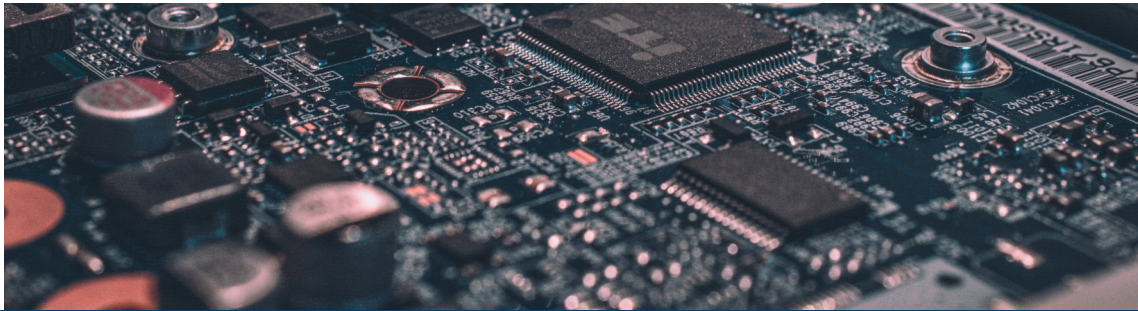




MANY WAYS TO GPUS

GPU INTRODUCTION

13 October 2022 | Andreas Herten, Kaveh Haghighi-Mood | Forschungszentrum Jülich

Outline

GPU Architecture

- Empirical Motivation

- Comparisons

- GPU Architecture

- Summary

Programming GPUs

- Libraries

- Directives

- CUDA C/C++

- Performance Analysis

Conclusion

References

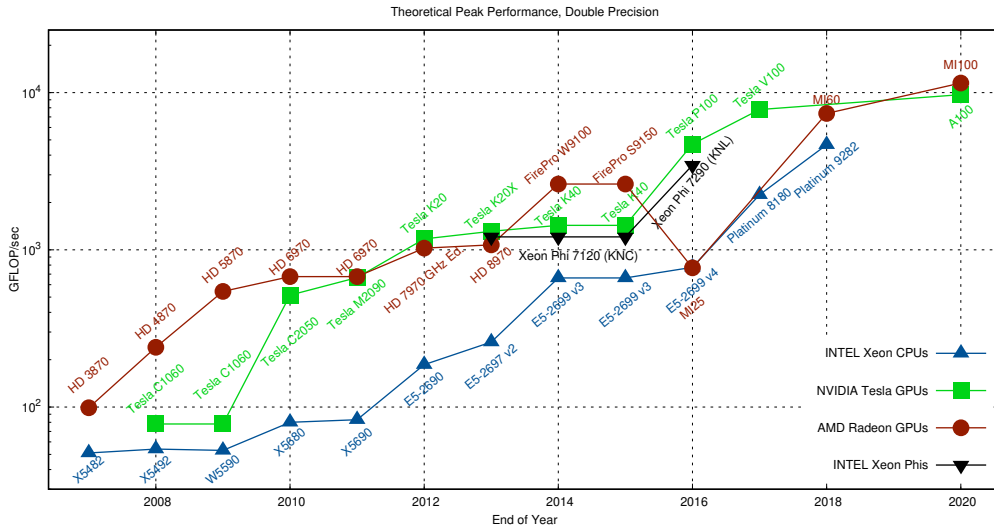
*Image references are collected in
References section at end of slides*

Title image: Debiève [1]

GPU Architecture

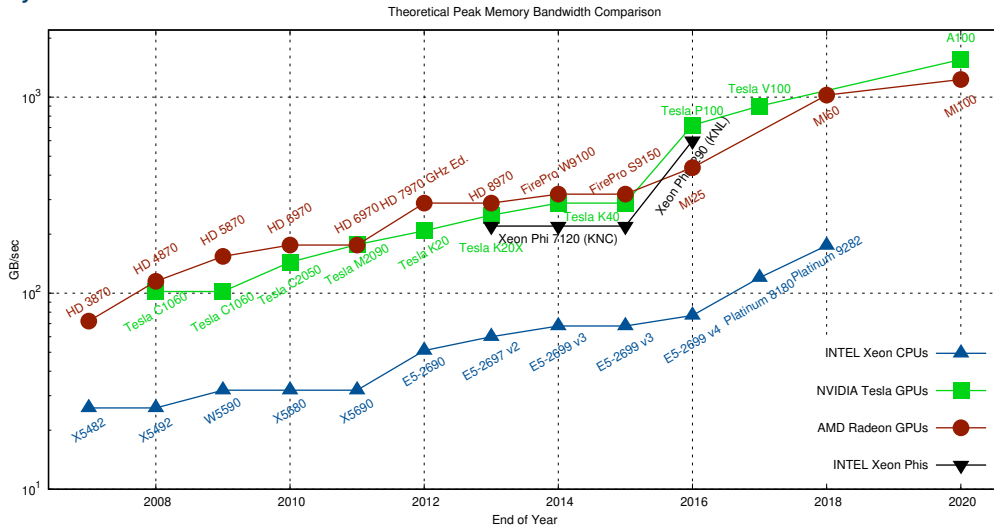
Status Quo Across Architectures

Performance



Status Quo Across Architectures

Memory Bandwidth



CPU vs. GPU

A matter of specialties



CPU vs. GPU

A matter of specialties



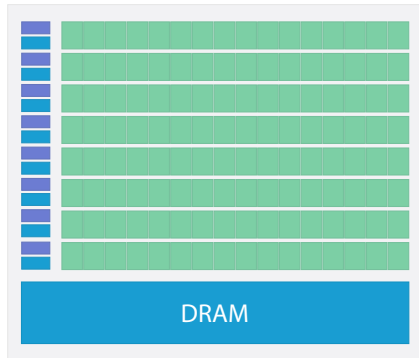
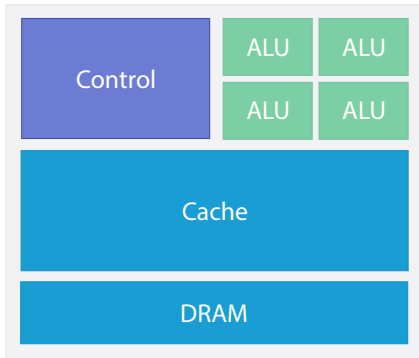
Transporting one



Transporting many

CPU vs. GPU

Chip

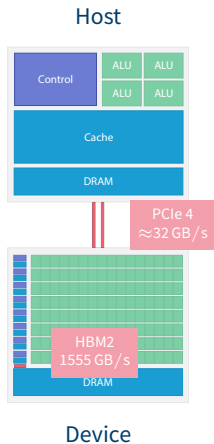


GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)

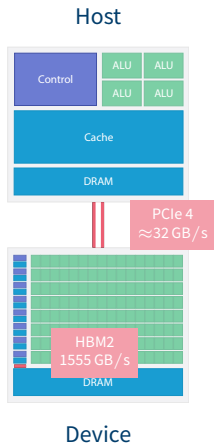


GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)

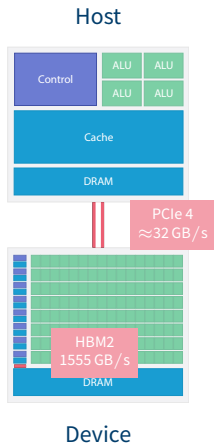


GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually



GPU Architecture Design

GPU optimized to **hide latency**

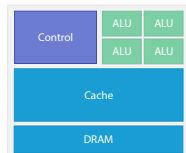
- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

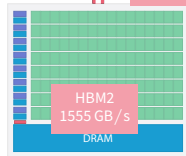
- Two engines: Overlap compute and copy



Host



PCIe 4
≈ 32 GB/s



Device

GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

- Two engines: Overlap compute and copy



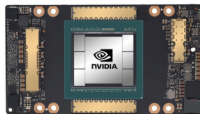
V100

32 GB RAM, 900 GB/s

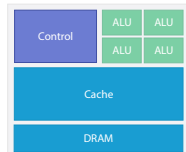


A100

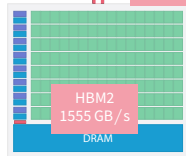
40 GB RAM, 1555 GB/s



Host



PCIe 4
≈ 32 GB/s



Device

GPU Architecture Design

GPU optimized to **hide latency**

- Memory

- GPU has small (40 GB), but high-speed memory 1555 GB/s
- Stage data to GPU memory: via PCIe 4 bus (32 GB/s)
- Stage automatically (*Unified Memory*), or manually

- Two engines: Overlap compute and copy



- SIMT

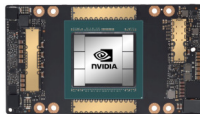
V100

32 GB RAM, 900 GB/s

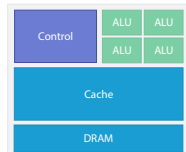


A100

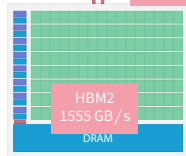
40 GB RAM, 1555 GB/s



Host



PCIe 4
≈ 32 GB/s



Device

SIMT

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Scalar

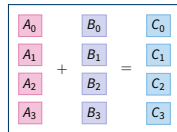
A_0	+	B_0	=	C_0
A_1	+	B_1	=	C_1
A_2	+	B_2	=	C_2
A_3	+	B_3	=	C_3

SIMT

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector

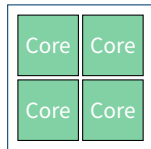
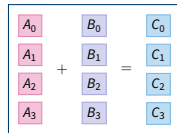


SIMT

$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector

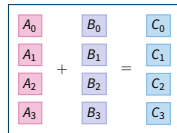


SIMT

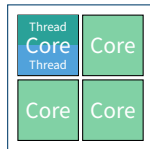
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

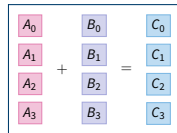


SIMT

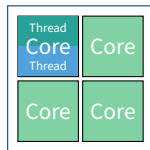
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

Vector



SMT

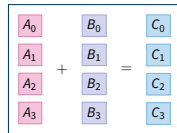


SIMT

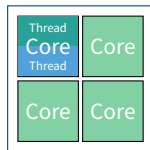
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

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

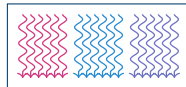
Vector



SMT



SIMT

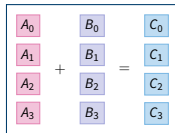


SIMT

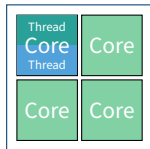
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$

- 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 

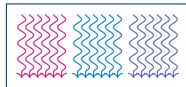
Vector



SMT

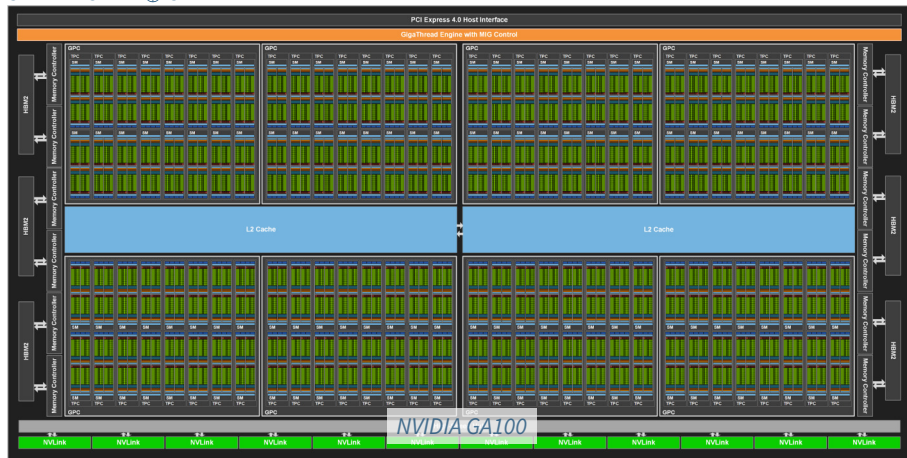


SIMT



SIMT

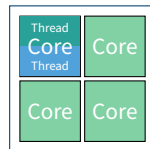
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



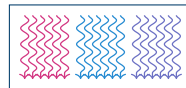
Vector

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

SMT



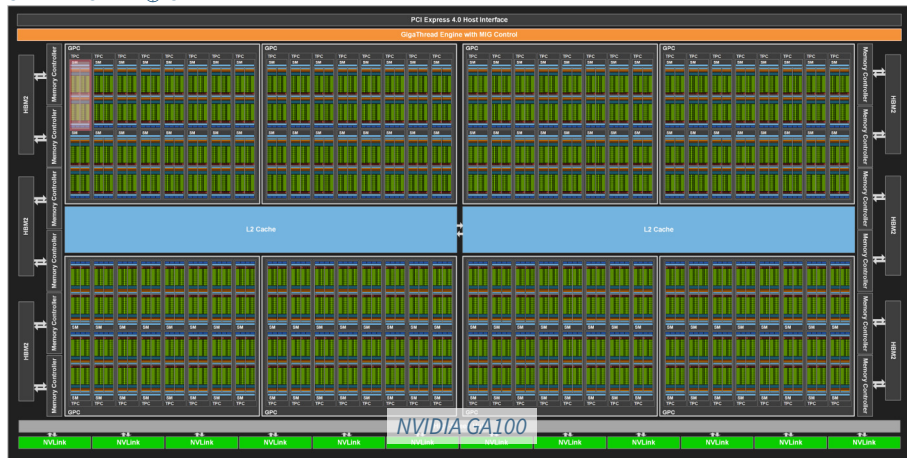
SIMT



Graphics: Nvidia Corporation [6]

SIMT

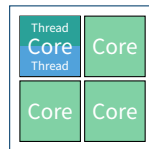
$$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$$



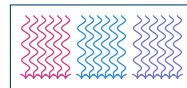
Vector

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

SMT



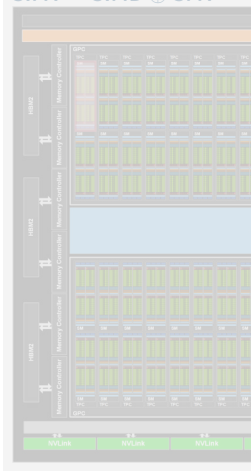
SIMT



Graphics: Nvidia Corporation [6]

SIMT

$\text{SIMT} = \text{SIMD} \oplus \text{SMT}$



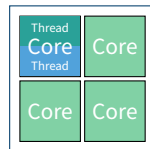
Multiprocessor



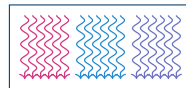
Vector

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

SMT



SIMT

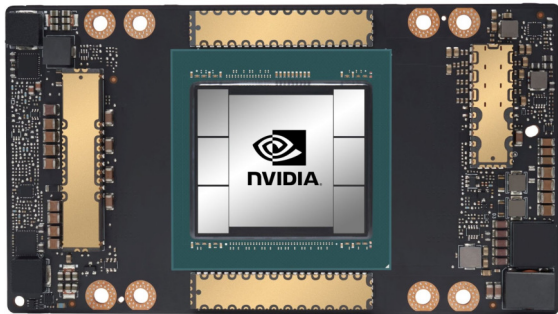


Graphics: Nvidia Corporation [6]

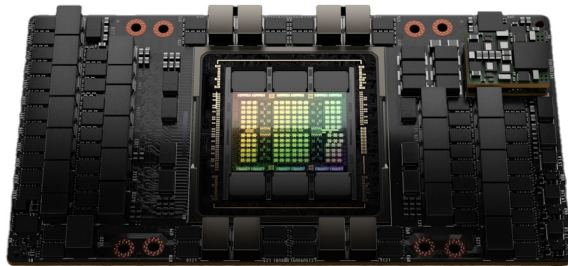
A100 vs H100

Comparison of current vs. next generation

A100



H100



A100 vs H100

Comparison of current vs. next generation

A100



H100



A100 vs H100

Comparison of current vs. next generation

A100

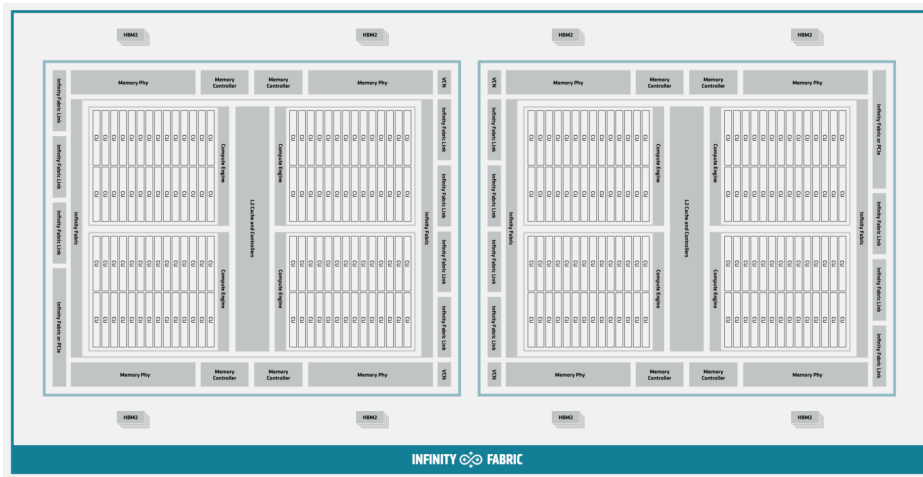


H100



AMD Instinct MI250

One GPU with Two Chiplets



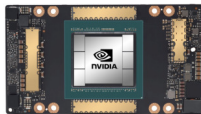
CPU vs. GPU

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

State of the GPU

C* C/C++

● Full vendor support

■ Vendor support, but not (yet) entirely comprehensive

◐ Indirect, but comprehensive support, by vendor

▲ Comprehensive support, but not by vendor

★ Limited, probably indirect support – but at least some

/ No direct support available, but of course one could ISO-C-bind your way through it or directly link the libraries

	CUDA		HIP		SYCL		OpenACC		OpenMP	
	C*	F	C*	F	C*	F	C*	F	C*	F
NVIDIA	●1	●2	▲3	/4	▲5	/6	●7	●8	■9	■10
AMD	◐11	★12	●13	/4	◐14	/6	▲15	▲/★16	●17	●17
Intel	◐18	/19	★20	/4	●21	/6	★22	★22	●24	●24

	Standard		Kokkos		ALPAKA		Python
	C*	F	C*	F	C*	F	
NVIDIA	●25	●26	▲27	★28	▲29	/30	■▲31
AMD	/32	/32	▲33	★28	▲34	/30	★35
Intel	/36	■37	▲38	★28	★39	/30	★40

State of the GPU: Footnotes I

- 1: CUDA C/C++, supported through CUDA Toolkit
- 2: CUDA Fortran, proprietary Fortran extension supported by NVIDIA HPC SDK
- 3: HIP programs can directly use NVIDIA GPUs via a CUDA backend; HIP is maintained by AMD
- 4: No such thing like HIP for Fortran
- 5: SYCL can be used on NVIDIA GPUs with *experimental* support either in [SYCL](#) directly or in [DPC++](#), or via [hipSYCL](#)
- 6: No such thing like SYCL for Fortran
- 7: OpenACC C/C++ supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC C compiler and Clacc
- 8: OpenACC Fortran supported on NVIDIA GPUs directly (and best) through NVIDIA HPC SDK; additional, somewhat limited support by GCC Fortran compiler and [Flacc](#)
- 9: OpenMP in C supported on NVIDIA GPUs through NVIDIA HPC SDK (but not full OpenMP feature set available), by GCC, and Clang
- 10: OpenMP in Fortran supported on NVIDIA GPUs through NVIDIA HPC SDK (but not full OpenMP feature set available), by GCC, and Flang
- 25: pSTL features supported on NVIDIA GPUs through NVIDIA HPC SDK

State of the GPU: Footnotes II

- 26: Standard Language parallel features supported on NVIDIA GPUs through NVIDIA HPC SDK
- 27: Kokkos supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 28: Kokkos is a C++ model, but at least the authors provided an ISO C Binding example for Fortran
- 29: Alpaka supports NVIDIA GPUs by calling CUDA as part of the compilation process
- 30: Alpaka is a C++ model
- 31: There is a vast community of offloading Python code to NVIDIA GPUs, like CuPy, Numba, cuNumeric, and many others; NVIDIA actively supports a lot of them, but has no direct product like *CUDA for Python*; so, the status is somewhere in between
- 11: [hipify](#) by AMD can translate CUDA calls to HIP calls which runs natively on AMD GPUs
- 12: AMD offers a Source-to-Source translator to convert some CUDA Fortran functionality to OpenMP for AMD GPUs ([gpufort](#)); in addition, there are ROCm library bindings for Fortran in [hipfort](#) OpenACC/CUDA Fortran Source-to-Source translator [gpufort](#): <https://github.com/ROCmSoftwarePlatform/gpufort>
- 13: HIP is the preferred native programming model for AMD GPUs
- 14: SYCL can use AMD GPUs, for example with [hipSYCL](#) or [DPC++ for HIP AMD](#)

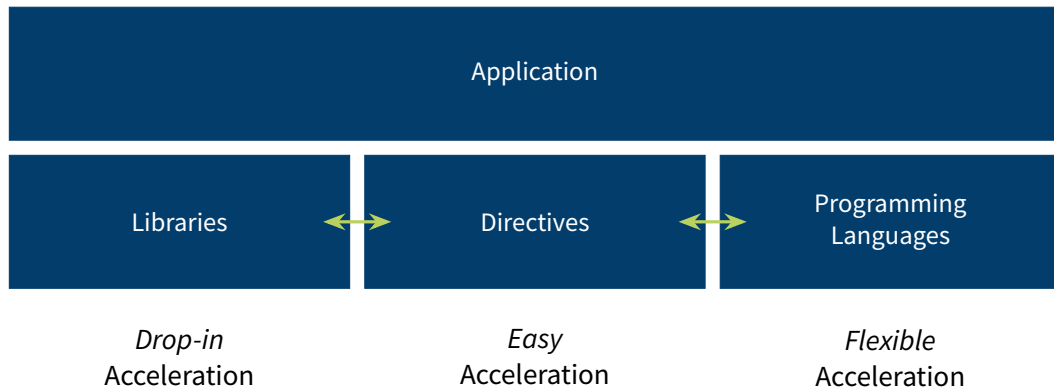
State of the GPU: Footnotes III

- 15: OpenACC C/C++ can be used on AMD GPUs via GCC or Clacc; also, [Intel's OpenACC to OpenMP Source-to-Source translator](#) can be used to generate OpenMP directives from OpenACC directives
- 16: OpenACC Fortran can be used on AMD GPUs via GCC; also, AMD's [gpubfort](#) Source-to-Source translator can move OpenACC Fortran code to OpenMP Fortran code, and also Intel's translator can work
- 17: AMD offers a dedicated, Clang-based compiler for using OpenMP on AMD GPUs: AOMP; it supports both C/C++ (Clang) and Fortran (Flang, [example](#))
- 32: Currently, no (known) way to launch Standard-based parallel algorithms on AMD GPUs
- 33: Kokkos supports AMD GPUs through HIP
- 34: Alpaka supports AMD GPUs through HIP
- 35: AMD does not officially support GPU programming with Python (also not semi-officially like NVIDIA), but third-party support is available, for example through [Numba](#) or a [HIP version of CuPy](#)
- 18: [SYCLomatic](#) translates CUDA code to SYCL code, allowing it to run on Intel GPUs; also, Intel's [DPC++ Compatibility Tool](#) can transform CUDA to SYCL
- 19: No direct support, only via ISO C bindings, but at least an example can be [found on GitHub](#); it's pretty scarce and not by Intel itself, though
- 20: [CHIP-SPV](#) supports mapping CUDA and HIP to OpenCL and Intel's Level Zero, making it run on Intel GPUs

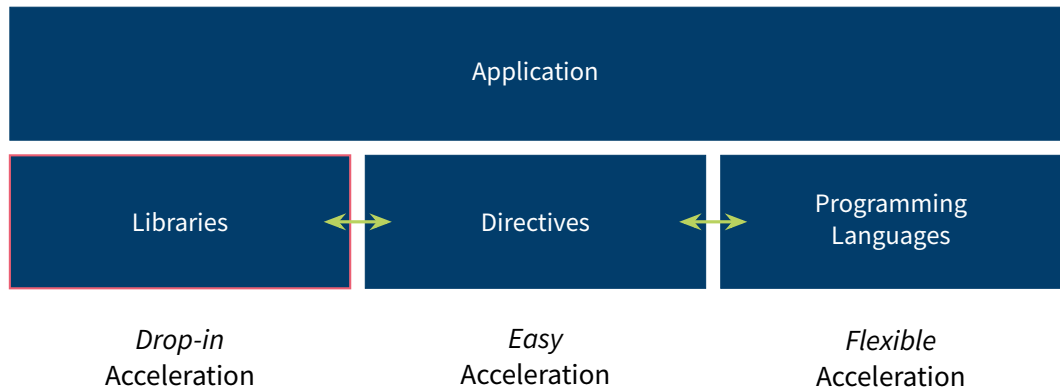
State of the GPU: Footnotes IV

- 21: SYCL is the prime programming model for Intel GPUs; actually, SYCL is only a standard, while Intel's implementation of it is called DPC++ (*Data Parallel C++*), which extends the SYCL standard in places
- 22: OpenACC can be used on Intel GPUs by translating the code to OpenMP with [Intel's Source-to-Source translator](#)
- 24: Intel has [extensive support for OpenMP](#) through their latest compilers
- 36: Currently, no (known) way to launch Standard-based parallel algorithms on Intel GPUs
- 37: With [Intel oneAPI 2022.3](#), Intel supports DO CONCURRENT with GPU offloading
- 38: Kokkos supports Intel GPUs through SYCL
- 39: [Alpaka v0.9.0](#) introduces experimental SYCL support
- 40: Not a lot of support available at the moment, but notably [DPNP](#), a SYCL-based drop-in replacement for Numpy

Summary of Acceleration Possibilities



Summary of Acceleration Possibilities



Programming GPUs

Libraries

Libraries

Programming GPUs is easy: Just don't!

Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries

Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries



Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries



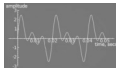
cuBLAS



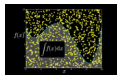
cuSPARSE



cuDNN



cuFFT



cuRAND



CUDA Math



{A} ARRAYFIRE

Numba

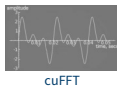


CuPy

Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries



Numba



CuPy



- 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);
```

Initialize

```
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);
```

Allocate GPU memory

Copy data to GPU

```
cublasSaxpy(n, a, d_x, 1, d_y, 1);
```

Call BLAS routine

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

Copy result to host

```
cudaFree(d_x); cudaFree(d_y);  
cublasDestroy(handle);
```

Finalize

Programming GPUs

Directives

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

```
#pragma acc loop
```

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

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

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

- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- Compiler interprets directives, creates according instructions

GPU Programming with Directives

Keepin' you portable

- Annotate serial source code by directives

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

- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- 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
- A little harder to debug

OpenACC / OpenMP

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

OpenACC / OpenMP

Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {  
    #pragma omp target map(to:x[0:n]) map(tofrom:y[0:n]) loop  
    for (int i = 0; i < n; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
float a = 42;  
int n = 10;  
float x[n], y[n];  
// fill x, y  
  
saxpy_acc(n, a, x, y);
```

Programming GPUs

CUDA C/C++

Programming GPUs Directly

Finally...

Programming GPUs 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

Programming GPUs 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; and more in NVIDIA HPC SDK

Programming GPUs 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; and more in NVIDIA HPC SDK

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

SYCL Intel's unified programming model for CPUs and GPUs (also: DPC++)

Programming GPUs 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; and more in NVIDIA HPC SDK

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

SYCL Intel's unified programming model for CPUs and GPUs (also: DPC++)

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

Programming GPUs 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; and more in NVIDIA HPC SDK

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

SYCL Intel's unified programming model for CPUs and GPUs (also: DPC++)

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



CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Thread



CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads



CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block



CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Block



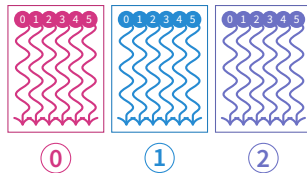
CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks



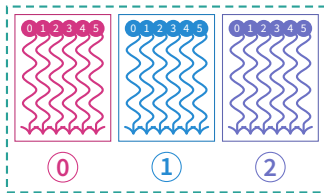
CUDA's Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid



CUDA's Parallel Model

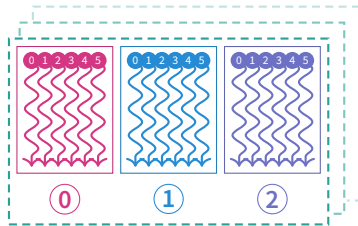
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



CUDA's Parallel Model

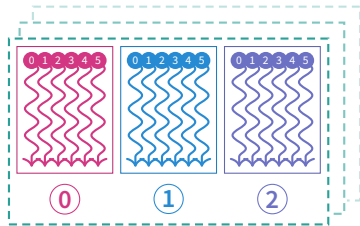
In software: Threads, Blocks

- Methods to exploit parallelism:

- Threads → Block

- Blocks → Grid

- Threads & blocks in 3D



- Parallel function: **kernel**

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

- Access own ID by global variables `threadIdx.x`, `blockIdx.y`, ...

- Execution entity: **threads**

- Lightweight → fast switching!

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

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();
```



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];  
}
```

Specify kernel

ID variables

Guard against
too many threads

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

```
// fill x, y
```

```
cudaMallocManaged(&x, n * sizeof(float));  
cudaMallocManaged(&y, n * sizeof(float));
```

Allocate GPU-capable
memory

Call kernel
2 blocks, each 5 threads

```
saxpy_cuda<<<2, 5>>>(n, a, x, y);
```

Wait for
kernel to finish

```
cudaDeviceSynchronize();
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
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

```
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

```
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

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

```
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

```
__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

```
__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];  
}
```

Programming GPUs

Performance Analysis

GPU Tools

The helpful helpers helping helpless (and others)

- NVIDIA

- `cuda-gdb` GDB-like command line utility for debugging

- `compute-sanitizer` Check memory accesses, race conditions, ...

- `Nsight` IDE for GPU developing, based on Eclipse (Linux, OS X) or Visual Studio (Windows) or VScode

- `Nsight Systems` GPU program profiler with timeline

- `Nsight Compute` GPU kernel profiler

- AMD

- `rocProf` Profiler for AMD's ROCm stack

- `uProf` Analyzer for AMD's CPUs and GPUs

Nsight Systems

CLI

```
$ nsys profile --stats=true ./poisson2d 10 # (shortened)
```

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
90.9	160,407,572	30	5,346,919.1	1,780	25,648,117	cuStreamSynchronize

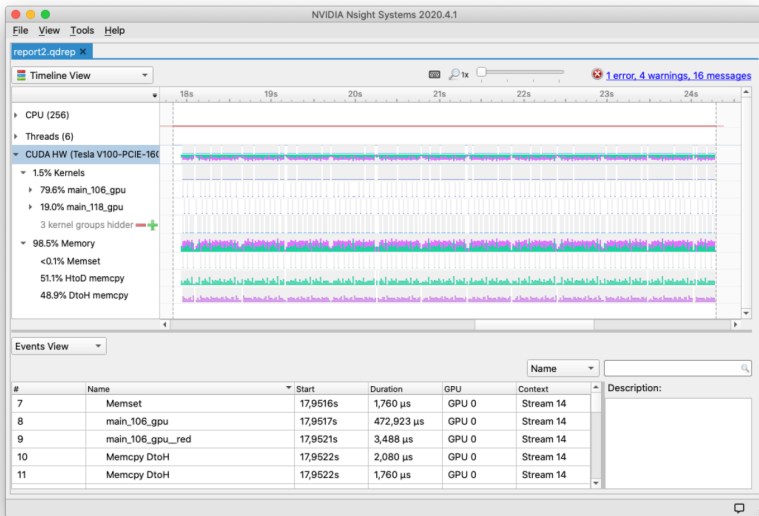
CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	158,686,617	10	15,868,661.7	14,525,819	25,652,783	main_106_gpu
0.0	25,120	10	2,512.0	2,304	3,680	main_106_gpu__red

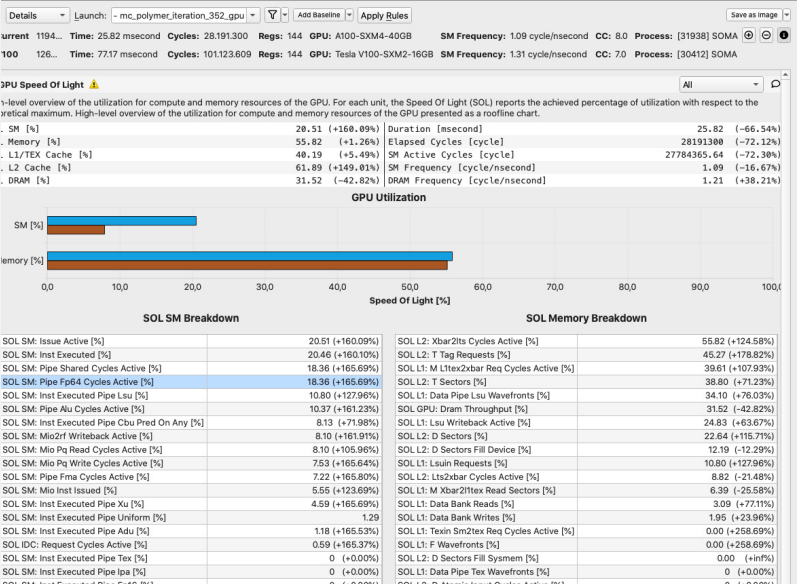


Nsight Systems

GUI



Nsight Compute GUI



Conclusion

Conclusion, Resources

- GPUs provide highly-parallel computing power
- Many ways to use them
 - Libraries** Best performance, but need to map
 - Directives** Easy to use, but needs to fit
 - Native** Most performance, but sometimes a little hard

Conclusion, Resources

- GPUs provide highly-parallel computing power
- Many ways to use them
 - Libraries** Best performance, but need to map
 - Directives** Easy to use, but needs to fit
 - Native** Most performance, but sometimes a little hard

Conclusion, Resources

- GPUs provide highly-parallel computing power
- Many ways to use them
 - Libraries** Best performance, but need to map
 - Directives** Easy to use, but needs to fit
 - Native** Most performance, but sometimes a little hard

Conclusion, Resources

- GPUs provide highly-parallel computing power
- Many ways to use them
 - Libraries** Best performance, but need to map
 - Directives** Easy to use, but needs to fit
 - Native** Most performance, but sometimes a little hard

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



Appendix

Appendix

Glossary

References

Glossary I

AMD Manufacturer of CPUs and GPUs. 54, 55, 56, 57, 58, 59, 92, 93

API A programmatic interface to software by well-defined functions. Short for application programming interface. 54, 55, 56, 57, 58, 59

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 93

HIP GPU programming model by AMD to target their own and NVIDIA GPUs with one combined language. Short for Heterogeneous-compute Interface for Portability. 54, 55, 56, 57, 58, 59

NVIDIA US technology company creating GPUs. 22, 23, 24, 54, 55, 56, 57, 58, 59, 81, 92

Glossary II

- OpenACC** Directive-based programming, primarily for many-core machines. 48, 49, 50, 51, 52
- OpenCL** The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 54, 55, 56, 57, 58, 59
- OpenMP** Directive-based programming, primarily for multi-threaded machines. 48, 49, 50, 51, 52
- ROCm** AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 54, 55, 56, 57, 58, 59
- SAXPY** Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 69, 70

Glossary III

CPU Central Processing Unit. 6, 7, 8, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 54, 55, 56, 57, 58, 59, 92, 93

GPU Graphics Processing Unit. 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 30, 38, 39, 40, 41, 42, 43, 44, 47, 48, 49, 50, 53, 54, 55, 56, 57, 58, 59, 70, 80, 81, 86, 87, 88, 89, 92, 93

SIMD Single Instruction, Multiple Data. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

SIMT Single Instruction, Multiple Threads. 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

SM Streaming Multiprocessor. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

SMT Simultaneous Multithreading. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

References: Images, Graphics I

- [1] Alexandre Debiève. *Title Graphic: Bowels of computer*. Freely available at Unsplash. URL: <https://unsplash.com/photos/F07JI1wj0tU> (page 2).
- [2] 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).
- [3] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (pages 6, 7).
- [4] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (pages 6, 7).

References: Images, Graphics II

- [5] Nvidia Corporation. *Pictures: Volta GPU*. Volta Architecture Whitepaper. URL: <https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf>.
- [6] Nvidia Corporation. *Pictures: Ampere GPU*. Nvidia Devblogs: NVIDIA Ampere Architecture In-Depth. URL: <https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/> (pages 22–24).
- [7] Nvidia Corporation. *Pictures: Hopper GPU*. Nvidia Developer Technical Blog: NVIDIA Hopper Architecture In-Depth. URL: <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>.
- [8] AMD. *Pictures: MI250 GPU*. AMD CDNA2 Architecture Whitepaper. URL: <https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf>.

References: Images, Graphics III

- [9] Wes Breazell. *Picture: Wizard*. URL: <https://thenounproject.com/wes13/collection/its-a-wizards-world/> (pages 39–43).