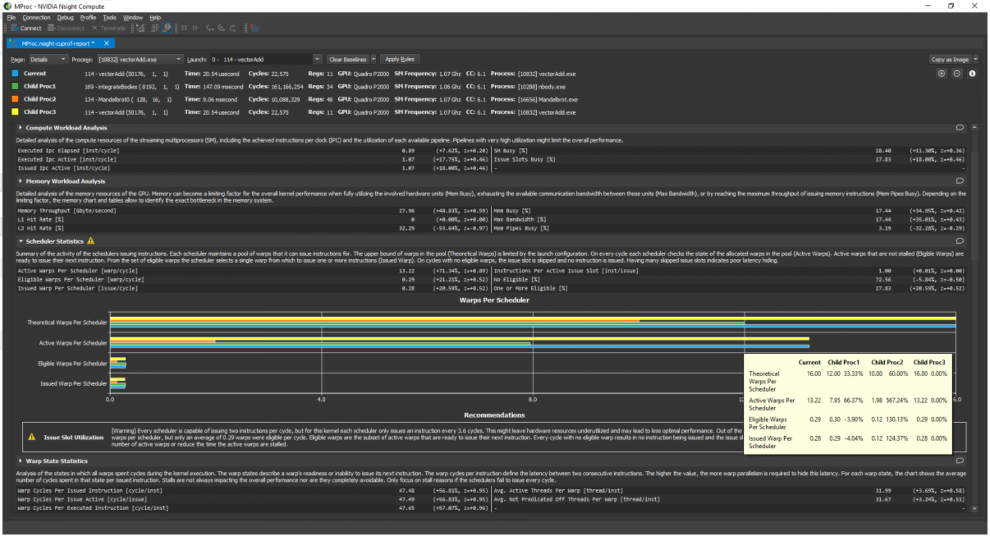
Visual Profiler

Your new favor



New Tools

20/20 N



Wrapping Up

Summary

Summary of Acceleration Possibilities

Application

Libraries | OpenACC Directives | Programming Languages

Drop-in Acceleration

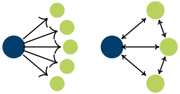
Easy Acceleration

Flexible Acceleration

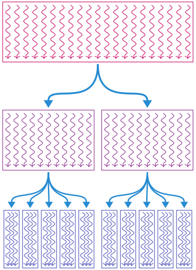
Advanced Topics

So much more interesting things to show!

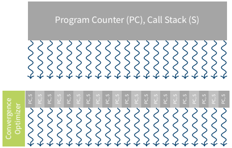
- Memory spaces (shared, pinned, …); memory transfer optimization
- Atomic  operations
- Optimize applications for GPU architecture (access patterns, streams)
- Drop-in BLAS acceleration with NVBLAS (`$LD_PRELOAD`)
- Cooperative groups, independent thread progress
- Half precision FP16
- Use multiple GPUs
  - On one node
  - Across many nodes → MPI



- …
- Some of that: Addressed at **dedicated training courses**



Cooperative Groups



Independent Thread Progress

Wrapping Up

GPUs on JUWELS/JURECA

Compilation

CUDA

- Module: `module load CUDA/10.1.105`
- Compile: `nvcc file.cu`
  Default host compiler: `g++`; use `nvcc_pgc++` for PGI compiler
- cuBLAS: `g++ file.cpp -I$CUDA_HOME/include -L$CUDA_HOME/lib64 -lcublas -lcudart`

OpenACC

- Module: `module load PGI/19.3-GCC-8.3.0`
- Compile: `pgc++ -acc -ta=tesla file.cpp`

MPI

- Module: `module load MVAPICH2/2.3.1-GDR` (also needed: `GCC/8.3.0`)
  Enabled for CUDA (*CUDA-aware*); no need to copy data to host before transfer

Running

- Dedicated GPU partitions

  JUWELS

`--partition=gpus`	46 nodes (Job limits: $<$1 d)
`--partition=develgpus`	10 nodes (Job limits: $<$2 h, $\leq$ 2 nodes)

  JURECA

`--partition=gpus`	70 nodes (Job limits: $<$1 d, $\leq$ 32 nodes)
`--partition=develgpus`	4 nodes (Job limits: $<$2 h, $\leq$ 2 nodes)

- Needed: Resource configuration with `--gres`

  `--gres=gpu:4`

  `--gres=mem1024,gpu:2 --partition=vis` *only JURECA*

→ See online documentation

- Also: Online job reports (interactive, PDFs)

Example

- 96 tasks in total, running on 4 nodes
- Per node: 4 GPUs

```
#!/bin/bash -x
#SBATCH --nodes=4
#SBATCH --ntasks=96
#SBATCH --ntasks-per-node=24
#SBATCH --output=gpu-out.%j
#SBATCH --error=gpu-err.%j
#SBATCH --time=00:15:00

#SBATCH --partition=gpus
#SBATCH --gres=gpu:4

srun ./gpu-prog
```

Wrapping Up

Conclusion

Conclusion

- GPUs can improve your performance many-fold
- For a fitting, parallelizable application
- Libraries are easiest
- Direct programming (plain CUDA, HIP) is most powerful
- OpenACC/OpenMP is somewhere in between (and portal
- Many abstraction layers available (mostly using C++)
- There are many tools helping the programmer

→ Download JSC Guest Student OpenACC Hands-On 2019 at `http://bit.ly/gsp-oacc`

Thank you for your attention!

a.herten@fz-juelich.de

Appendix

Appendix

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
- SYCL Hello World, SYCL Vector Addition

Volta Performance

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1462 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.5
Peak Tensor TFLOPS[1]	NA	NA	NA	120
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

Figure: Tesla V100 performance characteristics in comparison [9]

Appendix

Glossary & References

Glossary I

AMD Manufacturer of CPUs and GPUs. 3, 33, 38, 42, 56, 71, 72, 78, 98, 100

API A programmatic interface to software by well-defined functions. Short for application programming interface. 51, 56, 99

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

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 3, 42, 49, 56, 57, 58, 70, 71, 72, 74, 87, 91, 99

DSL A Domain-Specific Language is a specialization of a more general language to a specific domain. 2, 73, 74

Glossary II

HIP GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 56, 71, 72, 78, 91

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 98

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. 7, 86, 88

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 8, 70, 86, 88

MPI The Message Passing Interface, a API definition for multi-node computing. 85, 87

NVIDIA US technology company creating GPUs. 3, 7, 8, 33, 38, 56, 71, 72, 78, 82, 94, 97, 98, 99, 100, 101

Glossary III

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with high bandwidth. 101

OpenACC Directive-based programming, primarily for many-core machines. 49, 52, 53, 54, 74, 87, 91

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 3, 49, 56, 76, 78

OpenGL The *Open Graphics Library*, an API for rendering graphics across different hardware architectures. 3

OpenMP Directive-based programming, primarily for multi-threaded machines. 49, 52, 91

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 100

Glossary IV

POWER8 Version 8 of IBM's POWERprocessor, available also under the OpenPOWER Foundation. 99

ROCm AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 56, 78

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 35, 57, 58

Tesla The GPU product line for general purpose computing computing of NVIDIA. 7, 8, 33, 70

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 42, 49

Glossary V

V100 A large GPU with the Volta architecture from NVIDIA. It employs NVLink 2 as its interconnect and has fast *HBM2* memory. Additionally, it features *Tensorcores* for Deep Learning and Independent Thread Scheduling. 33, 70

Volta GPU architecture from NVIDIA (announced 2017). 25, 33, 101

References I

[2] Kenneth E. Hoff III et al. "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. ISBN: 0-201-48560-5. DOI: `10.1145/311535.311567`. URL: `http://dx.doi.org/10.1145/311535.311567` (page 3).

[3] 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 3).

[4] Jack Dongarra et al. *TOP500*. June 2019. URL: `https://www.top500.org/lists/2019/06/` (page 3).

References II

[5] Jack Dongarra et al. *Green500*. June 2019. URL: https://www.top500.org/green500/lists/2019/06/ (page 3).

[6] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (pages 4, 5).

[10] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 36, 37, 41).

References: Images, Graphics I

[1] Igor Ovsyannykov. *Yarn*. Freely available at Unsplash. URL: https://unsplash.com/photos/hvILKk7SlH4.

[7] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/ (page 10).

[8] Shearings Holidays. *Picture: Shearings coach 636*. URL: https://www.flickr.com/photos/shearings/13583388025/ (page 10).

[9] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf (pages 23, 24, 95).