COMPUTING WITH GPUS AT JSC

ESM SYMPOSIUM JSC 2019

28 May 2019 | Andreas Herten | Forschungszentrum Jülich ***Handout Version***

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Outline

2020: Booster!

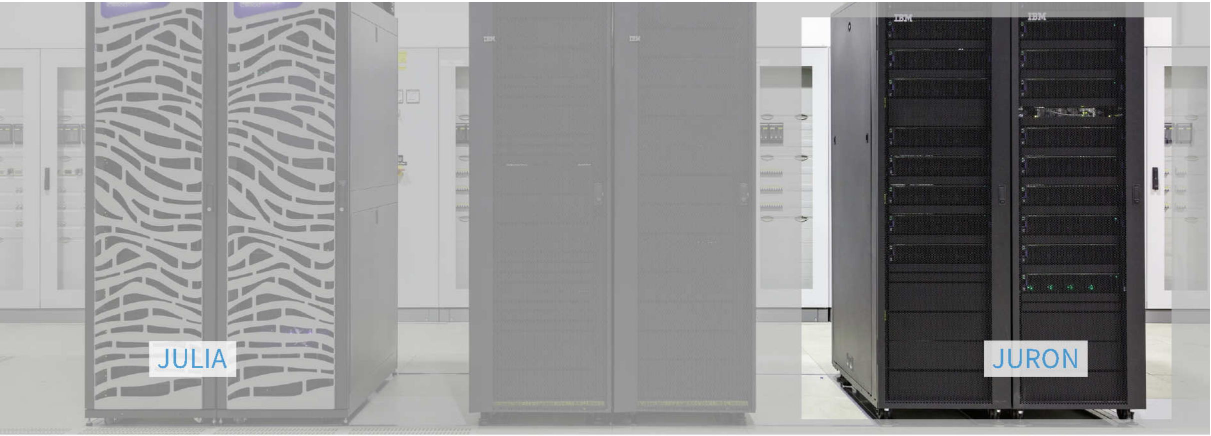
JUWELS – Jülich's New Large System

- 2500 nodes with Intel Xeon CPUs (2 $\times$ 24 cores)
- 46 + 10 nodes with 4 NVIDIA Tesla V100 cards
- 10.4 (CPU) + 1.6 (GPU) PFLOP/s peak performance (Top500: #26)

JURECA – Jülich's Multi-Purpose Supercomputer

- 1872 nodes with Intel Xeon E5 CPUs (2 × 12 cores)
- 75 nodes with 2 NVIDIA Tesla K80 cards (look like 4 GPUs)
- JURECA Booster: 1640 nodes with Intel Xeon Phi *Knights Landing*
- 1.8 (CPU) + 0.44 (GPU) + 5 (KNL) PFLOP/s peak performance (Top500: #44)
- Mellanox EDR InfiniBand

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE



JURON – A Human Brain Project *Pilot System*

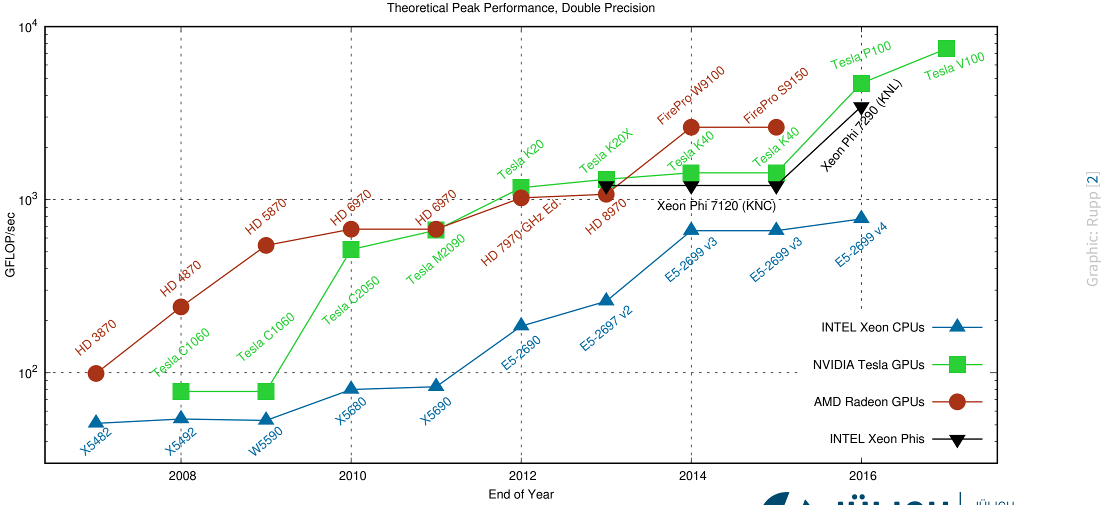
- 18 nodes with IBM POWER8*NVL* CPUs (2 × 10 cores)
- Per Node: 4 NVIDIA Tesla P100 cards (16 GB HBM2 memory), connected via NVLink
- GPU: 0.38 PFLOP/s peak performance

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Why?

Status Quo Across Architectures

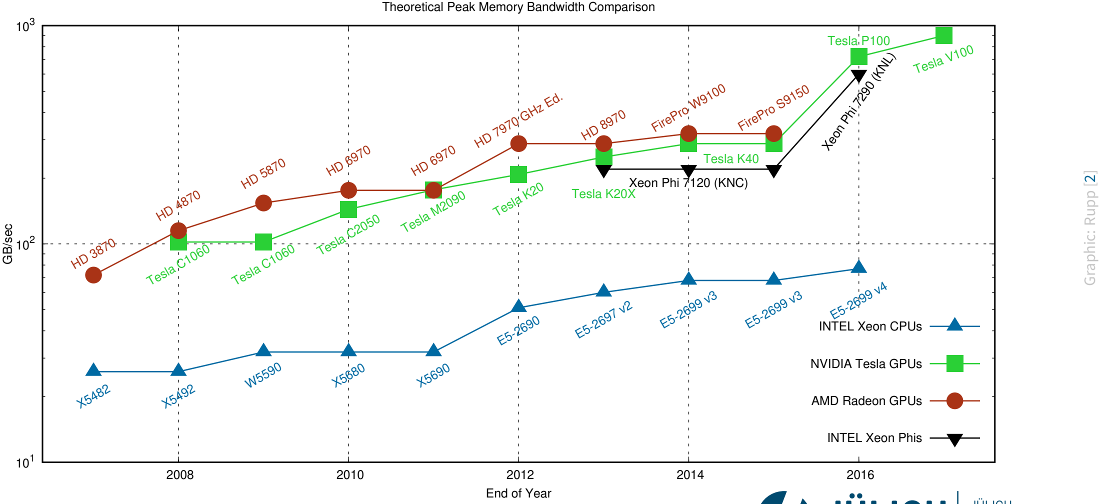
Memory Bandwidth



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Status Quo Across Architectures

Memory Bandwidth



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

CPU vs. GPU

A matter of specialties

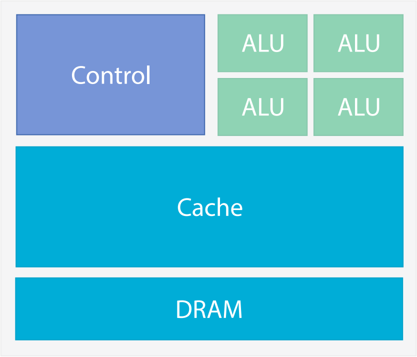
Transporting one

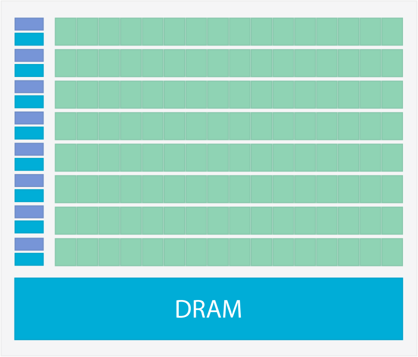
Transporting many

Graphics: Lee [3] and Bob Adams [4]

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

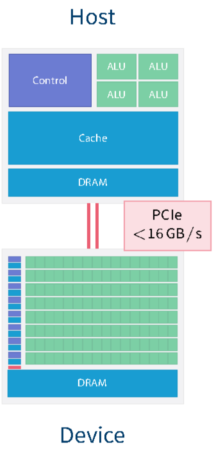
Memory

Memory

GPU memory ain't no CPU memory

- GPU: accelerator / extension card

→ Separate device from CPU
  Separate memory
- 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 Unified Memory

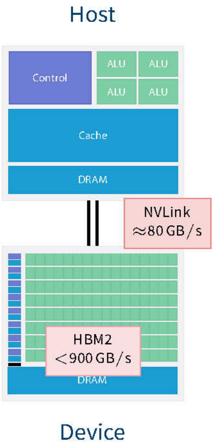
- Memory transfers need special consideration!

Do as little as possible!

- Formerly: Explicitly copy data to/from GPU

Now: Done automatically (performance…?)

- P100: 16 GB RAM, 720 GB/s; V100: 16 (32) GB RAM, 900 GB/s



GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

Memory

Async

Following different streams

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

→ Overlap tasks

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

- GPU needs to be fed: Schedule many computations
- CPU can do other work while GPU computes; synchronization

GPU Architecture

Overview

Aim: Hide Latency

Everything else follows

SIMT

Asynchronicity

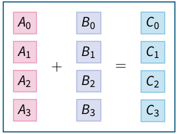
Memory

SIMT

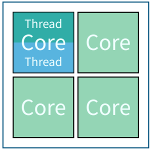
Of threads and warps

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

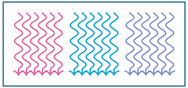
Vector



SMT



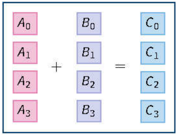
SIMT



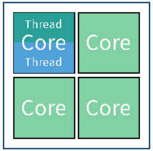
SIMT

Of t

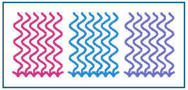
Vector



SMT



SIMT

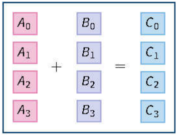


Graphics: volta-pictures

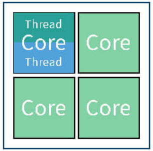
SIMT

Of t

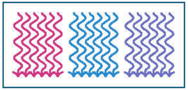
Vector



SMT



SIMT

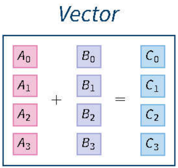


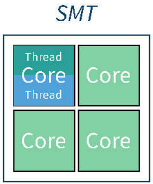
Graphics: volta-pictures

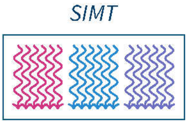
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

SIMT

Of t







Graphics: volta-pictures

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

Programming GPUs

JÜLICH SUPERCOMPUTING CENTRE

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!**

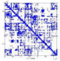
Use applications & libraries

Wizard: Breazell [6]

Libraries

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

Use applications & libraries



cuSPARSE

cuBLAS

cuDNN

OpenCV

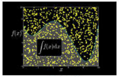
{ } ArrayFire

Thrust

Numba

theano

cuFFT



cuRAND

CUDA Math

Wizard: Breazell [6]

cuBLAS

Parallel algebra

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

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

cuBLAS

Code example

```
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

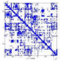
Finalize

Libraries

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

Use applications & libraries

cuBLAS

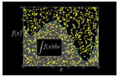


cuSPARSE

cuDNN

OpenCV

cuFFT



cuRAND

CUDA Math

Thrust

ArrayFire

Numba

theano

Wizard: Breazell [6]

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/`

Thrust

Code example with lambdas

```
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(),  // Input 1
    d_y.begin(),             // Input 2
    d_y.begin(),             // Output
    a * _1 + _2              // Operation
);

x = d_x;
```


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

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

Might eventually be re-merged into OpenMP standard

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 parallel loop copy(y) copyin(x)
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

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

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

OpenACC

Code example

```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);
```

GPU hands-on later!

Programming GPUs

Languages

Programming GPU Directly

Finally…

- Two solutions:

  OpenCL Open Computing Language by Khronos Group (Apple, IBM, NVIDIA, …) *2009*

  - Platform: Programming language (OpenCL C/C++), API, and compiler
  - Targets CPUs, GPUs, FPGAs, and other many-core machines
  - Fully open source
  - Different compilers available

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

  - **Platform: Drivers, programming language (CUDA C/C++), API, compiler, debuggers, profilers, …**
  - **Only NVIDIA GPUs**
  - **Compilation with `nvcc` (free, but not open)**
    `clang` has CUDA support, but CUDA needed for last step
  - **Also: CUDA Fortran**
- Choose what flavor you like, what colleagues/collaboration is using
- Hardest: Come up with parallelized algorithm

CUDA SAXPY

With runtime-managed data transfers

```
__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();
```

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

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

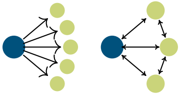
  → See appendix

- OpenCL: CodeXL (Open Source, GPUOpen/AMD) – debugging, profiling.

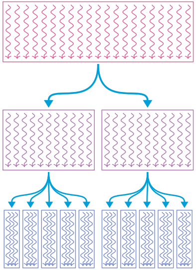
Advanced Topics

So much more interesting things to show!

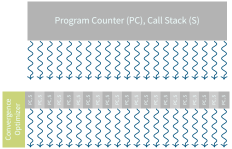
- Memory spaces (shared, pinned, …); memory transfer optimization
- Atomic operations
- Optimize applications for GPU architecture (access patterns, streams)
- Drop-in BLAS acceleration with NVBLAS (`$LD_PRELOAD`)
- Cooperative groups, independent thread progress
- Tensor Cores for Deep Learning (→ Appendix)
- Libraries, Abstractions: Kokkos (→ Appendix), Alpaka, SYCL, HIP …
- 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

Using GPUs on JURECA & JUWELS

Compiling

CUDA

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

OpenACC

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

MPI

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

Running

- Dedicated GPU partitions

  JUWELS

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

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
```

Conclusion, Resources

- GPUs provide highly-parallel computing power
- We have many devices installed at JSC, ready to be used!
- Training courses by JSC

  CUDA Course April 2020

  OpenACC Course 28 - 29 October 2019
- Generally: see online documentation and sc@fz-juelich.de
- Further consultation via our lab: *NVIDIA Application Lab*

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

Volta Performance

9- @ &*>+3/ 2	9- @ CD0	9- @ E D0	9- @ &100	9- @ V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	1B2	128	64	64
FP32 Cores / GPU	2880	30C2	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	B60	B6	1CB2	2560
TeDsor Cores / SM	EF	EF	EF	8
TeDsor Cores / GPU	EF	EF	EF	640
GPU Goost ClocH	810/8C5 MI J	1114 MI J	1480 MI J	1462 MI J
PeaHFP32 TFKL PS[1]	5	6M	10M	15
PeaHFP64 TFKL PS[1]	1MC	M1	5M	CM
PeaHTeDsor TFKL PS[1]	EF	EF	EF	120
TextNre UDOs	240	1B2	224	320
MePorQRDterSace	384TUOGVVW	384TUOGVVW	40B6TUOI GM2	40B6TUOI GM2
MePorQSOe	Up to 12 GG	Up to 24 GG	16 GG	16 GG
K2 CacXe SOe	1536 KG	30C2 KG	40B6 KG	6144 KG
SXareY MePorQSOe / SM	16 KG/32 KG/48 KG	B6 KG	64 KG	CoDSZNraUe Np to B6 KG
WeZOter FOe SOe / SM	256 KG	256 KG	256 KG	256KG
WeZOter FOe SOe / GPU	3840 KG	6144 KG	14336 KG	20480 KG
TVP	235 [atts	250 [atts	300 [atts	300 [atts
TraDsOtors	CM UOOD	8 UOOD	15M UOOD	21M UOOD
GPU VO SOe	551 PP\	601 PP\	610 PP\	815 PP\
MaDNSactNrOZ Process	28 DP	28 DP	16 DP FODF] T^	12 DP FFE

[1] PeaHTFKL PS rates are UaseY oDGPU Goost ClocH

Figure: Tesla V100 performance characteristics in comparison [**volta-pictures**]

Appendix

GPU Architecture: Tensor Cores

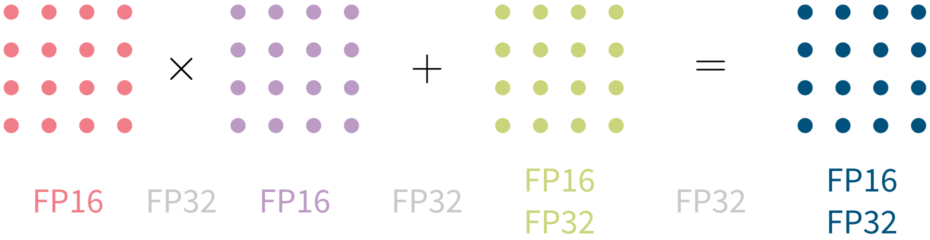
Tensor Cores

New since Volta

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

→ 64 floating-point FMA operations per clock (mixed precision)

- Turing architecture: `int`-based Tensor Cores



Appendix

GPU Programming: Abstraction Libraries/DSL

JÜLICH SUPERCOMPUTING CENTRE

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: **Kokkos**, Alpaka, Futhark, HIP, 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/

Appendix

NVIDIA GPU Tools

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`

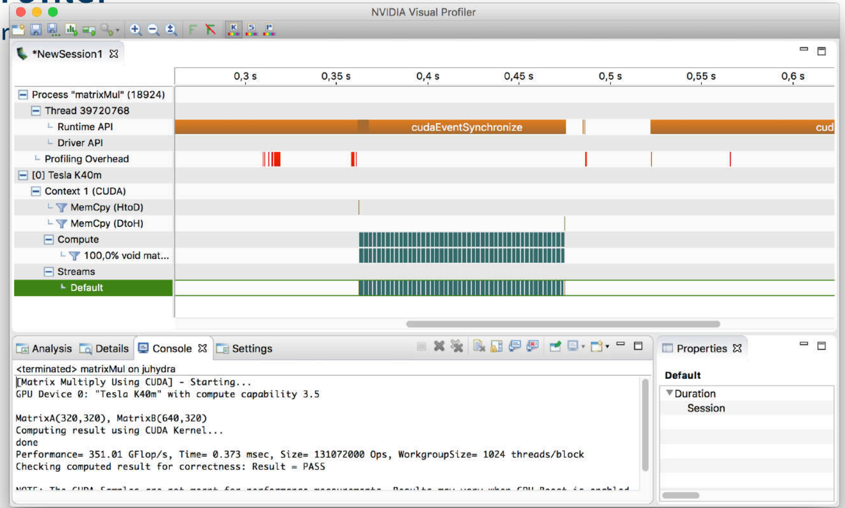
```
$ nvprof --metrics flop_sp_efficiency ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
[Matrix Multiply Using CUDA] - Starting...
==37122== NVPROF is profiling process 37122, command: ./matrixMul -wA=1024 -hA=1024 -wB=1024 -hB=1024
GPU Device 0: "Tesla P100-SXM2-16GB" with compute capability 6.0

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%
```


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Visual Profiler

Your new favor



Appendix

Glossary & References

Glossary I

API
: A programmatic interface to software by well-defined functions. Short for application programming interface. 35, 41

CUDA
: Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 31, 41, 42, 47, 50, 58, 67

DSL
: A Domain-Specific Language is a specialization of a more general language to a specific domain. 52, 57, 58

JSC
: Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 2, 50, 65

JURECA
: A multi-purpose supercomputer with 1800 nodes at JSC. 2, 4, 46, 48

Glossary II

JURON One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. 5

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 2, 3, 46, 48

MPI The Message Passing Interface, a API definition for multi-node computing. 45, 47

NVIDIA US technology company creating GPUs. 3, 4, 5, 41, 44, 50, 52, 53, 60, 65, 66, 67, 68

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with high bandwidth. 5, 67

OpenACC Directive-based programming, primarily for many-core machines. 36, 37, 38, 39, 47, 50, 58

Glossary III

OpenCL The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 41, 44

OpenMP Directive-based programming, primarily for multi-threaded machines. 36

P100 A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast *HBM2* memory. 5

Pascal GPU architecture from NVIDIA (announced 2016). 67

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 67

POWER8 Version 8 of IBM's POWERprocessor, available also under the OpenPOWER Foundation. 5, 67

SAXPY Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 24, 42

Glossary IV

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

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 31

Volta GPU architecture from NVIDIA (announced 2017). 56

References I

[2] Karl Rupp. *Pictures: CPU/GPU Performance Comparison*. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (pages 7, 8).

[6] Wes Breazell. *Picture: Wizard*. URL: https://thenounproject.com/wes13/collection/its-a-wizards-world/ (pages 25, 26, 30).

References: Images, Graphics I

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

[3] Mark Lee. *Picture: kawasaki ninja*. URL: https://www.flickr.com/photos/pochacco20/39030210/. License: Creative Commons BY-ND 2.0 (page 9).

[4] Bob Adams. *Picture: Hylton Ross Mercedes Benz Irizar coach*. URL: https://www.flickr.com/photos/satransport/13197324714/. License: Creative Commons BY-SA 2.0 (page 9).

[5] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf.