Introduction to GPU computing and CUDA

Dorothea vom Bruch

GDR-InF Annual workshop

GPU Hands-on project

October 2020

Outline

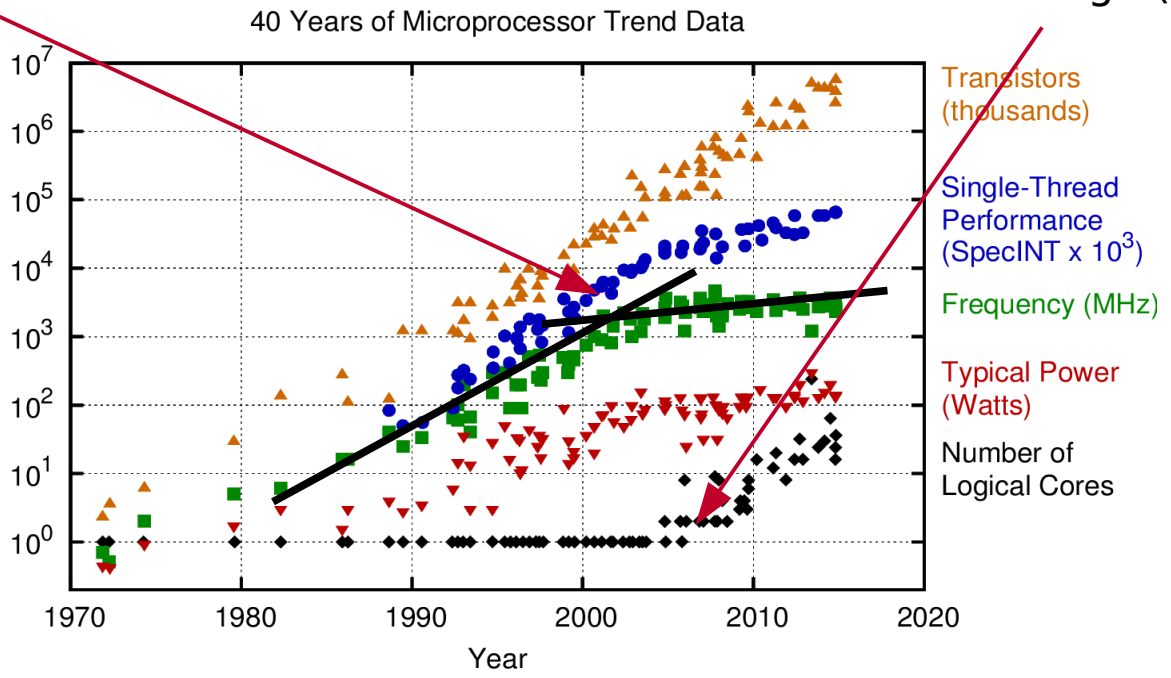
- Why GPUs?
- Parallel programming
- What is a GPU?
- Programming GPUs with CUDA

Why GPUs?

Moore’s law today

Clock speed stopped increasing due to heat limit

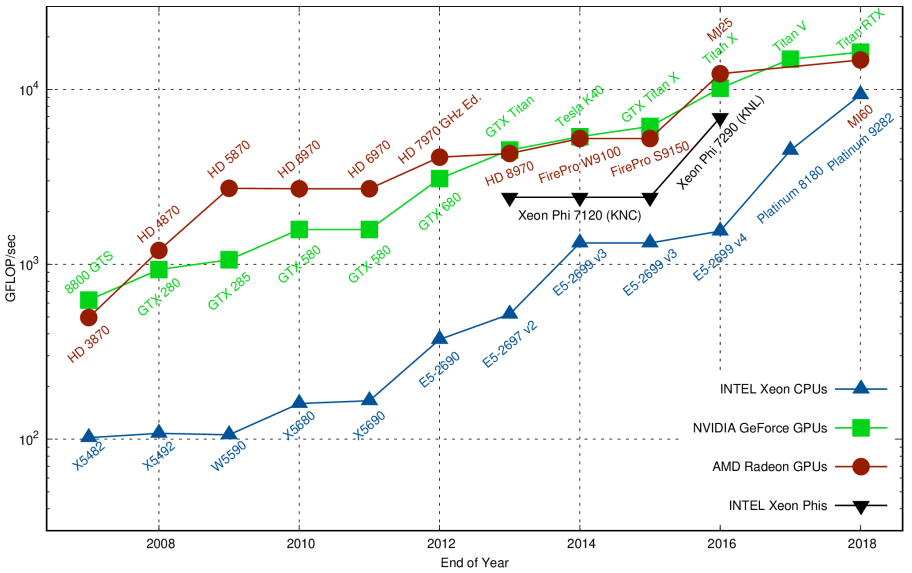
Multiple core processors emerge (Intel i7: 4 cores)



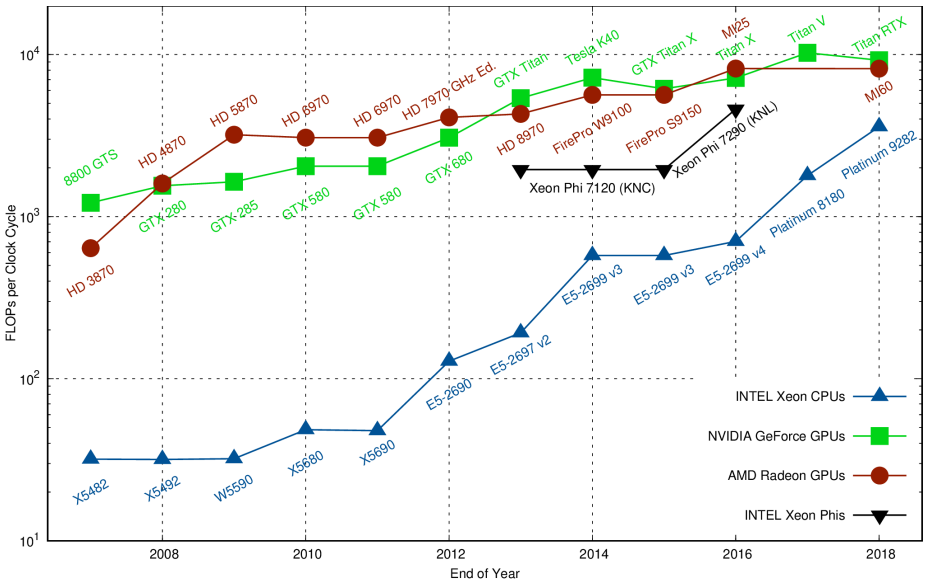
Evolution of peak FLOPs

- Gaming industry evolves steadily → continuous high demand for consumer GPUs
- With the trend for AI in many different areas → continuous demand for professional GPUs

Theoretical peak performance, single precision

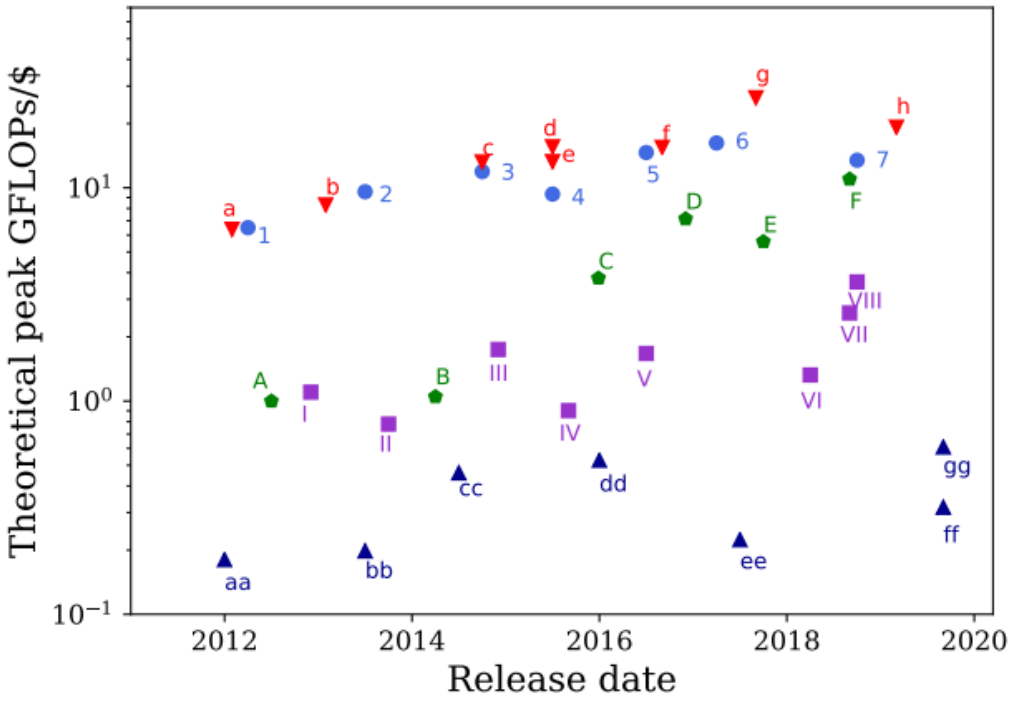


Theoretical peak FLOPs per clock cycle, single precision



Produced with https://github.com/karlrupp/cpu-gpu-mic-comparison

Theoretical FLOPs/$: GPUs & CPUs



Nvidia consumer GPUs
AMD consumer GPUs
Nvidia scientific GPUs
AMD scientific GPUs
Intel & AMD CPUs

a: RadeonHD7950
b: RadeonHD8950
c: RadeonR9285
d: RadeonR9390
e: RadeonR9FuryX
f: RadeonRX4604GB
g: RadeonRXVega56
h: RadeonVII

A: FireProW9000
B: FireProW9100
C: FireProW4300
D: RadeonProWX7100
E: RadeonProWX9100
F: RadeonProWX8200

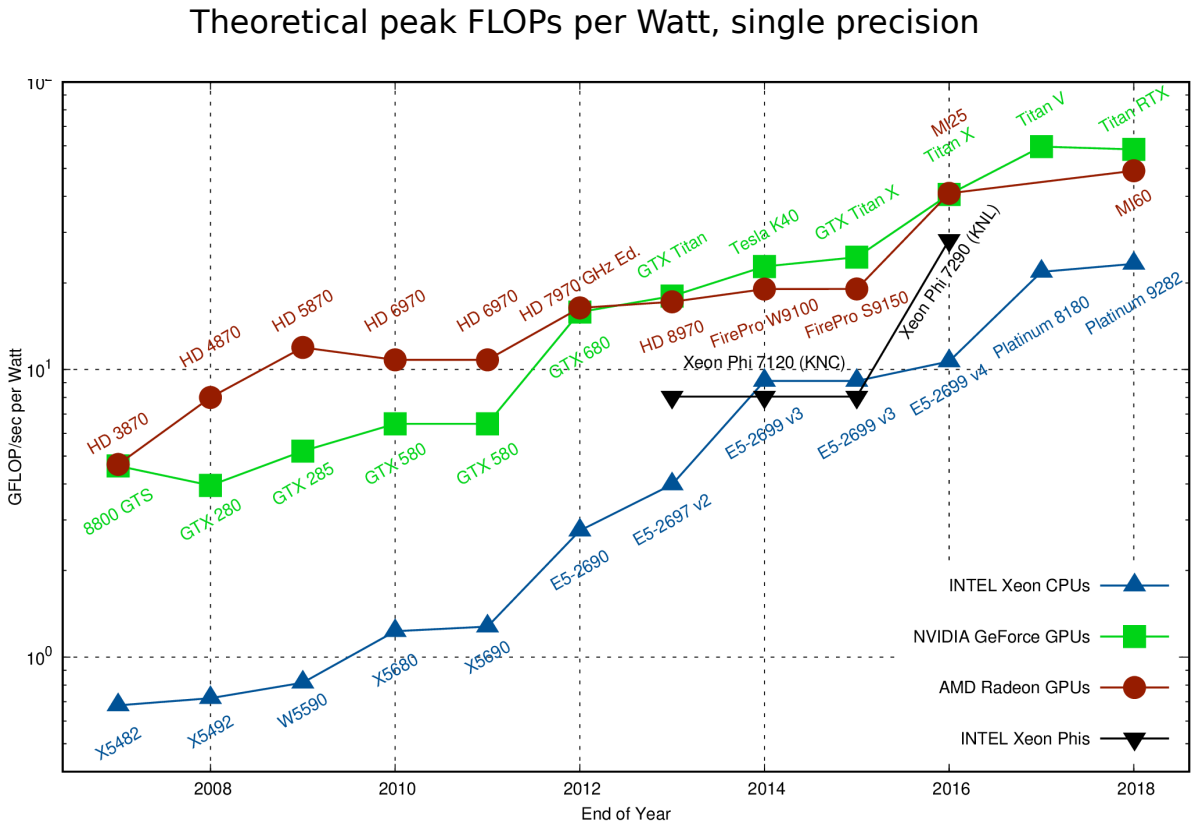
1: GTX680
2: GTX760
3: GTX970
4: GTX980Ti
5: GTX1060
6: GTX1080Ti
7: RTX2080Ti

I: K20
II: K40
III: K80
IV: M60
V: P100-16GB
VI: V100-32GB
VII: RTX6000
VIII: T4

aa: E5-2690
bb: E5-2697-v2
cc: E5-2630-v3
dd: E5-2630-v4
ee: Platinum-8180
ff: EPYC-7702
gg: EPYC-7452

https://arxiv.org/pdf/2003.11491.pdf

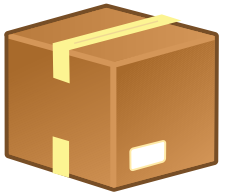
GPU power efficiency



Why the GPU computing trend?

Best theoretical FLOPs/$

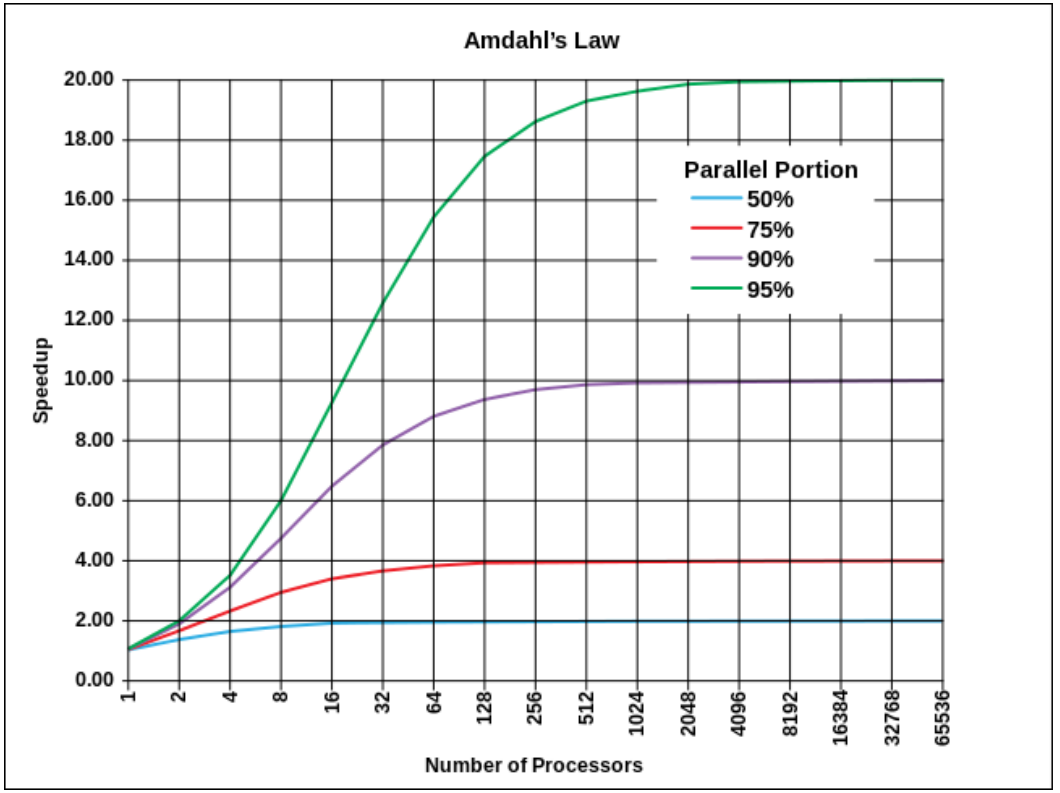
Power efficient



Many FLOPs in one device
→ compact system possible

Parallel programming

Amdahl’s law



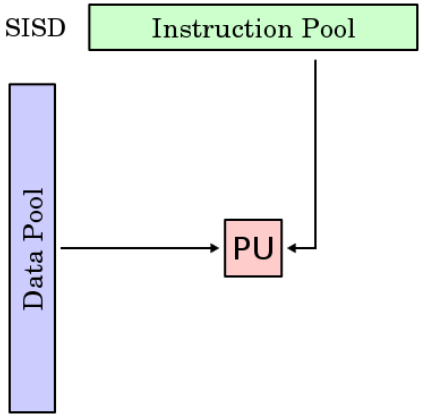
Speedup in latency = 1 / (S + P/N)

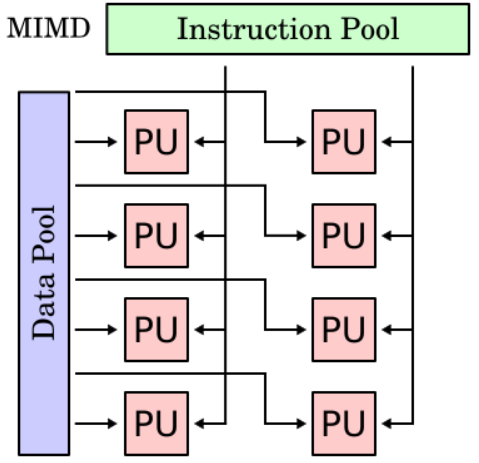
- S: sequential part of program
- P: parallel part of program
- N: number of processors

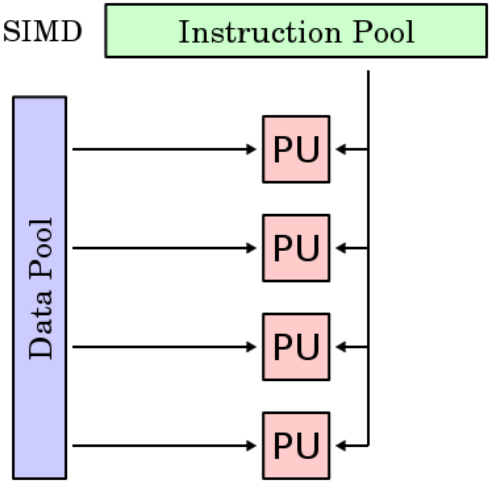
- Parallel part: identical, but independent work
- Consider how much of the problem can actually be parallelized!

SISD vs. SIMD

SISD	MIMD	SIMD
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Data
Uniprocessor machines	Multi-core, grid-, cloud-computing	e.g. vector processors

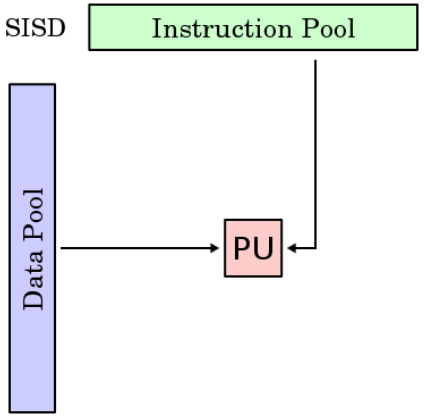


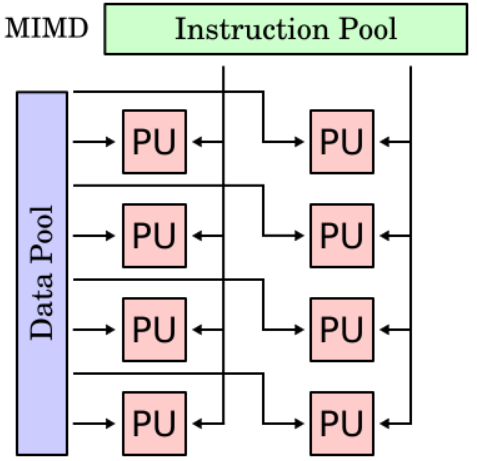


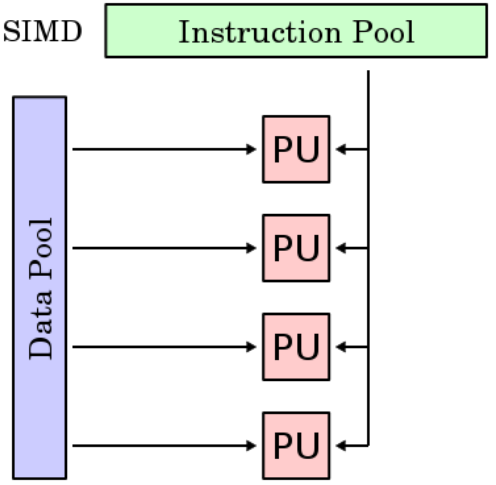


SISD vs. SIMD

SISD	MIMD	SIMT
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Threads
Uniprocessor machines	Multi-core, grid-, cloud-computing	GPUs







Single Instruction Multiple Threads

SIMT

- Similar to programming a vector processor
- Use threads instead of vectors
- No need to read data into vector register
- Only one instruction decoder available

  → all threads have to execute the same instruction

- Abstraction of vectorization:
  - Each element of vector is processed by an independent thread
  - One instruction fills entire vector
  - # of threads = vector size

What is a GPU?

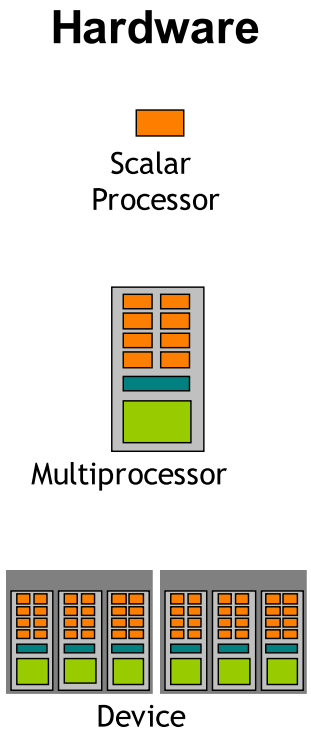
What GPUs are originally designed for

- Graphics pipeline: huge amount of arithmetic on independent data:
  - Transforming positions
  - Generating pixel colors
  - Applying material properties and light situation to every pixel

Hardware needs

- Access memory simultaneously and contiguously
- Bandwidth more important than latency
- Floating point and fixed-function logic

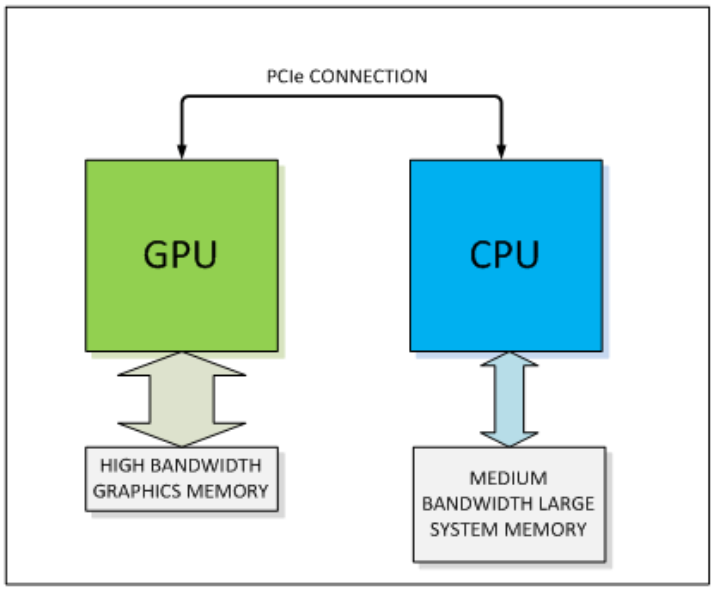
What is a GPU?

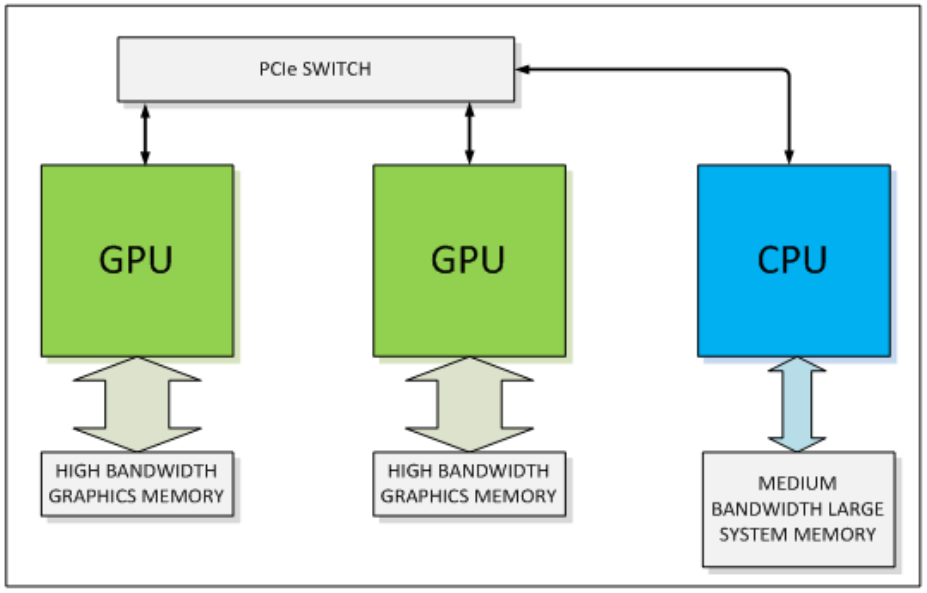


(CUDA terminology)

- Several processors are grouped into a “multiprocessor”
- Several multiprocessors make up a GPU

A GPU’s natural habitat





PCIe generation	1 lane	16 lanes	Year
3.0	985 MB/s	15.75 GB/s	2010
4.0	1.97 GB/s	31.5 GB/s	2017

Latest generation Nvidia & AMD GPUs have PCIe 4.0

Nvidia Ampere architecture

PCIe interface

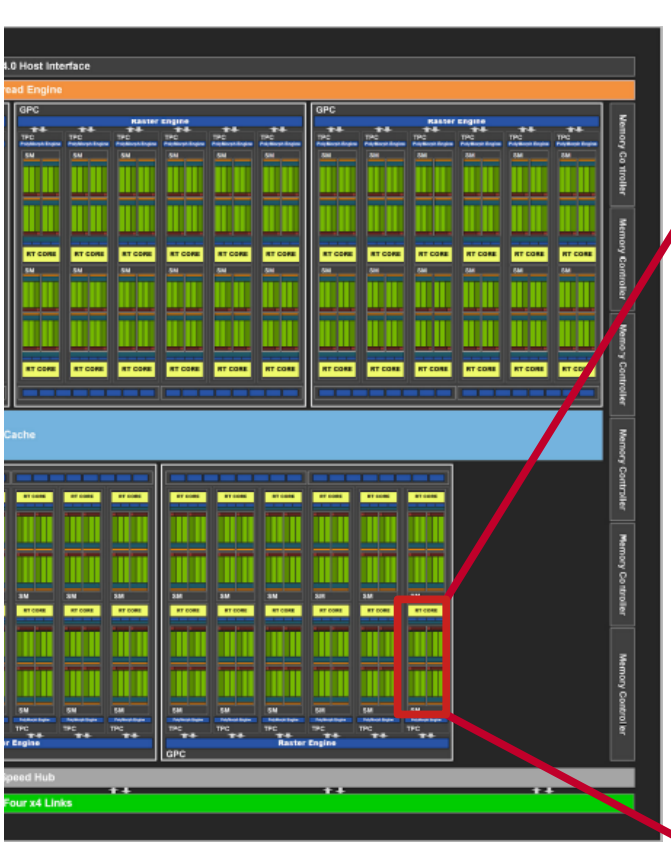
Memory controller

Graphics Processing Cluster

Streaming Multiprocessor

NVLink interface

Nvidia Ampere: Streaming Multiprocessor



Scheduler

FP32 cores

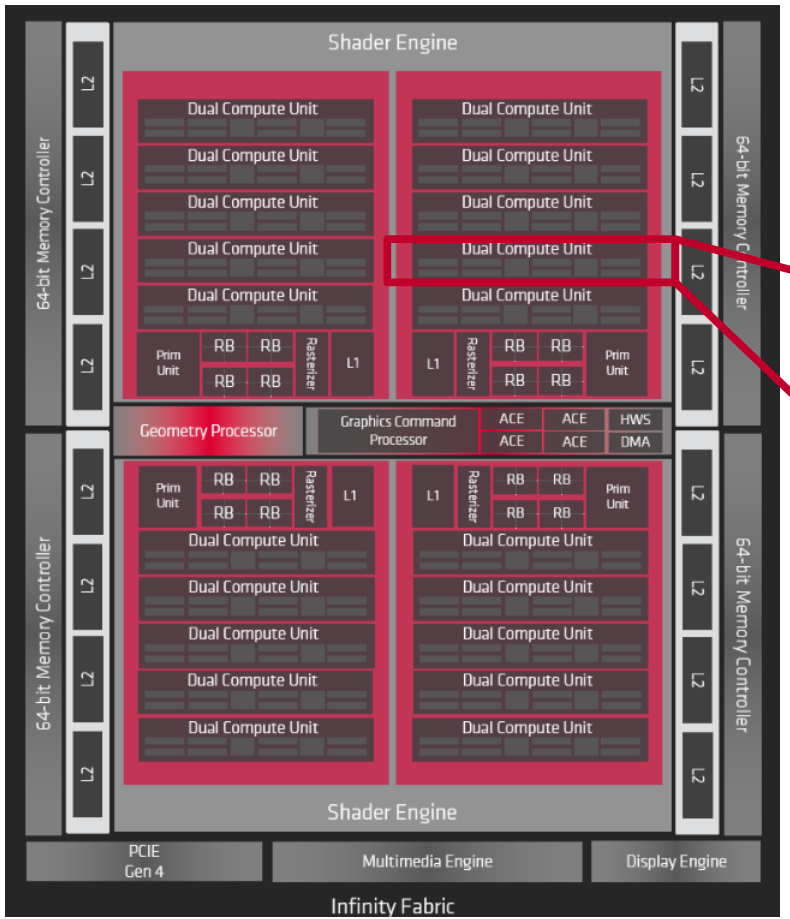
FP32/INT32 cores: can process either INT or FP

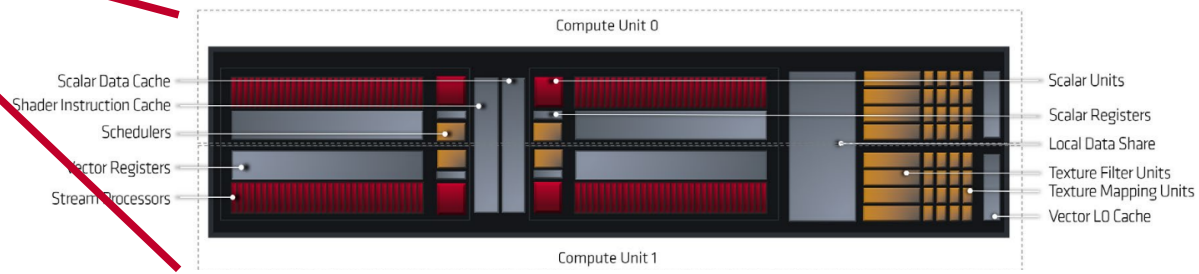
Tensor cores

Ray tracing cores (RT)

Ampere GA 102 white paper

GPU Architecture: AMD RDNA





RDNA white paper

Types of GPUs

	Scientific GPUs	**Gaming GPUs**
Precision	~3 times more single precision TFLOPS than double precision → suited for double precision	~40 times more single precision TFLOPS than double precision → not well suited for double precision
Error correction	Error correction available	Error correction **not** available
Bandwidth	NVLink & PCIe	Only PCIe
Price	~5-6 x the price of gaming GPUs	Several hundred dollars Depending on model

GPU vs. CPU

CPU: Minimize latency

- Large, low latency cache
- High frequencies
- Speculative executions

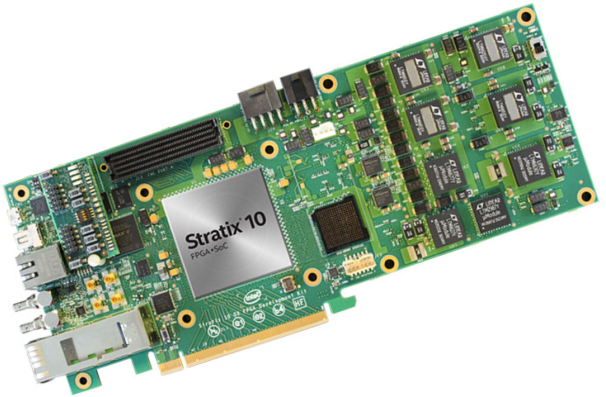
Optimal serial performance

GPU: Hide latency

- Small cache with higher latency
- Lower frequencies
- No speculative executions
- Thousands of threads
  → always have work to do

Optimal parallel performance

GPU vs. FPGA



- Higher latency
- Connection via PCIe (or NVLink)
- Bandwidth limited by PCIe
- Very good floating point operation performance
- Lower engineering cost
- Backward compatibility

- Low & deterministic latency
- Connectivity to any data source
- High bandwidth
- Intermediate floating point performance
- High engineering cost
- Not so easy backward compatibility

CPU – GPU - FPGA

	CPU	GPU	FPGA
Latency	$\mathcal{O}$ (10) µs	$\mathcal{O}$ (100) µs	Deterministic, $\mathcal{O}$ (100) ns
I/O with processor	Ethernet, USB, PCIe	PCIe, Nvlink	Connectivity to any data source via printed circuit board (PCB)
Engineering cost	Low entry level (programmable with c++, python, etc.)	Low entry level (programmable with CUDA, OpenCL, etc.)	Some high-level syntax available, traditionally VHDL, Verilog (specialized engineer)
Single precision floating point performance	$\mathcal{O}$ (10) TFLOPs	$\mathcal{O}$ (10) TFLOPs	Optimized for fixed point performance
Serial / parallel	Optimized for serial performance, increasingly using vector processing	Optimized for parallel performance	Optimized for parallel performance
Memory	$\mathcal{O}$ (100) GB RAM	$\mathcal{O}$ (10) GB	$\mathcal{O}$ (10) MB (on the FPGA itself, not the PCB)
Backward compatibility	Compatible, except for vector instruction sets	Compatible, except for specific features only available on modern GPUs	Not easily backward compatible

https://arxiv.org/pdf/2003.11491.pdf

Programming GPUs

GPU Programming Environments

Early days: Problems had to be translated to graphics language via OpenGL
Today: several programming interfaces exist

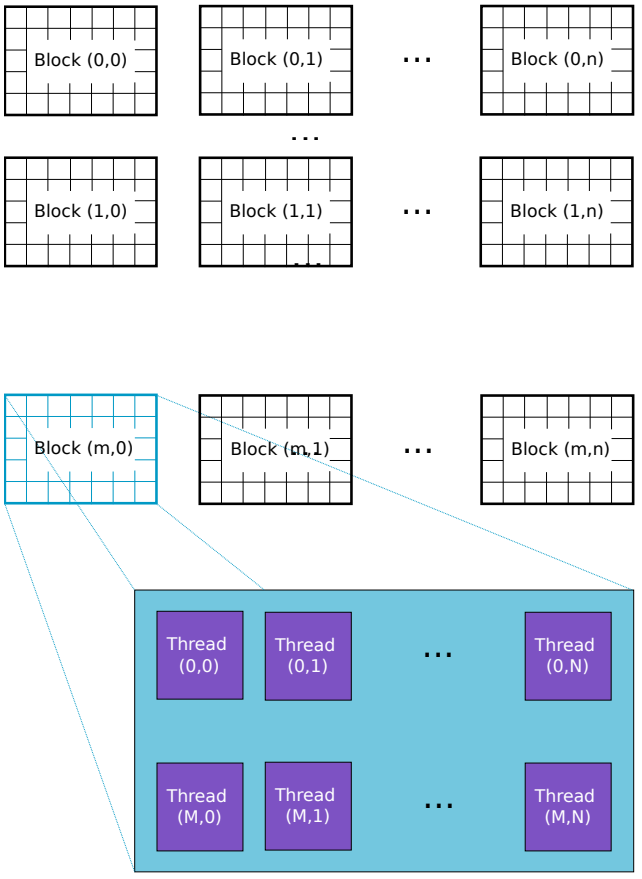
- Nvidia's application programming interface: CUDA
  - Only works with Nvidia GPUs
  - Very well documented, many tutorials, low entry level
- AMD ROCm (HIP): Open source platform for GPU computing
  - Supports both AMD and Nvidia GPUs
  - New development → still work in progress, not that many examples / tutorials yet
- OpenCL: Framework for heterogeneous platforms
  - CPUs, GPUs, FPGAs, DSPs, etc.
  - Maintained by the Khronos group, based on C99 and C++11
- SYCL: Single source C++ heterogeneous programming platform, built on OpenCL
  - Will be supported by Intel GPUs

CUDA

- CUDA is widely used in the GPU computing community
- Underlying concepts easily translate to the other programming interfaces

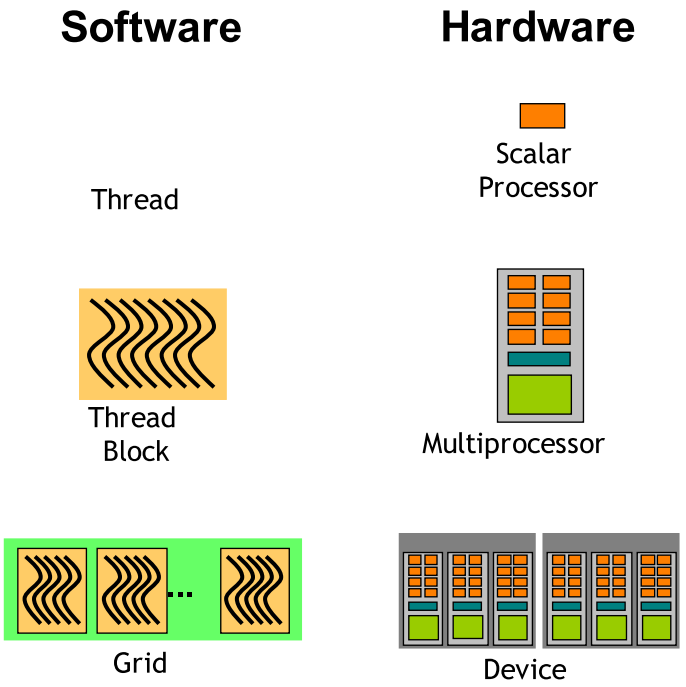
- Compiles with nvcc
- Very similar to C/C++ code

Parallelization



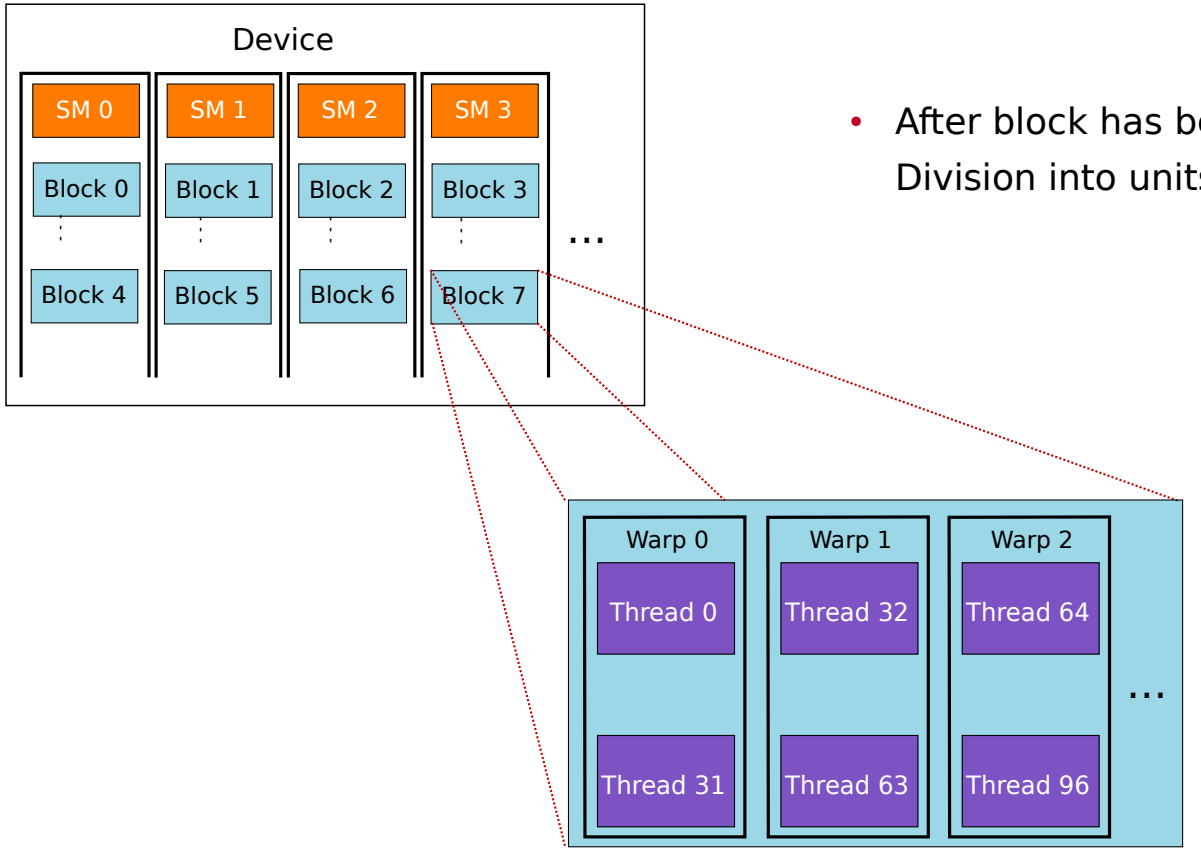
- Any GPU code we write will be executed on many “threads” at once
- These threads are organized in a “grid”, where a fixed set of threads is grouped into one “block”
- Each thread processes the same instructions (kernel), but on different data
- Up to three dimensions for blocks and threads
- Maximum of 1024 threads / block

Hardware implementation



- Execution order of blocks is arbitrary
- Scheduled on Streaming Multiprocessors (SMs) according to resource usage:
  - Memory
  - Registers
  - Thread number limit (2024 threads / block)
- Several blocks can run on the same SM

Hardware implementation



- After block has been assigned to one SM:
  Division into units called warps = 32 threads

Function declarations

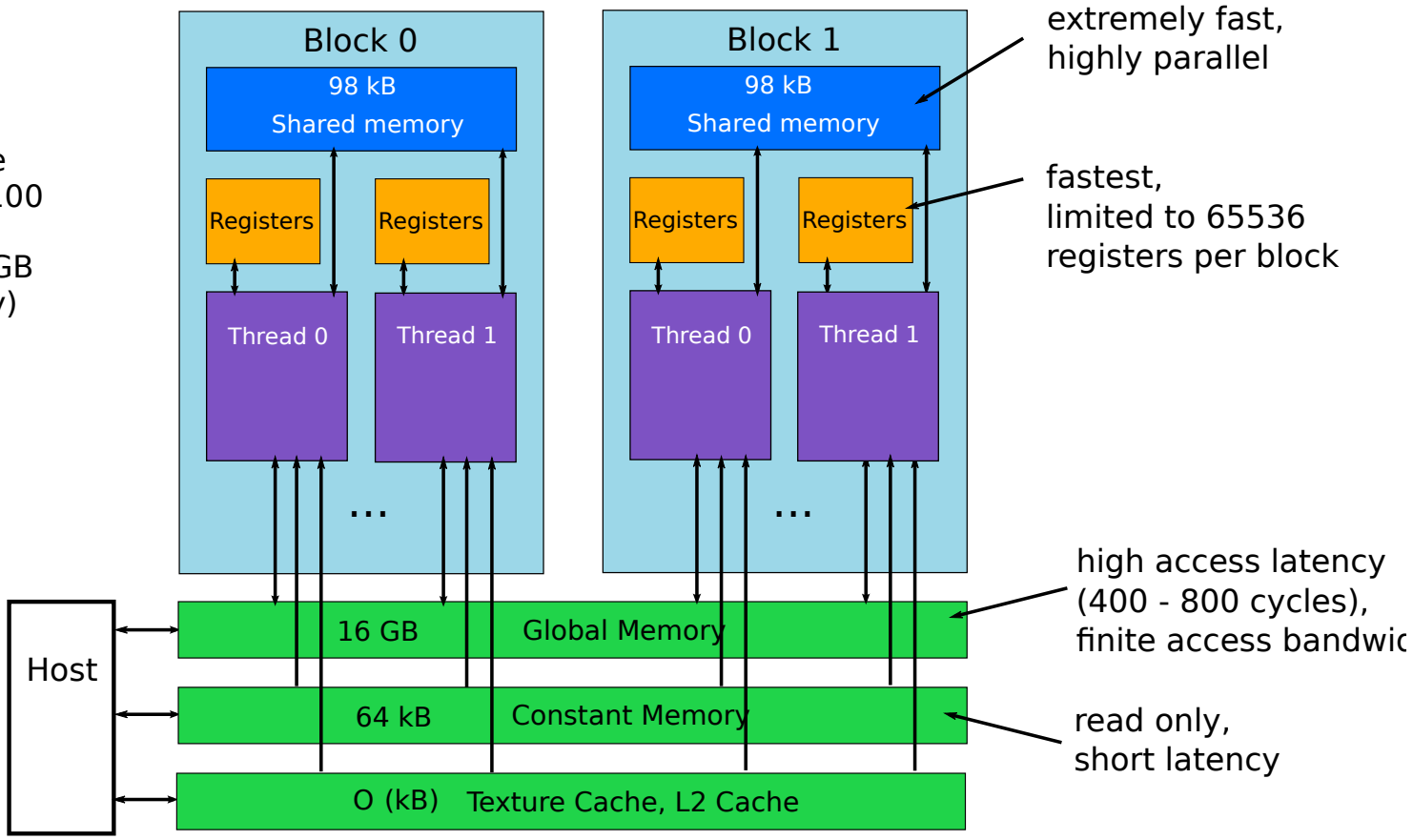
	Called from	**Executed on**	**Comment**
__global__	Host	Device	Defines kernel, returns void
__device__	Device	Device	
__host__	Host	Host	Optional

__device__ and __host__ can be combined, useful if same function is executed on host OR device

Memory layout

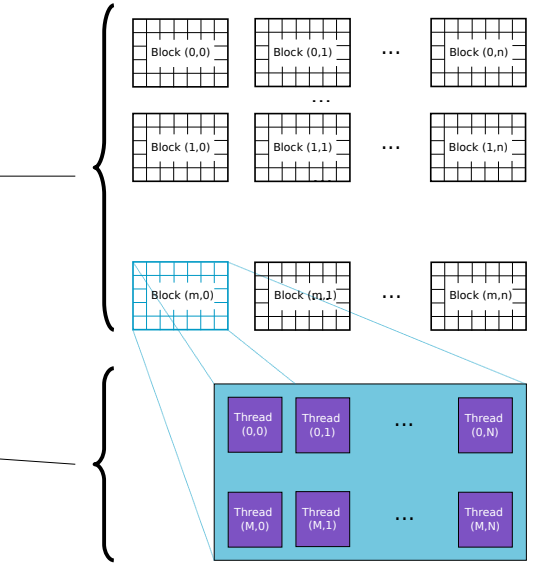
Specs from the
16 GB Tesla V100

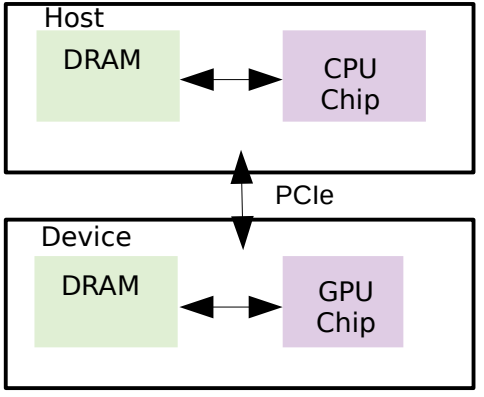
(A100 has 40 GB
global memory)



Communication & Synchronization

- Within the blocks of a grid:
  - Synchronization of all thread in a grid only possible after kernel has finished
  - Communication via global memory
- Within on block:
  - Synchronization of threads is possible within one block: `__syncthreads()`
  - Communication via shared & global memory
- Between the host and the device:
  - Copies between global memory and CPU RAM



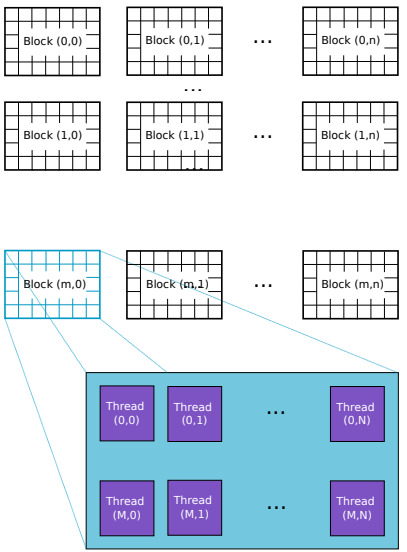


Race conditions

- Caution when modifying the same value in memory from different threads:
  - Need to read, modify, write value: three operations
  - Outcome depends on timing of the different threads
  - Thread 1 can modify after thread 2 read a value, but before thread 2 writes a new value!
- Use atomic operations:
  - Read-modify-write cannot be interrupted: appears to be one operation
  - atomicAdd(), atomicSub(), atomicInc(), atomicDec(), …
- Needed for both shared and global memory

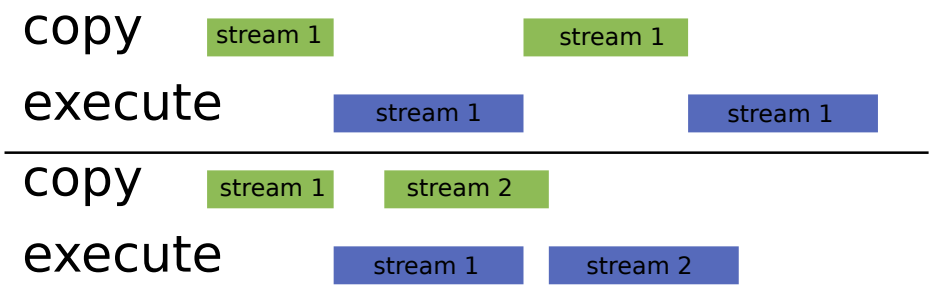
Grid sizes

- Block size:
  - Multiple of the warp size (32 threads)
  - Consider registers used within one kernel: # registers / block is limited
- Grid size: ideally multiple of the number of streaming multiprocessors
- Most efficient grid dimensions can vary with the GPU device



Asynchronous execution

- Independent tasks can be executed concurrently:
  - Computation on host
  - Computation on device
  - Memory transfers
- “Async” function calls → pipeline of tasks only synchronized within one stream
- `cudaDeviceSynchronize()`: waits for all streams to finish



How to call a CUDA function

```
__global__ void hello_world( ) {
  int BlockIdx = blockIdx.x;
  int ThreadIdx = threadIdx.x;

  // Do Stuff with BlockIdx and ThreadIdx

}
int main( ) {
dim3 n_blocks(1);
dim3 n_threads(1);
hello_wold<<<n_blocks, n_threads>>>();
return 0;
}
```

Indicates that function runs on device, is called from host; compiled with nvcc

Built-in variable to access index of current block

Built-in variable to access index of current thread within block

Structure designed to store size of grid and block

<<< >>>: options for grid launch:
# of blocks,
# of threads / block
(): arguments can be passed to kernel function

How to allocate and free memory

```
int a_host = 8;
int *a_dev;

cudaMalloc( (void**)&a_dev, sizeof(int) );

cudaMemcpy( a_dev, &a_host, sizeof(int), cudaMemcpyHostToDevice );

DoStuff<<<1,1>>>( a_dev);

cudaDeviceSynchronize();

cudaFree( a_dev);
```

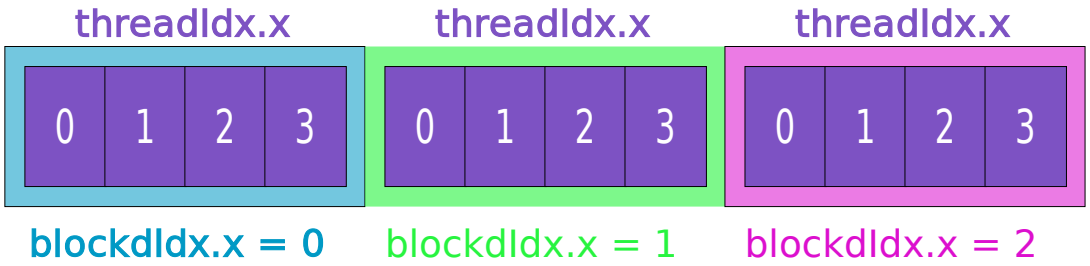
Pointer to allocated memory on device is returned

Size of memory to be allocated

Pointer to source

Pointer to destination

Index calculation



- Unique index = x + y * size
- int index = threadIdx.x + blockIdx.x * blockDim.x;
- blockDim.x: number of threads in a block (in x direction), accessible from the kernel
- gridDim.x: number of blocks in the grid (in x direction), accessible from the kernel

Cuda tools: nvidia-smi

Nvidia-smi is available with every CUDA installation

```
(base) [dvombruc@lpnlhcbgpu:cuda-introduction-solution]# nvidia-smi
Sun May 10 18:42:52 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  Off  | 00000000:D8:00.0 Off |                  Off |
| N/A   28C    P0    34W / 250W |      0MiB / 16160MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Cuda tools: DeviceQuery

```
(base) [dvombruc@lpnlhcbgpu:deviceQuery]# pwd
/home/dvombruc/cuda-samples/NVIDIA_CUDA-10.1_Samples/1_Utilities/deviceQuery
(base) [dvombruc@lpnlhcbgpu:deviceQuery]# ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-PCIE-16GB"
  CUDA Driver Version / Runtime Version          10.2 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 16160 MBytes (16945512448 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1380 MHz (1.38 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 7 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 216 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
```

Summary

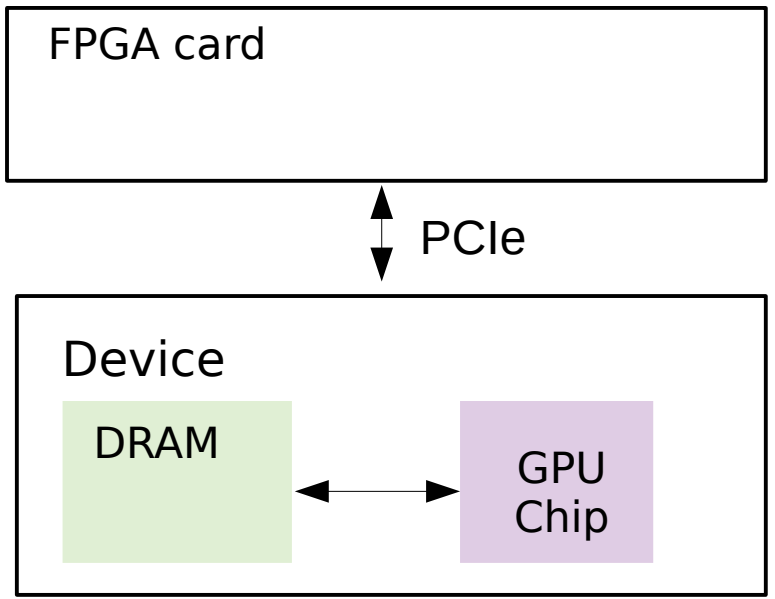
- GPUs are well suited for inherently parallel problems:
  run the same instructions on independent data
- Offer most theoretical TFLOPs/$
- Power efficient
- Several programming environments available
- CUDA is well documented, tested and widely used in the community
- CUDA concepts easily translate to other programming environments

Backup

Talk to a GPU: NVLink, GPUDirect

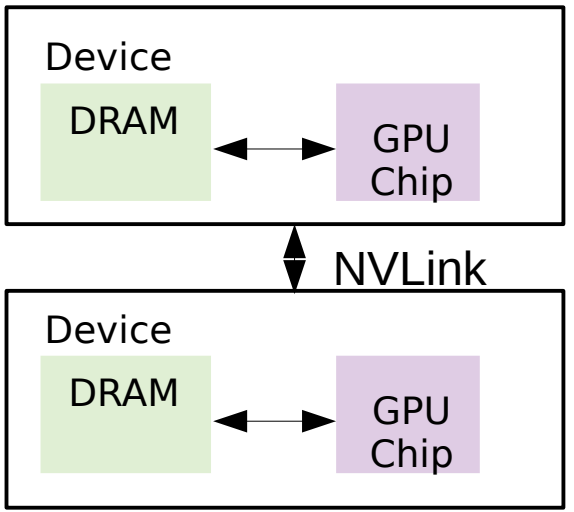
GPUDirect:

- Direct memory access (DMA) transfer directly over PCIe switch
- Only available for scientific Nvidia GPUs



NVLink:

- Communications protocol developed by Nvidia
- Can be used between among multiple GPUs
- 160 / 300 / 600 GB/s data rate (1st / 2nd / 3rd generation)



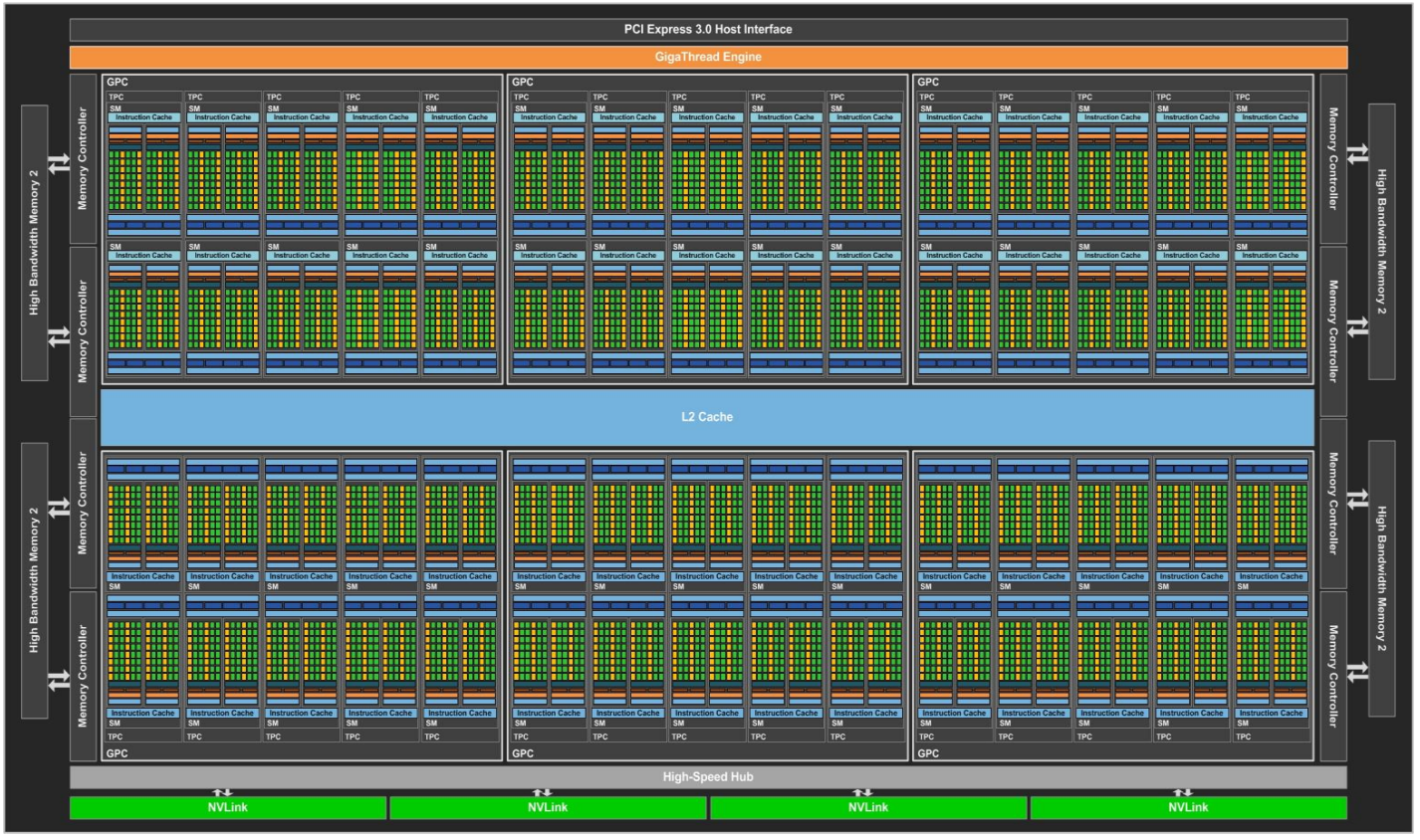
Some Nvidia GPUs

Feature	GeForce GTX 2080 Ti	GeForce GTX 3080	Tesla V100	Tesla A100
# cores	4352	8704	5120	6912
Max. frequency	1.35 GHz	1.44	1.37 GHz	1.44 GHz
Cache (L2)	6 MB	5 MB	6 MB	40 MB
DRAM	11 GB GDDR6	10 GB GDDR6X	32 GB HBM2	40 GB HBM2
Max TFLOPs	13.4	30	15.7	19.5
TDP	250 W	320 W	250 W	250 W

Gaming GPUs

Scientific GPUs

GPU Architecture: Nvidia Pascal



Nvidia Tesla P100 white paper

Nvidia: Pascal Streaming Multiprocessor

Scheduler

Dispatch units

64 single precision cores (FP32)

32 double precision cores (FP64)

Load / store units

Special function units

Nvidia Tesla P100 white paper