

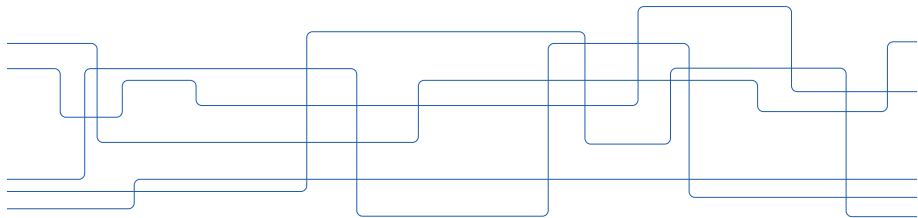


# Introduction to Accelerated HPC Architectures

**Dirk Pleiter**

**KTH, EECS, PDC and CST**

**2021-10-07**





# Overview

Introduction

Example: NVIDIA A100

CUDA Programming Model

OpenACC Programming Model

Conclusion



# Content

Introduction


Example: NVIDIA A100

CUDA Programming Model

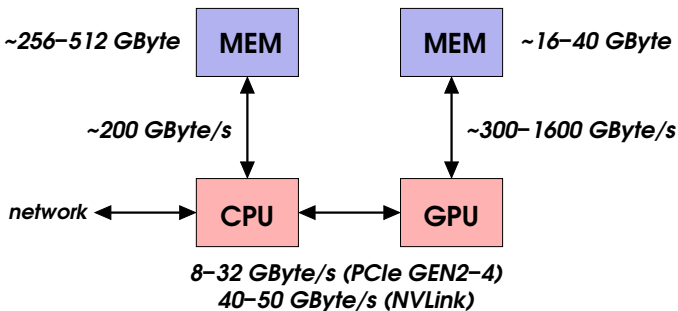
OpenACC Programming Model

Conclusion

# GPU Computing

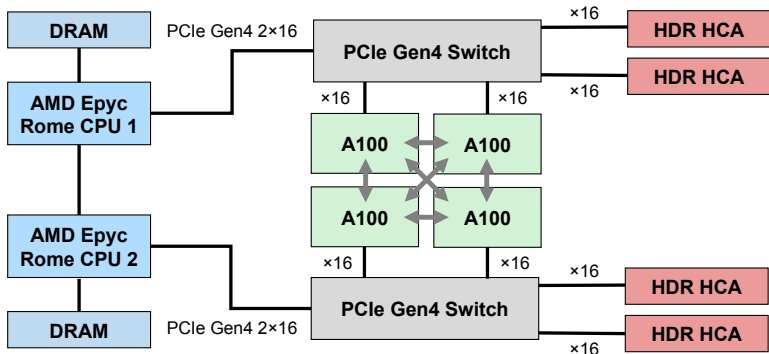
- ▶ GPU architecture is optimized for high degree of regular parallelism
- ▶ Relevant features
  - ▶ High memory bandwidth
  - ▶ Highly multithreaded
  - ▶ Hardware thread scheduling
- ▶ GPUs enabled for HPC
  - ▶ NVIDIA GPUs with CUDA, OpenACC, OpenMP, OpenCL support
  - ▶ Upcoming: AMD GPUs with OpenCL, OpenACC, HIP support
- ▶ GPUs are not stand-alone devices
  - ▶  Host CPU required

# Simple GPU Node Architecture



- ▶ Different storage devices
  - ▶ **Host memory** attached to CPU
  - ▶ **Device memory** attached to GPU
- ▶ Capacity and performance differ by  $O(5 \dots 10)$

# JUWELS Booster Node Architecture





# GPU Programming

- ▶ Accelerator model
  - ▶ Main application running on CPU
  - ▶ Kernel offload on GPU
  - ▶ Overall performance may suffer from Amdahl's law
- ▶ Typical kernel execution flow:
  - ▶ Allocate memory on GPU
  - ▶ Initialise memory on GPU or transfer data from host to GPU
  - ▶ Launch kernel
  - ▶ Transfer results from GPU to host
  - ▶ Deallocate memory on GPU

# GPU Programming Models (1/2)

- ▶ Native GPU programming
  - ▶ CUDA
    - ▶ Proprietary programming model from NVIDIA that extends C and C++
    - ▶ Also a version for Fortran available (CUDA Fortran)
  - ▶ HIP
    - ▶ C++-based programming model from AMD also supporting NVIDIA devices
  - ▶ OpenCL (Open Computing Language)
    - ▶ Open standard for parallel programming of heterogeneous systems
    - ▶ Support of different devices: CPU, GPU, DSP, FPGA
  - ▶ SYCL
    - ▶ Open standard for a higher-level programming model for various hardware accelerators



# GPU Programming Models (2/2)

- ▶ Use of libraries
  - ▶ Examples: cuBLAS, cuFFT, CUSP, MAGMA, Thrust
- ▶ Directive based programming
  - ▶ Definition of directives to specify loops and regions of code to be offloaded to GPU
  - ▶ Supported languages: C, C++, Fortran
  - ▶ OpenACC
    - ▶ OpenACC 3.1 released in November 2020
  - ▶ OpenMP
    - ▶ OpenMP 4.0 or newer required
    - ▶ OpenMP 5.1 released in November 2020



# Content

Introduction

Example: NVIDIA A100

CUDA Programming Model

OpenACC Programming Model

Conclusion

# Ampere GA100 Architecture Overview



[NVIDIA, 2020]

- ▶ NVIDIA Ampere architecture introduced in 2020
- ▶ Up to 128 Streaming Multiprocessors (SM)
- ▶ Shared L2 caches
- ▶ Memory (HBM), host interface (PCIe GEN4), interconnect (NVLink)
- ▶ Thread block scheduler (GigaThread Engine)

# GA100: Streaming Multiprocessor

- ▶ 4 processing blocks
- ▶ Processing
  - ▶  $4 \times 8$  FP64 units
  - ▶  $4 \times 16$  FP32 units
  - ▶  $4 \times 1$  Tensor Cores
  - ▶  $4 \times 8$  load/store units
- ▶ Data memory
  - ▶ 65,536 32-bit registers
  - ▶ 192 kByte shared memory/  
L1 cache
- ▶ Instructions
  - ▶ L1 and L0 instruction cache
  - ▶ 4 warp schedulers
  - ▶ 4 dispatch units



[NVIDIA, 2020]

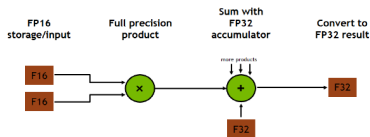
# GA100: FP64/FP32/FP16

- ▶ IEEE 754-2008 double-/single-precision FMA
- ▶ Performance due to massive parallelism
  - ▶ 2 Flop/cycle per floating-point unit
  - ▶ Double precision performance (assuming 108 SMs):
    - ▶  $(108 \text{ SM}) \cdot (32 \text{ FP64/SM}) \cdot (2 \text{ Flop/FP64/cycle}) = 6,912 \text{ Flop/cycle}$
    - ▶ At  $f = 1410 \text{ MHz}$  (boost clock): 9.7 TFlop/s
  - ▶ Support for  $2\times$  the throughput of FP32 operations

# GA100: Tensor Cores

- ▶ Each Tensor Core performs  $D \leftarrow A \times B + C$ 
  - ▶  $A$ ,  $B$ ,  $C$ , and  $D$  are  $4 \times 4$  matrices

$$D = \begin{matrix} \text{FP16 or FP32} \\ \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix} \\ \text{FP16} \end{matrix} \begin{matrix} \text{FP16} \\ \begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix} \\ \text{FP16} \end{matrix} + \begin{matrix} \text{FP16 or FP32} \\ \begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{pmatrix} \\ \text{FP16 or FP32} \end{matrix}$$



- ▶ Different data types supported
  - ▶ FP64, FP32, FP16, INT8, INT4
  - ▶ Options for using higher precision data types for accumulation
- ▶ Double precision performance
  - ▶  $(108 \text{ SM}) \cdot (1 \text{ TC/SM}) \cdot (128 \text{ Flop/cycle}) = 13,824 \text{ Flop/cycle}$
  - ▶ At  $f = 1410 \text{ MHz}$  (boost clock):  $19.5 \text{ TFlop/s}$

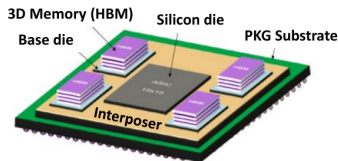
# GA100: Thread Scheduling

- ▶ **Thread block** = set of threads
- ▶ SM schedules threads in groups of 32 threads = **Warp**
- ▶ Scheduling parameters (for GA100):

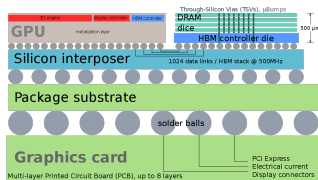
Number of threads/warp	32
Maximum number of warps/SM	64
Maximum number of threads/SM	2048
Maximum number of thread blocks/SM	32

# GA100: Memory Subsystem

- ▶ In-package HBM2 memory
- ▶ 5 memory stacks, 8 GByte/stack
- ▶ Very wide bus (2 · 512 bit) and relative low data rate (2.43 GT/s)
- ▶ Aggregate bandwidth: 1.555 TByte/s



[H. Jun (SK Hynix), 2013]

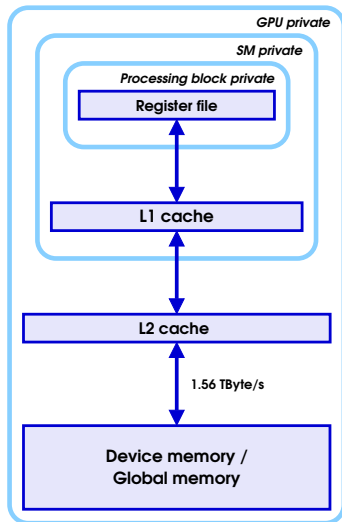


[Wikipedia]



# GA100: Memory Hierarchy

- ▶ Register file
  - ▶ Size: 256 kiByte/SM
- ▶ L1 data cache and shared memory
  - ▶ Size 192 kiByte/SM or 20 MiByte/GPU
- ▶ L2 cache
  - ▶ 40 MiByte/GPU



# Comparison of NVIDIA GPUs

	M2075	K20	P100	V100	A100
Clock [GHz]	1.15	0.71	1.33	1.31	1.10
Boost clock [GHz]			1.48	1.53	1.41
Number of SMs	14	13	56	80	108
Number of FP64 units	224	832	1,792	2,560	3,456
Peak FP64 [TFlop/s]	0.5	1.2	4.8	6.7	7.6
Peak FP64 [TFlop/s,TC]					15.1
Peak FP32 [TFlop/s]	1.0	3.5	9.5	13.4	15.1
Peak FP16/BF16 [TFlop/s,TC]				107	242
Memory capacity [GiByte]	6	5	16	16/32	40
Memory bandwidth [TByte/s]	0.15	0.21	0.72	0.90	1.56



# Content

Introduction

Example: NVIDIA A100

CUDA Programming Model

OpenACC Programming Model

Conclusion



# CUDA

- ▶ **CUDA** = Compute Unified Device Architecture
- ▶ Parallel programming language based on an extension of C and C++
- ▶ First SDK released in 2007
- ▶ Current version: 11.4.2 (September 2021)
- ▶ Language developed and controlled by NVIDIA
  - ▶ Only supported GPU devices are from NVIDIA
  - ▶ Available for x86, POWER and Arm host CPUs

Query device information:

```
#include <stdio.h>

int main()
{
    cudaDeviceProp deviceProp;

    if (cudaGetDeviceProperties(&deviceProp, 0) == cudaSuccess)
        printf(" Device: %s\n", deviceProp.name);

    return 0;
}
```

- ▶ CUDA source files must have extension `.cu`
- ▶ Compile using `nvcc`
- ▶ `nvcc` can be used to compile CUDA source files without linking and use `gcc/g++` to generate final executable

Query number of devices and select device #0:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int ndev;

    cudaGetDeviceCount(&ndev);
    if (ndev == 0) {
        fprintf(stderr, "No devices\n");
        exit(1);
    }
    printf("Number_of_devices:_%d\n", ndev);

    /* Select device #0 */
    cudaSetDevice(0);

    return 0;
}
```

- ▶ Device allocated at first access



# CUDA Memory Allocation

Allocate memory on device and copy data to/from device

```
cudaError_t cudaMalloc (void **devPtr, size_t size)
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
cudaError_t cudaFree (void *devPtr)
```

```
#define N 32

int main()
{
    int i;
    float ha[N]; /* Array on host */
    float *da; /* Pointer to array on device */

    for (i = 0; i < N; i++)
        ha[i] = i;

    cudaMalloc((void **) &da, N * sizeof(float));

    cudaMemcpy(da, ha, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(ha, da, N * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(da);

    return 0;
}
```

# CUDA Kernel

- ▶ **Kernel** = C function which is executed in parallel by  $N \geq 1$  different threads
- ▶ Declaration specifiers:

<code>--global--</code>	Function on device, callable from host
<code>--device--</code>	Function on device, callable from device
<code>--host--</code>	Function on host, callable from host (default)

- ▶ Kernel launch syntax:

```
mykernel<<<Dg,Db>>>(...)
```

- ▶ Dg: Size of the grid
- ▶ Db: Size of each block
- ▶ Each thread is given a unique ID: `threadIdx`
  - ▶ `threadIdx` is structure with 3 components: `x`, `y`, `z`





# CUDA Vector Copy

```
#include <stdio.h>
#define N 32

__global__ void copy(float *b, float *a)
{
    int i = threadIdx.x;
    b[i] = a[i];
}

int main()
{
    int i;
    float ha[N], hb[N];    /* Arrays on host */
    float *da, *db;       /* Pointer to arrays on device */

    for (i = 0; i < N; i++) ha[i] = i;

    cudaMalloc((void **) &da, N * sizeof(float));
    cudaMalloc((void **) &db, N * sizeof(float));

    cudaMemcpy(da, ha, N * sizeof(float), cudaMemcpyHostToDevice);

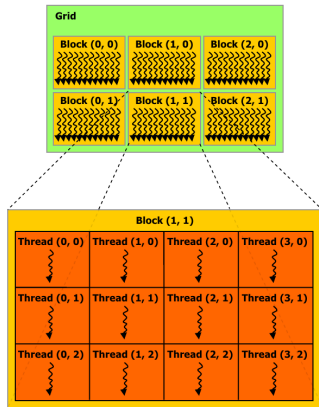
    dim3 Dg(1,1,1);
    dim3 Db(N,1,1);
    copy<<<Dg, Db>>>(db, da);

    cudaMemcpy(hb, db, N * sizeof(float), cudaMemcpyDeviceToHost);

    return 0;
}
```

# CUDA Thread Hierarchy

- ▶ **Block** = Set of threads
  - ▶ 1-, 2- or 3-dimensional index
  - ▶ Synchronization within threads possible
  - ▶ Maximum number of threads: 1024
  - ▶ Branching with penalty
  - ▶ Identifier: `threadIdx`
  
- ▶ **Grid** = Set of blocks
  - ▶ 1-, 2- or 3-dimensional index
  - ▶ Synchronization not possible
  - ▶ Branching without penalty
  - ▶ Identifier: `blockIdx`



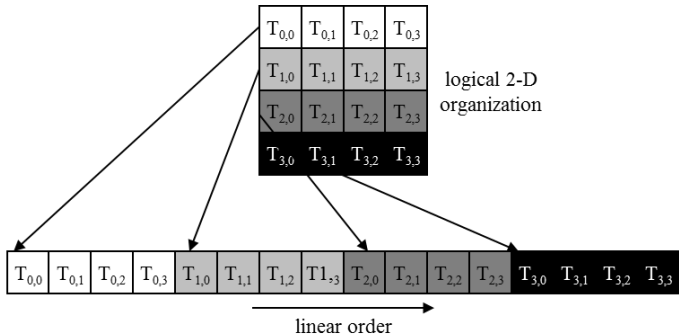


# Warps as Scheduling Units

- ▶ Each block is divided into thread warps running multiple threads
  - ▶ An implementation technique, not part of the CUDA programming model
  - ▶ Warps are scheduling units in SM
  - ▶ Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - ▶ The number of threads in a warp may vary in future generations

# Warps in Multi-dimensional Thread Blocks

- ▶ The thread blocks are first linearized into 1D in row major order
  - ▶ In x-dimension first, y-dimension next, and z-dimension last





# Blocks are Partitioned after Linearization

- ▶ Linearized thread blocks are partitioned
  - ▶ Thread indices within a warp are consecutive and increasing
  - ▶ Warp 0 starts with Thread 0
- ▶ Partitioning scheme is consistent across devices
  - ▶ Thus you can use this knowledge in control flow
  - ▶ However, the exact size of warps may change from generation to generation
- ▶ DO NOT rely on any ordering within or between warps
  - ▶ If there are any dependencies between threads, you must `__syncthreads()` to get correct results



# SMs are SIMD Processors

- ▶ SM acts as control unit for instruction fetch, decode, and control for multiple shared processing units
- ▶ All threads in a warp must execute the same instruction at any point in time
- ▶ This works efficiently if all threads follow the same control flow path
  - ▶ All if-then-else statements make the same decision
  - ▶ All loops iterate the same number of times

# Control Divergence

- ▶ Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - ▶ Some take the then-path and others take the else-path of an if-statement
  - ▶ Some threads take different number of loop iterations than others
- ▶ The execution of threads taking different paths are serialized in current GPUs
  - ▶ The control paths taken by the threads in a warp are traversed one at a time until there is no more
  - ▶ During the execution of each path, all threads taking that path will be executed in parallel
  - ▶ The number of different paths can be large when considering nested control flow statements

# Control Divergence Examples

- ▶ Divergence can arise when branch or loop condition is a function of thread indices
- ▶ Example kernel statement with divergence:

```
if (threadIdx.x > 2)
{ ... }
```

- ▶ This creates two different control paths for threads in a block
  - ▶ Decision granularity  $<$  warp size
  - ▶ Threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- ▶ Example without divergence:

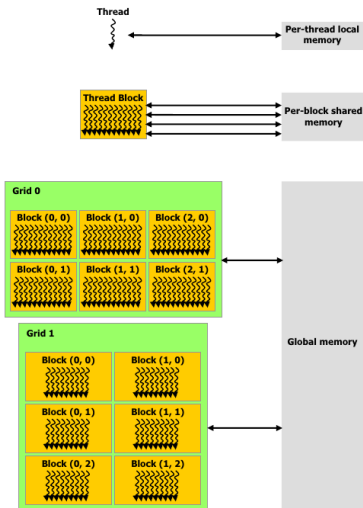
```
if (blockIdx.x > 2)
{ ... }
```

- ▶ Decision granularity is a multiple of blocks size
- ▶ All threads in any given warp follow the same path



# CUDA Memory Hierarchy

- ▶ Each thread can access registers and local memory
- ▶ Shared memory is visible to all threads of a block
- ▶ **Global memory space**
  - ▶ Resides in device memory
  - ▶ Accessible by all threads
- ▶ **Texture memory space**
  - ▶ Resides in device memory
  - ▶ Read via texture cache optimized for 2-d locality
- ▶ **Constant memory space**
  - ▶ Resides in device memory
  - ▶ Read-only, not dependent on thread ID



# CUDA Streams

- ▶ **Stream** = Sequence of operations that execute in issue-order on the GPU
- ▶ Streams can be used to improve device utilization
  - ▶ CUDA operations in different streams may run concurrently
  - ▶ CUDA operations from different streams may be interleaved
- ▶ Extended kernel launch syntax:

```
mykernel<<<Dg,Db,Ns,S>>>( ... )
```

- ▶ Dg: Size of the grid
- ▶ Db: Size of each block
- ▶ Ns: Bytes dynamically allocated in shared memory
- ▶ S : Associated stream (default 0)



# Content

Introduction

Example: NVIDIA A100

CUDA Programming Model

OpenACC Programming Model

Conclusion



# OpenACC: Introduction

- ▶ Approach: Provide guidance to compiler through directives
  - ▶ Ignored by compiler which does not understand OpenACC
- ▶ Programming model for CPUs and GPUs
- ▶ OpenACC elements:
  - ▶ Compiler directives
  - ▶ Library routines
  - ▶ Environment variables
- ▶ Portable across different architectures
- ▶ Compiler support
  - ▶ NVIDIA HPC Compilers (good)
  - ▶ gcc (improving)
  - ▶ clang (early technology)



# OpenACC parallel loop

- ▶ (Combined) construct that starts a parallel loop
- ▶ The reduction clause specifies a reduction operator on one or more variables

```
1 double sum = 0.0;
2
3 #pragma acc parallel loop
4 for (int i = 0; i < N; i++) {
5     x[i] = 1.0;
6     y[i] = 2.0;
7 }
8
9 #pragma acc parallel loop reduction(+:sum)
10 for (int i=0; i < N; i++) {
11     y[i] = i * x[i] + y[i];
12     sum += y[i];
13 }
```

# OpenACC kernel

- ▶ Construct defines a program region that is to be compiled into a sequence of kernels

```
1 double sum = 0.0;
2
3 #pragma acc kernels
4 {
5     for (int i = 0; i < N; i++) {
6         x[i] = 1.0;
7         y[i] = 2.0;
8     }
9
10    for (int i = 0; i < N; i++) {
11        y[i] = i * x[i] + y[i];
12        sum += y[i];
13    }
14 }
```



# OpenACC parallel **versus** kernel

- ▶ `kernel` construct
  - ▶ Identification of parallelisation opportunities is left to the compiler
  - ▶ Compiler has potentially more freedoms
  - ▶ Can cover large area of code with single directive
- ▶ `parallel` construct
  - ▶ User has to identify parallelisation opportunities
  - ▶ Will work where compiler fails to identify such opportunities
  - ▶ More explicit



# Content

Introduction

Example: NVIDIA A100

CUDA Programming Model

OpenACC Programming Model

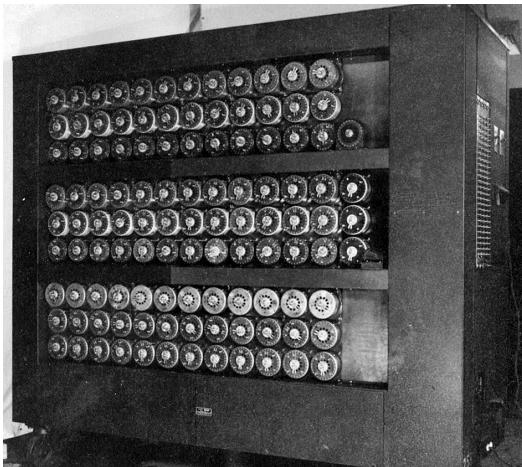
Conclusion



# Conclusion

- ▶ GPUs programming can be challenging
  - ▶ High level of parallelism and simplified architecture requires hardware aware programming
  - ▶ Note that optimisation strategies are typically also beneficial on CPUs
- ▶ Different programming models are available
  - ▶ A good choice is in the currently difficult as more GPU types are becoming available
- ▶ GPUs allow to significantly increase performance within a given power envelope resulting in an increasing number of GPU-accelerated systems

# Finish with an Architecture from Turing: Bombe



[United Kingdom Government, 1945]