Parallel Computing with CUDA

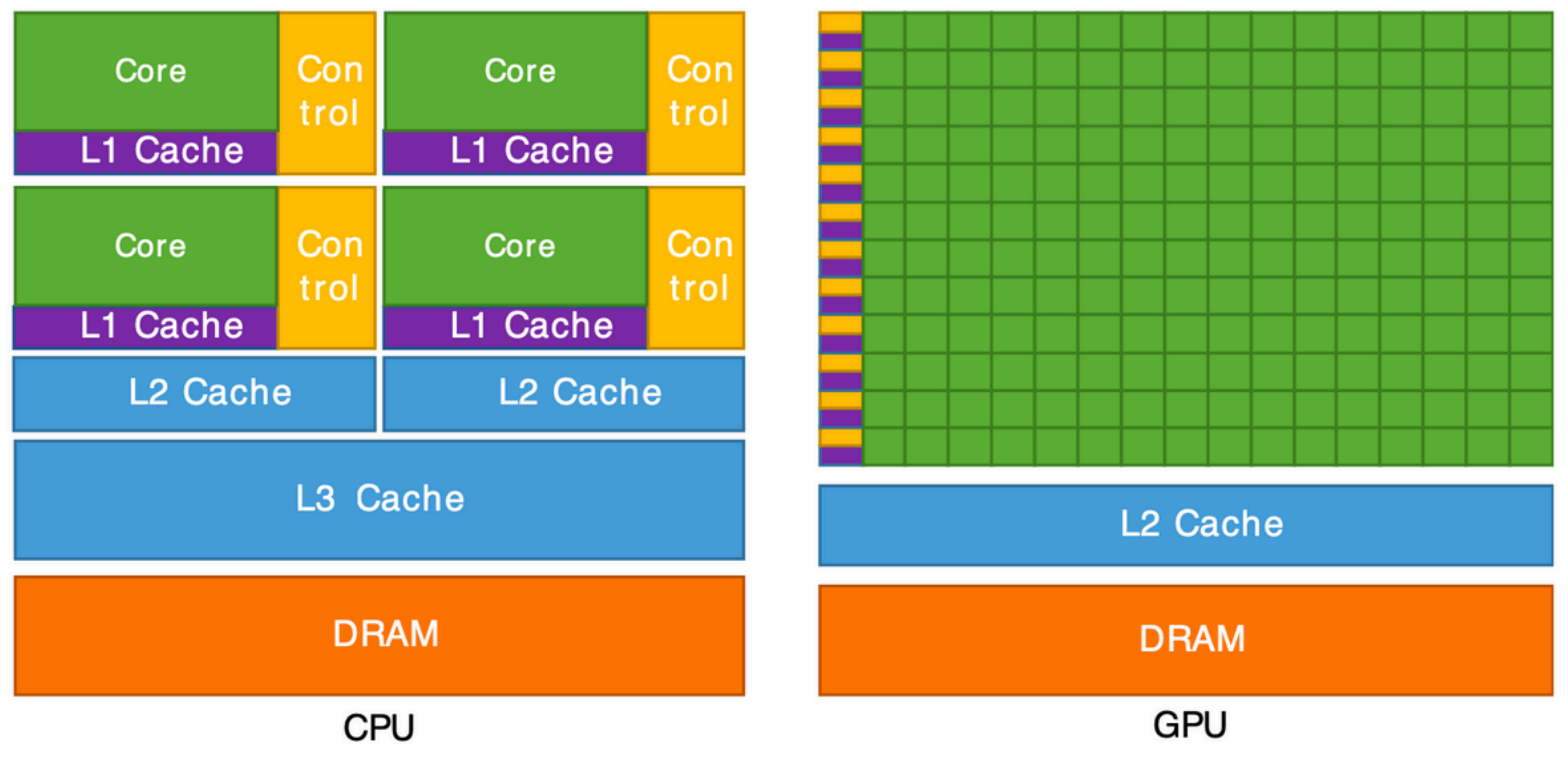
E317/517 HIGH-PERFORMANCE COMPUTING

Spring 2024

GPU Design

- Originally used to render graphics for shading, texturing and independent polygons for 3D objects
- CPUs control and execute the logic for general-purpose computing
- Thus, GPUs thus have many more processing units and higher memory bandwidth while CPUs have better ALU for instruction processing and faster clock speeds

  - CPUs can handle more complex workloads
  - GPUs have more ALUs/FLUs but less capable
  - CPUs have more cache memory
  - **GPUs are designed for parallelizable workloads**



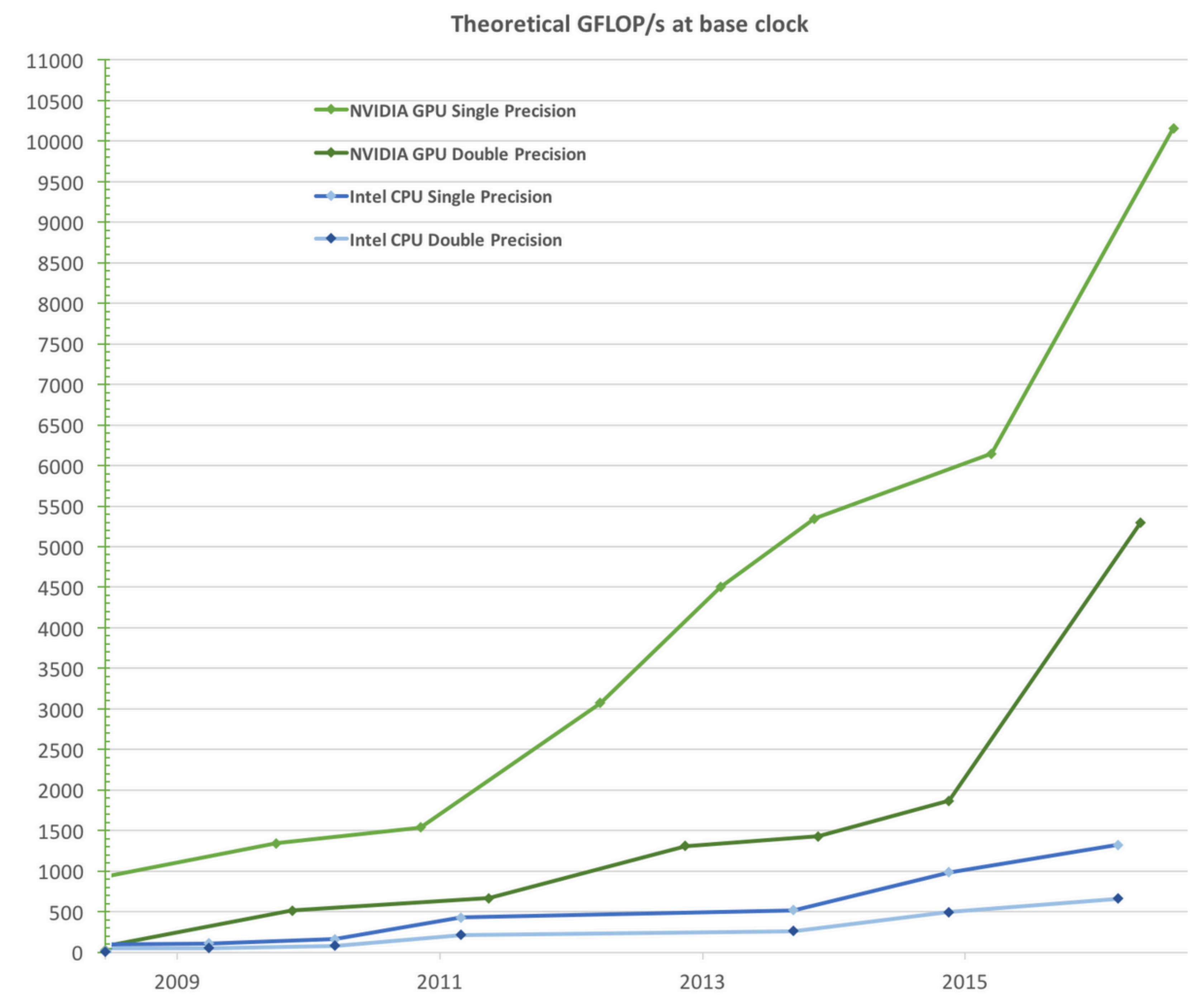
Comparing the relative capabilities of the basic elements of CPU and GPU architectures.

Applications for GPU Computing

- Floating-point heavy operations and simple data access patterns can speedup from GPUs (GPGPUs)
- **CUDA** = **C**ompute **U**nified **D**evice **A**rchitecture is a programming model for general purpose (GP) computing on NVIDIA GPUs that comes as an extension to C/C++ and Fortran
- CUDA API accelerates numerically intensive programs like matrix multiplication, FFTs, decryptions etc.
- Scientific and engineering fields leveraging CUDA and GPUs: computational fluid dynamics, bioinformatics, molecular dynamics, computational physics, quantum chemistry, medical imaging, data science, finance, climate/weather modeling and **deep learning**
- More examples can be found here: https://www.nvidia.com/en-us/accelerated-applications/
- Any downsides?

Heterogeneous Computing

- CPUs are designed for multi-tasking and fast serial processing, while GPUs are designed for high throughput parallel tasks
- GPUs are hosted on CPU-based systems; offload massively parallel and numerically intensive tasks to GPUs in heterogeneous computing
- Program flow in CUDA:
  - Load data to CPU memory and copy to GPU
  - Call computation to execute on GPU device
  - Fetch the results back to the CPU



Peak performance in Gflop/s of GPUs and CPUs in single and double precision, 2009-2016.

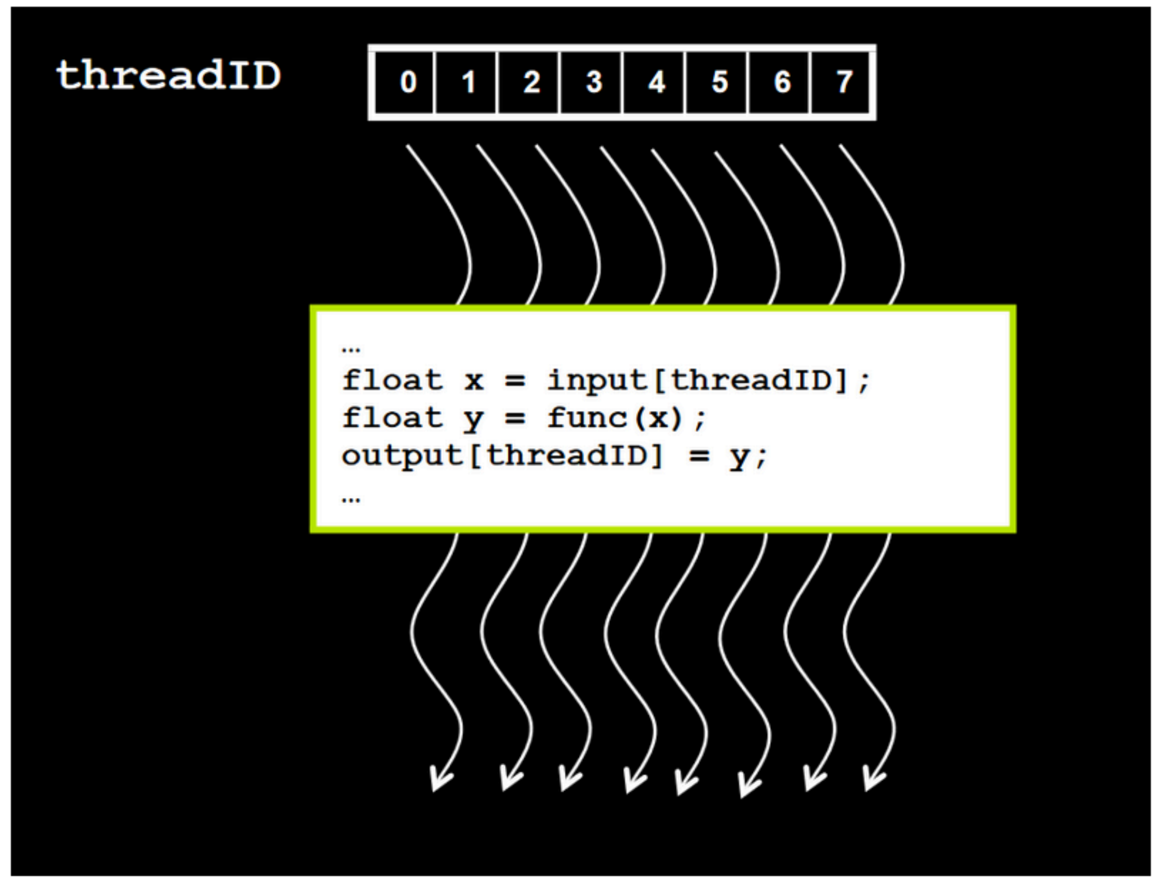
Execution Syntax in CUDA

- Parallel programming w/ CUDA needs NVIDIA specific GPU hardware and CUDA toolkit installed

- ***Kernel*** is a function to be executed on a GPU device, and can either be called from device itself or the CPU (host); *but always executes on the device!*

- Specified by the **__global__** specifier with an additional execution configuration syntax **<<<b, t>>>**

- To execute CUDA code, we need the NVIDIA CUDA compiler (nvcc) and specify the GPU architecture (or *compute capability*) we want to run parallel code on.

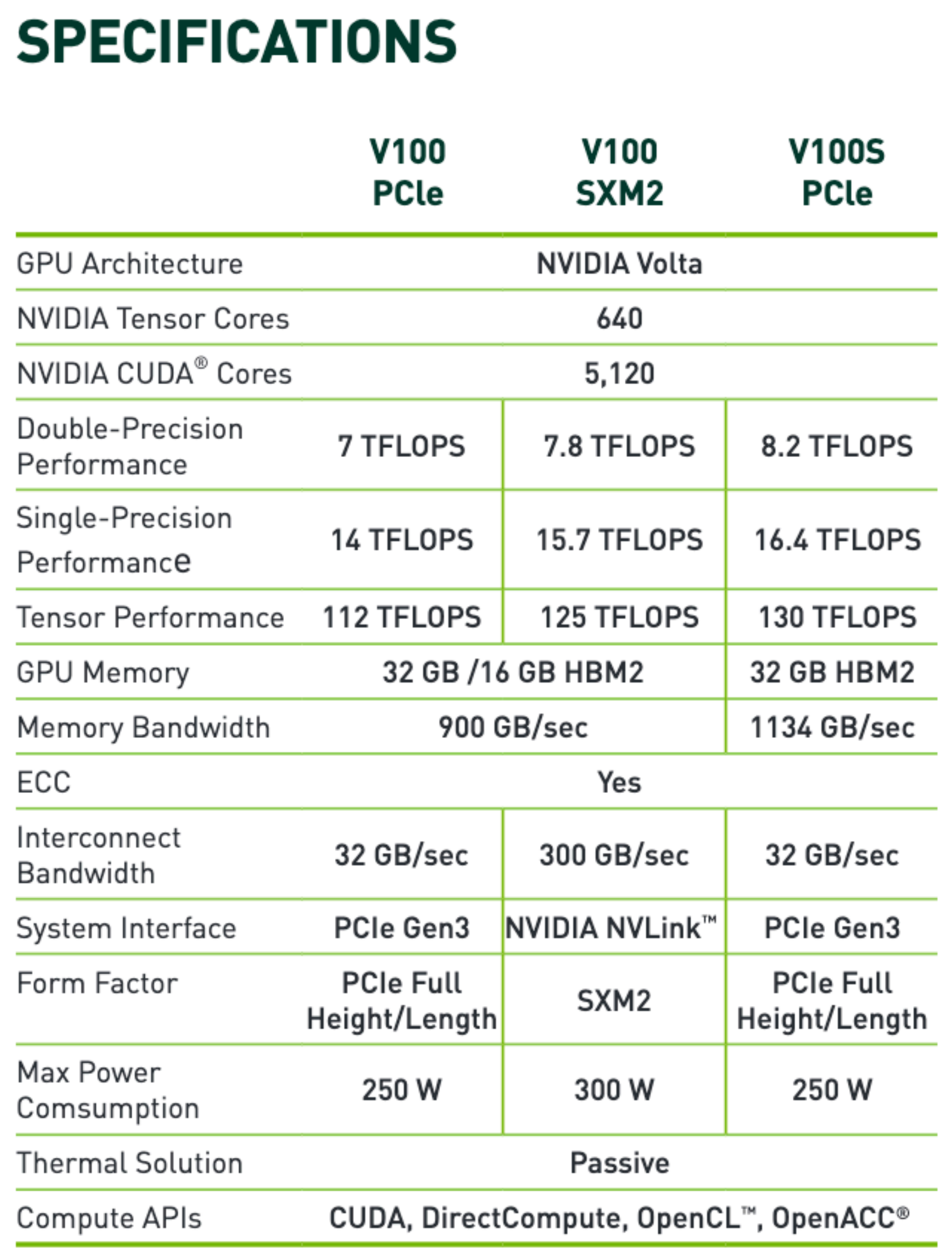


```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

Showing how a CUDA kernel is executed by an array of threads.

Compiling/Running CUDA C/C++

- Examples available here https://github.com/sahiltyagi4/graphalgoscuda
- Access a node partition with GPUs (may have limited access!) and load CUDA toolkit: *module load cuda/****version**** (IU GPU clusters have **9.0/9.1/10.0/10.1/11.0/11.2**)
- How to choose device to run CUDA code: **cudaSetDevice()**, **CUDA_VISIBLE_DEVICES** env
- How to compile CUDA C/C++: *nvcc -arch=*compute_arch* -o *target* *target.cu**
- How to debug and profile CUDA code: nvprof *./*target**
- For e.g., V100 uses has compute capability 7.0 and uses architecture ***sm_70***



SPECIFICATIONS

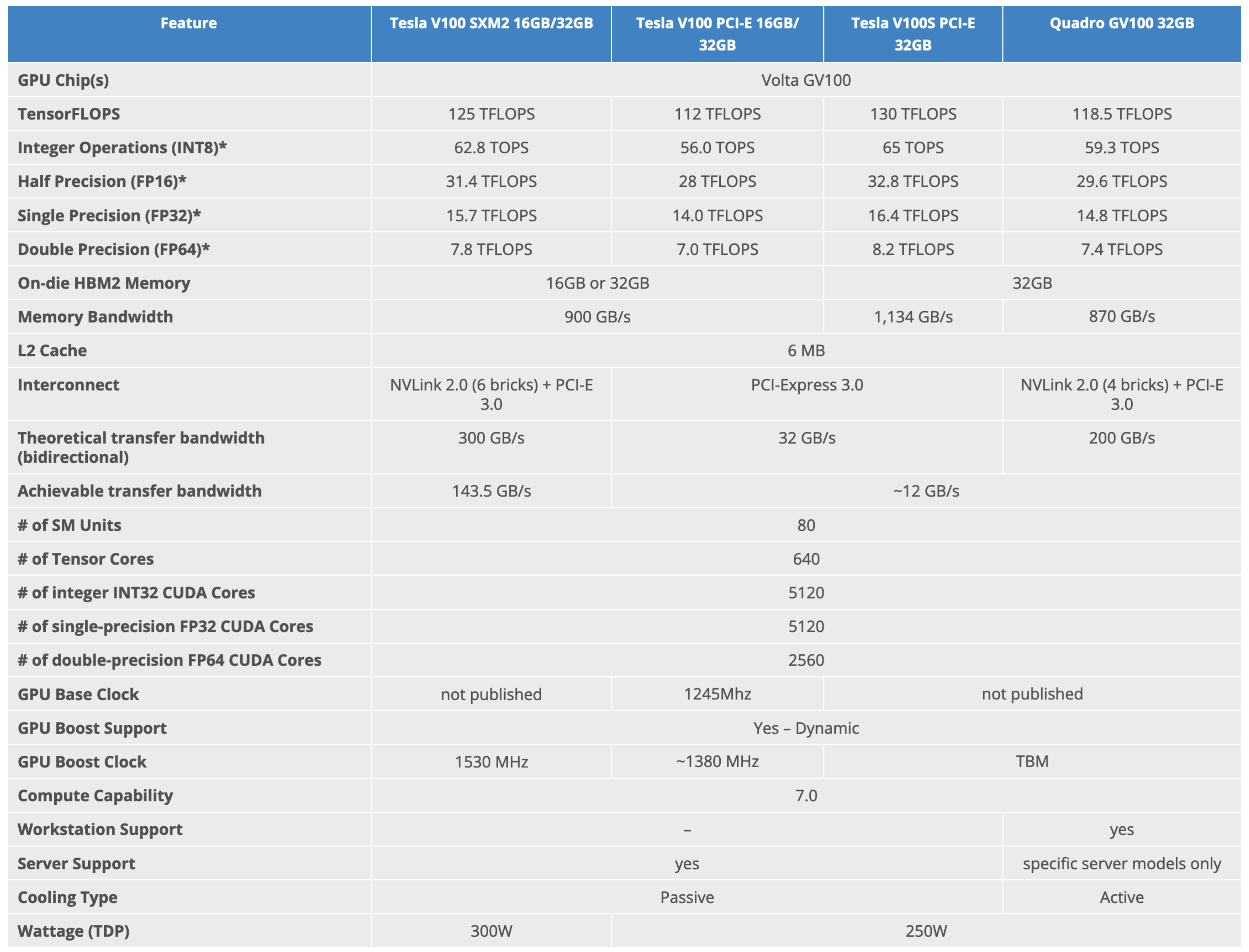
	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture	NVIDIA Volta		
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS
GPU Memory	32 GB /16 GB HBM2		32 GB HBM2
Memory Bandwidth	900 GB/sec		1134 GB/sec
ECC	Yes		
Interconnect Bandwidth	32 GB/sec	300 GB/sec	32 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink™	PCIe Gen3
Form Factor	PCIe Full Height/Length	SXM2	PCIe Full Height/Length
Max Power Comsumption	250 W	300 W	250 W
Thermal Solution	Passive		
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC®		

Basic CUDA Syntax and Functions

- The **__global__** specifier tells CUDA compiler what needs to be executed on the GPU device
- Memory allocations/manipulation in CUDA: **cudaMalloc()**, **cudaMemcpy()**, **cudaMallocHost()** (pinned memory), **cudaMallocManaged()** (unified memory), **cudaFree()**
- Thread synchronization done via **cudaDeviceSynchronize()**, but stalls the GPU pipeline
- Individual threads can be accessed via **threadIdx, blockIdx, blockDim, gridDim** identifiers
- **cudaEvent_t** built over CUDA streams gives an alternative to CPU timers without explicit device synchronization via functions like **cudaEventCreate()**, **cudaEventRecord()** , **cudaEventSynchronize()** and **cudaEventDestroy()**.
- **cudaError_t** provides error handling in CUDA (**cudaGetLastError()**, **cudaGetErrorString()**)

Thread execution in GPU

- What makes GPUs great for HP parallel computing?
- A **thread** is composed of instructions + data that runs on a *CUDA core*; based on SIMT architecture
- **CUDA cores** are the units that process the actual data one after another
- A **warp** is a group of 32 threads for SIMT execution; equivalent to VPUs on CPUs
- A **kernel** is a function parallelized by thread blocks and threads/block
- A **streaming multiprocessor (SM)** is a unit that executes the thread block of a kernel; equivalent to cores in CPU

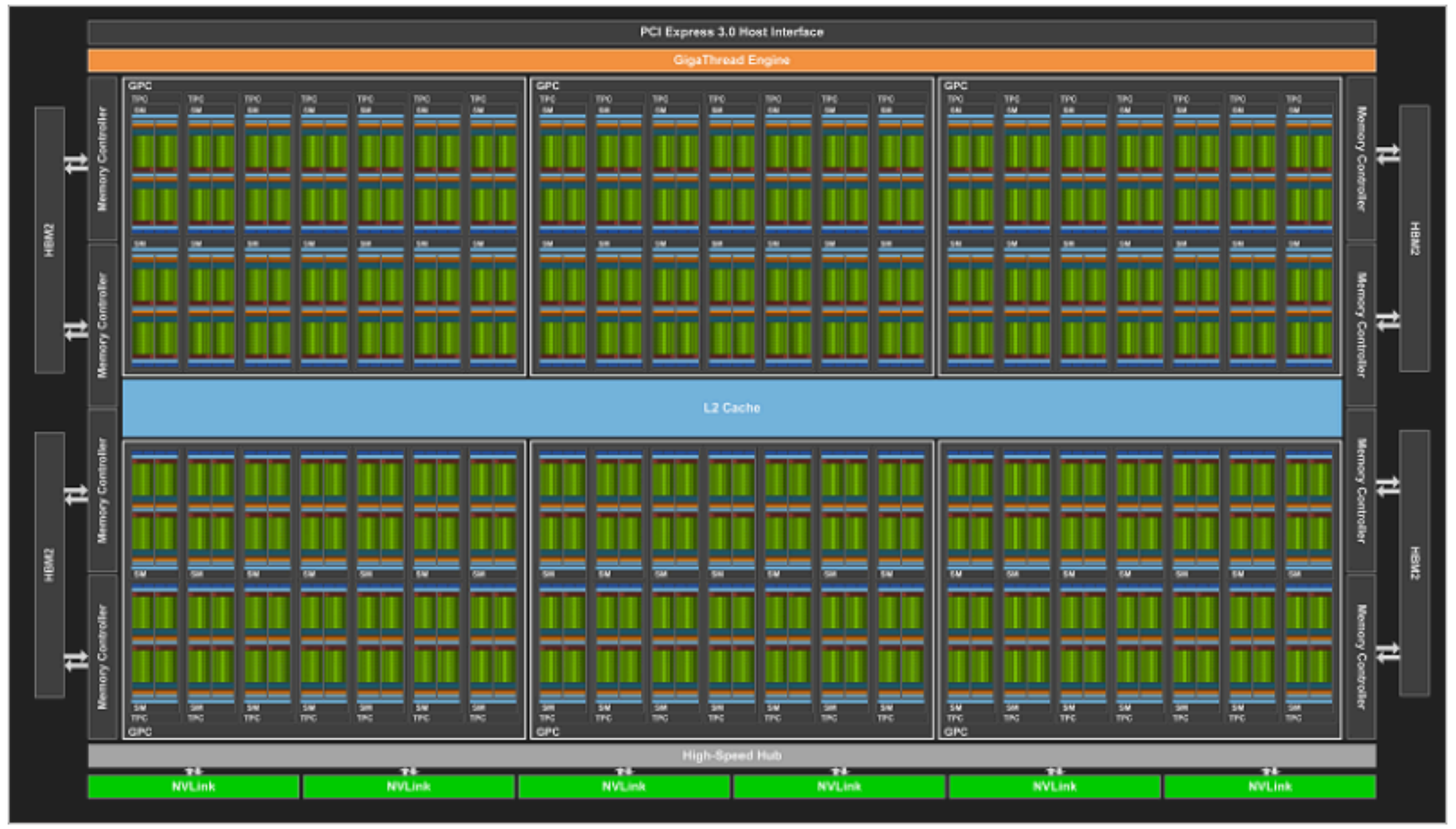


Feature	Tesla V100 SXM2 16GB/32GB	Tesla V100 PCI-E 16GB/ 32GB	Tesla V100S PCI-E 32GB	Quadro GV100 32GB
GPU Chip(s)	Volta GV100			
TensorFLOPS	125 TFLOPS	112 TFLOPS	130 TFLOPS	118.5 TFLOPS
Integer Operations (INT8)*	62.8 TOPS	56.0 TOPS	65 TOPS	59.3 TOPS
Half Precision (FP16)*	31.4 TFLOPS	28 TFLOPS	32.8 TFLOPS	29.6 TFLOPS
Single Precision (FP32)*	15.7 TFLOPS	14.0 TFLOPS	16.4 TFLOPS	14.8 TFLOPS
Double Precision (FP64)*	7.8 TFLOPS	7.0 TFLOPS	8.2 TFLOPS	7.4 TFLOPS
On-die HBM2 Memory	16GB or 32GB		32GB	
Memory Bandwidth	900 GB/s		1,134 GB/s	870 GB/s
L2 Cache	6 MB			
Interconnect	NVLink 2.0 (6 bricks) + PCI-E 3.0	PCI-Express 3.0		NVLink 2.0 (4 bricks) + PCI-E 3.0
Theoretical transfer bandwidth (bidirectional)	300 GB/s	32 GB/s		200 GB/s
Achievable transfer bandwidth	143.5 GB/s	~12 GB/s		
# of SM Units	80			
# of Tensor Cores	640			
# of integer INT32 CUDA Cores	5120			
# of single-precision FP32 CUDA Cores	5120			
# of double-precision FP64 CUDA Cores	2560			
GPU Base Clock	not published	1245Mhz	not published	
GPU Boost Support	Yes – Dynamic			
GPU Boost Clock	1530 MHz	~1380 MHz	TBM	
Compute Capability	7.0			
Workstation Support	–			yes
Server Support	yes			specific server models only
Cooling Type	Passive			Active
Wattage (TDP)	300W	250W		

Example: ***devicequery.cu***

SIMT execution on GPUs

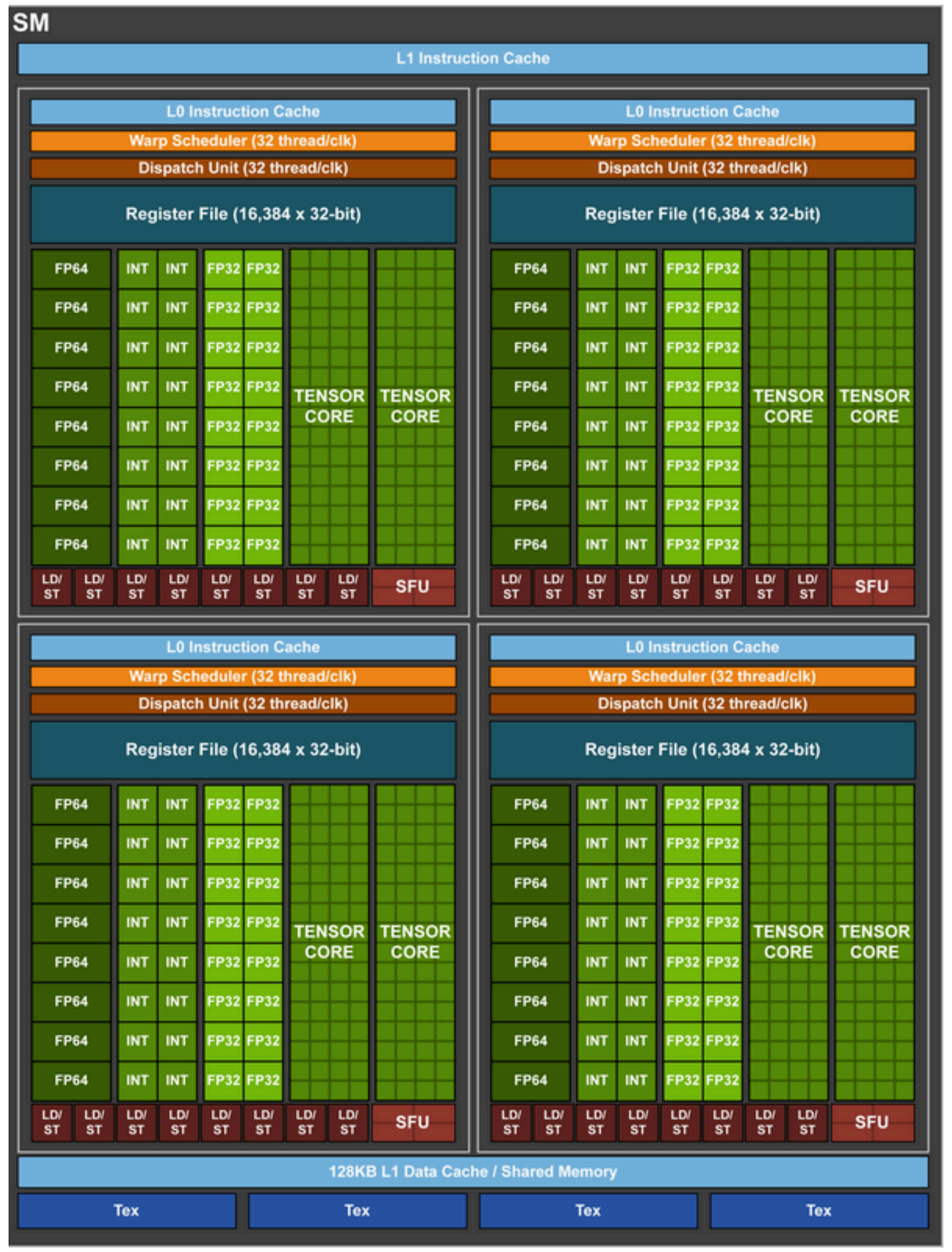
- Closely related to SIMD execution
- A single instruction acts on all the data in exactly the same way in SIMD
- SIMT loosens this restriction by executing instructions only on the active threads; accommodates branching
- At runtime, a block of threads is divided into warps for SIMT execution; each contains 32 threads with consecutive indexes and processed by a set of 32 CUDA cores
- Analogous to vectorized processing unit in CPU where vectors are chunked into fixed size and processed by vector lanes



Volta GV100 block diagram.

SM of NVIDIA Tesla V100

- Divided into 4 blocks which allows for flexible scheduling (upto 2 FP32 or INT32 or 1 FP64/cycle)
- Each SM contains these datatypes CUDA cores:
  - 64 FP32
  - 64 INT32
  - 32 FP64
- 8 Tensor cores
- 16 Special function units
- 4 Texture units

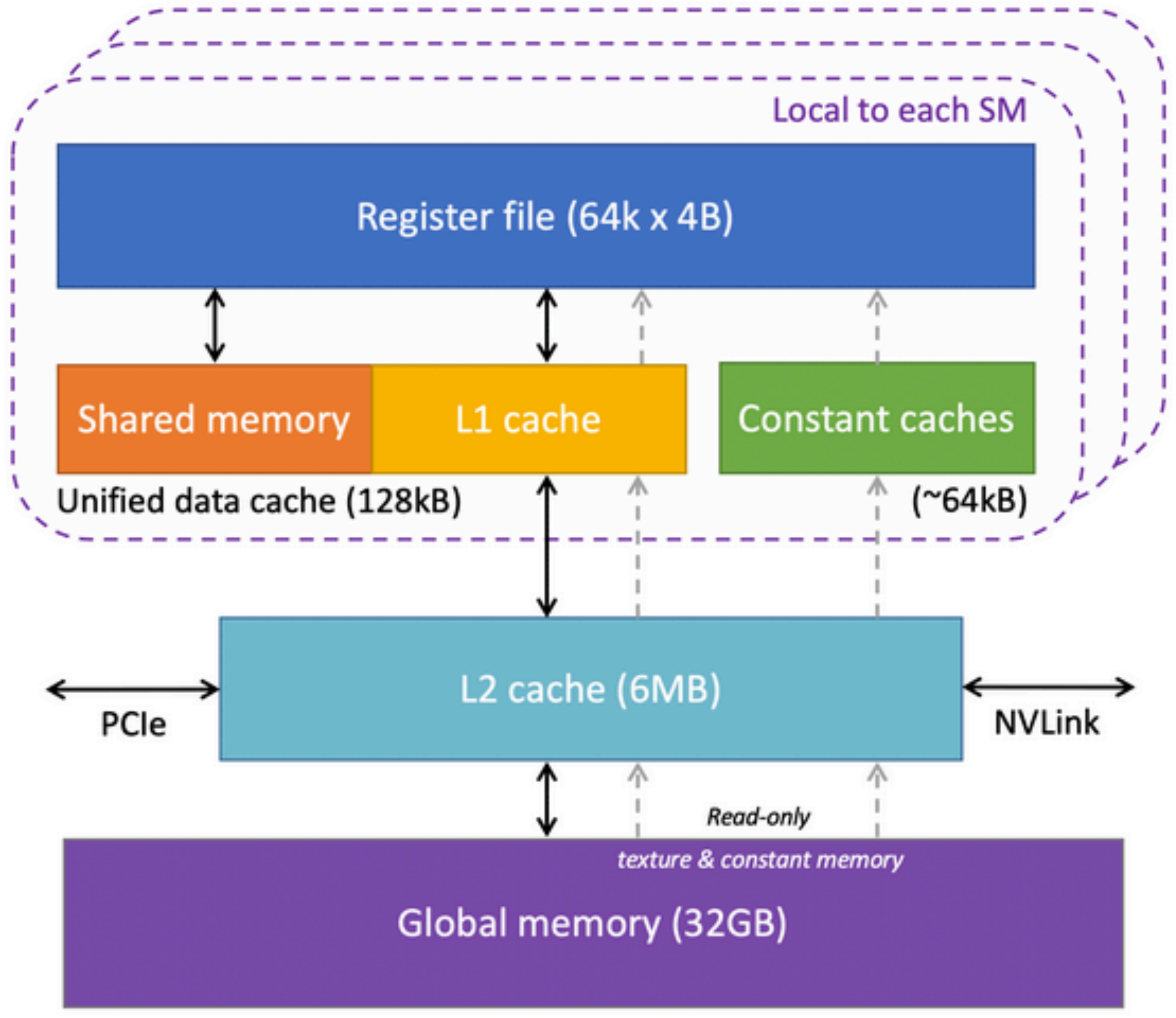


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

NVIDIA Volta Streaming Multiprocessor (SM) block diagram.

Memory Hierarchy in GPUs

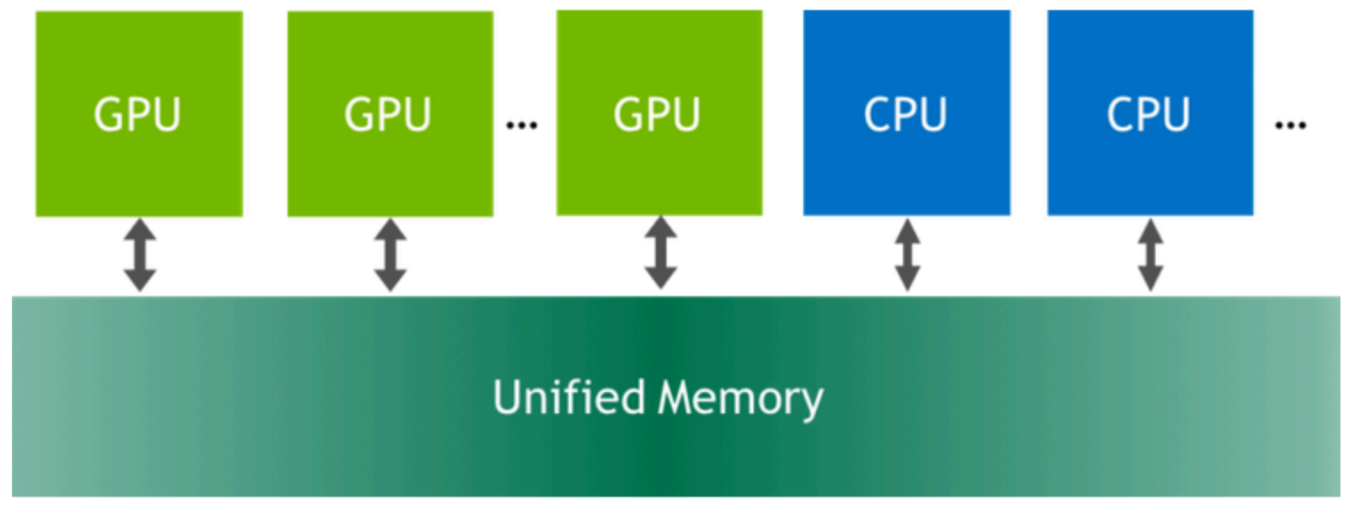
- Data location may have several hops to reach CUDA cores in a SM
- Memory closer to CUDA cores: *registers, L1 cache, shared memory, constant cache*
- Memory farther from CUDA cores: *L2 cache, global memory, local memory, texture* and *constant memory*
- Memory hierarchy similar to CPUs, but capacities vary; a SM has larger register files, L1 cache but lower global memory than a CPU
- Another memory considered with GPUs: *host memory*; data movement overhead reduced by sending data in larger batches



GPU memory levels and sizes for the NVIDIA Tesla V100.

Unified Memory System

- Single memory address space accessible from both CPU and GPU
- Unified memory allocated via **cudaMallocManaged()** call returns a pointer accessible from any processor
- Bytes of managed memory are first allocated on device memory, then host memory if needed (via page faults); Example: ***cudaunifiedmemory.cu***
- How to mitigate migration overhead between host and device in the above code?
    - Move initialization to the device kernel
    - Prefetch data to device before executing kernel (**cudaMemPrefetchAsync()**)



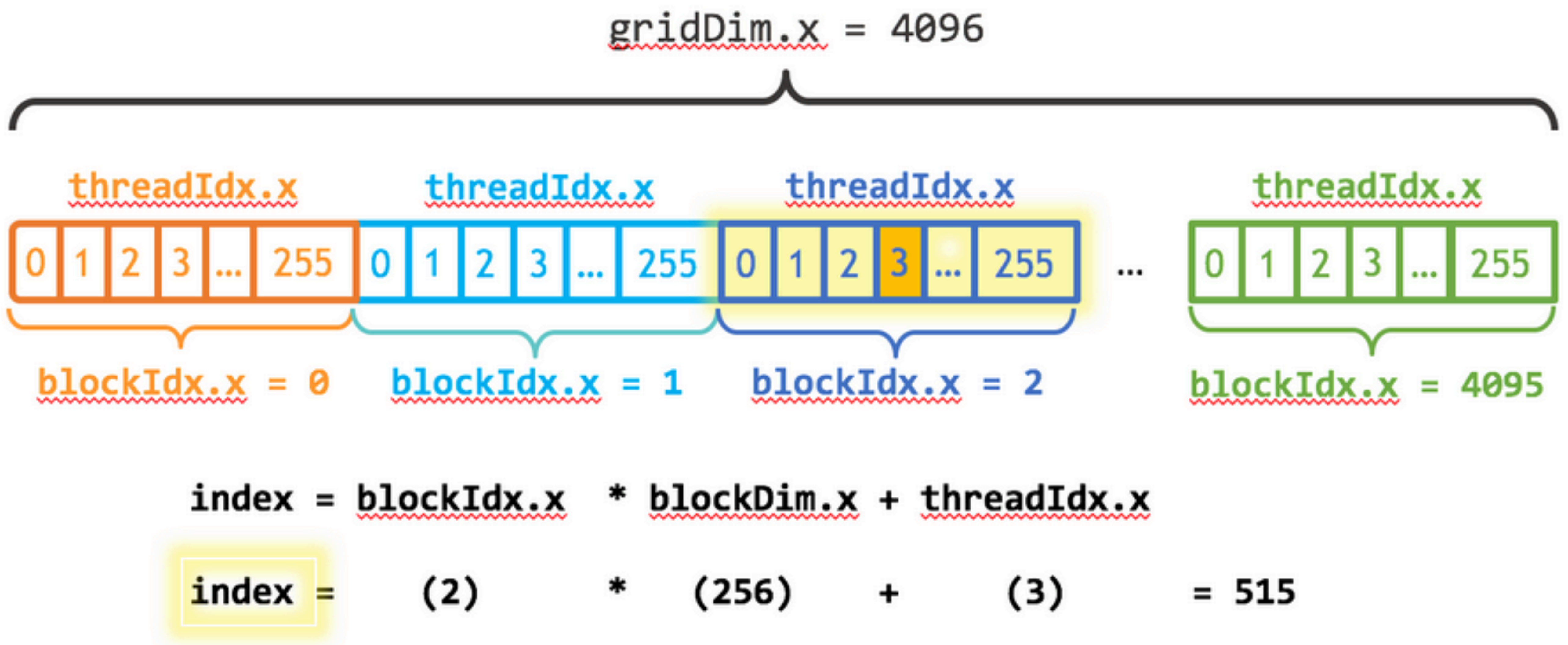
Kernel execution on GPU Hardware

- GPU kernels are executed on Streaming Multiprocessors (SMs) that contain CUDA cores
- Set of 32 cores arranged in SMs to execute full warp of threads
- Number of SMs used to execute a kernel call depends on the execution configuration: **<<<x, y>>>**
- **'x'** is the number of thread blocks and **'y'** is the number of threads per block
- A collection of subsequent blocks forms a **grid**
- Each of the **'x'** blocks is assigned to a different SM; each SM divides **'y'** threads in its current block into warps of 32 for execution
- SMs thus run multiple blocks independently in parallel on the GPU

Kernel execution on GPU Hardware…

- Each thread has a unique global ID, marked by ‘**index**’
- Helps execute thread-specific code in parallel, rather than perform whole compute on each thread
- We’ll see more in the ***cudaforloop.cu*** example

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```



CUDA C/C++ Examples

- Heterogeneous computing (executing on CPU *or* GPU): ***testgpu.cu***
- Get device statistics: ***devicequery.cu*** or via **NVIDIA SMI**
- Using CUDA Events and profiler: ***cudaevent.cu*** and **nvprof**
- Memory allocations with CUDA: ***cudamemory.cu, cuda_optimized_unifiedmem.cu, cudaprefetchunifiedmem.cu***
- Different types of memory allocations in CUDA: ***cudamalloctests.cu***
- Naive/True parallelization with CUDA: ***cudaforloop.cu*** and ***cudagridstride.cu***
- Parallelized vector addition: ***cuda_vectoraddition.cu, cudasaxpy.cu***
- Accelerating matrix multiplication: ***cuda_matrixmultiplication.cu*** (https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/)
- CUDA error handling: ***cuda_errorhandling.cu***

Thank you