2022 IEEE IUS

International Ultrasonics Symposium

Venice, Italy || October 10-13, 2022

us4us®

IPPT PAN

Ultrasound Signal Processing with GPUs – Introduction to Parallel Programming

NVIDIA CUDA

Presenter: Dr Marcin Lewandowski

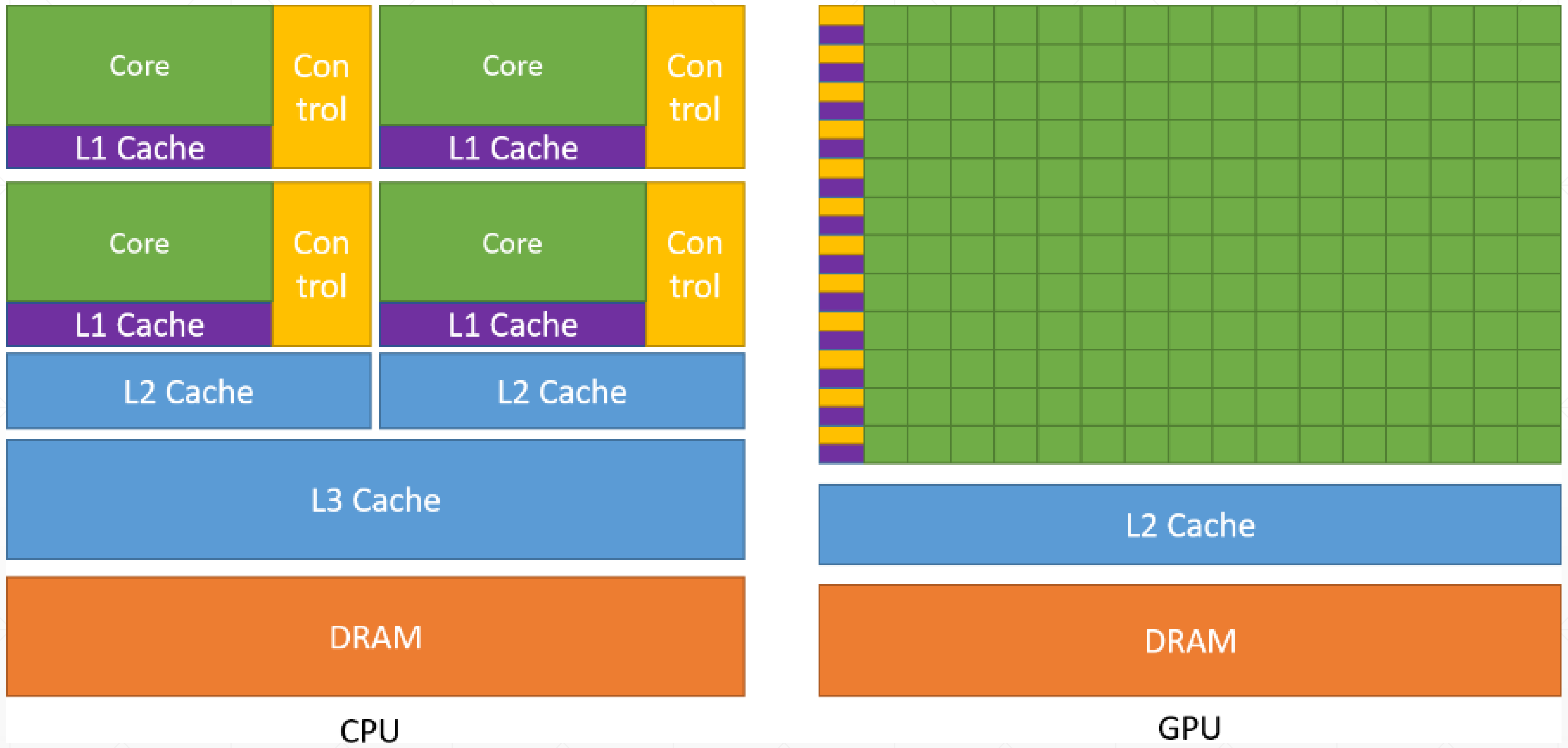
License / Attribution

GPU Architecture

Flynn Taxonomy of parallelism

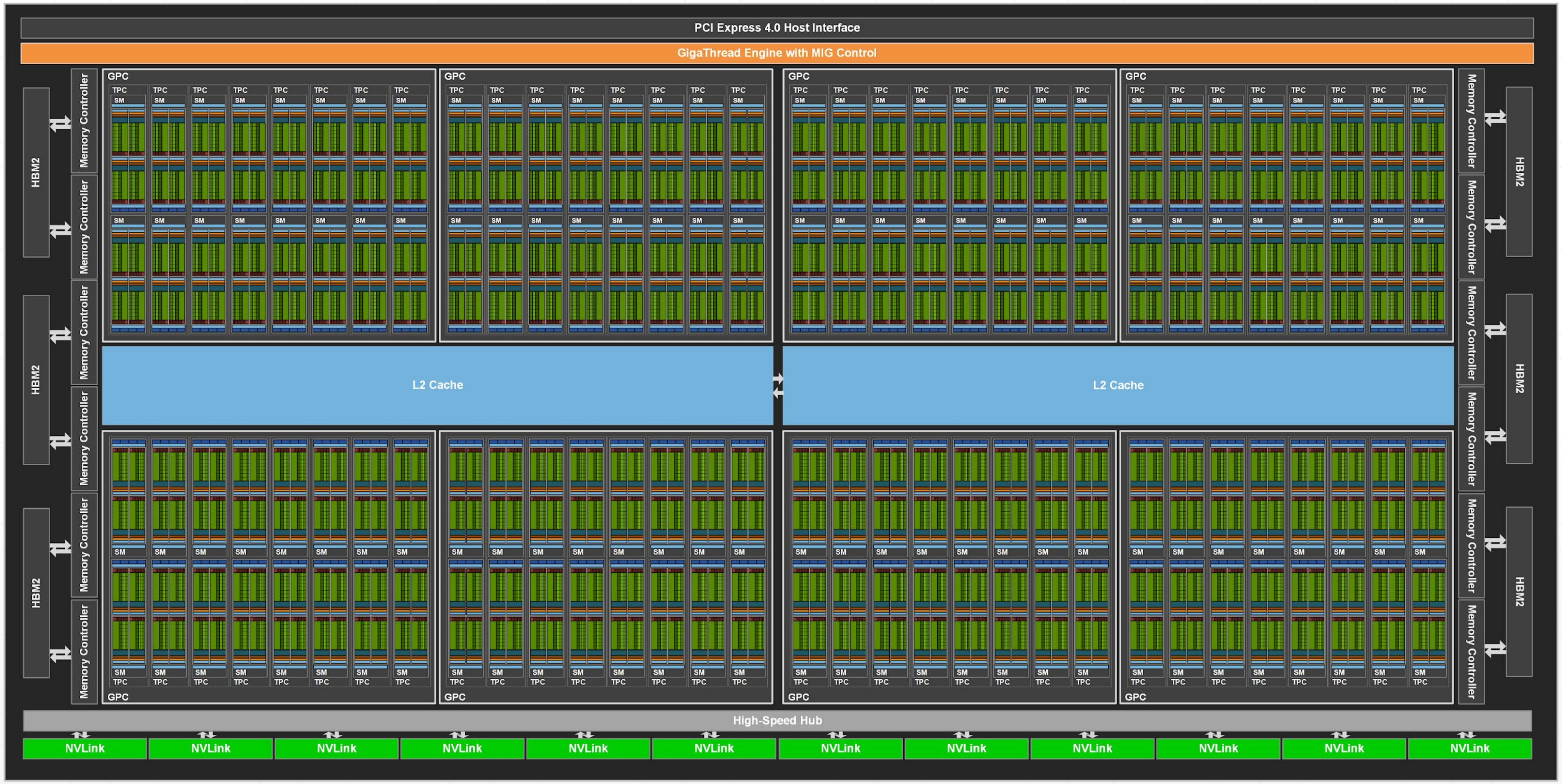
- Two dimensions:
  - Number of instruction streams: single vs. multiple
  - Number of data streams: single vs. multiple
- **SISD** – single-instruction single-data
  - Pipelining and ILP (Instruction Level Parallelism) on a uniprocessor
- **SIMD** – single-instruction multiple-data (aka Vector processor)
  - DLP (Data Level Parallelism) on a vector processor
- **MIMD** – multiple-instruction multiple-data
  - DLP, TLP (Thread Level Parallelism) on a parallel processor
  - SPMD: single-program multiple data

Architecture CPU vs. GPU



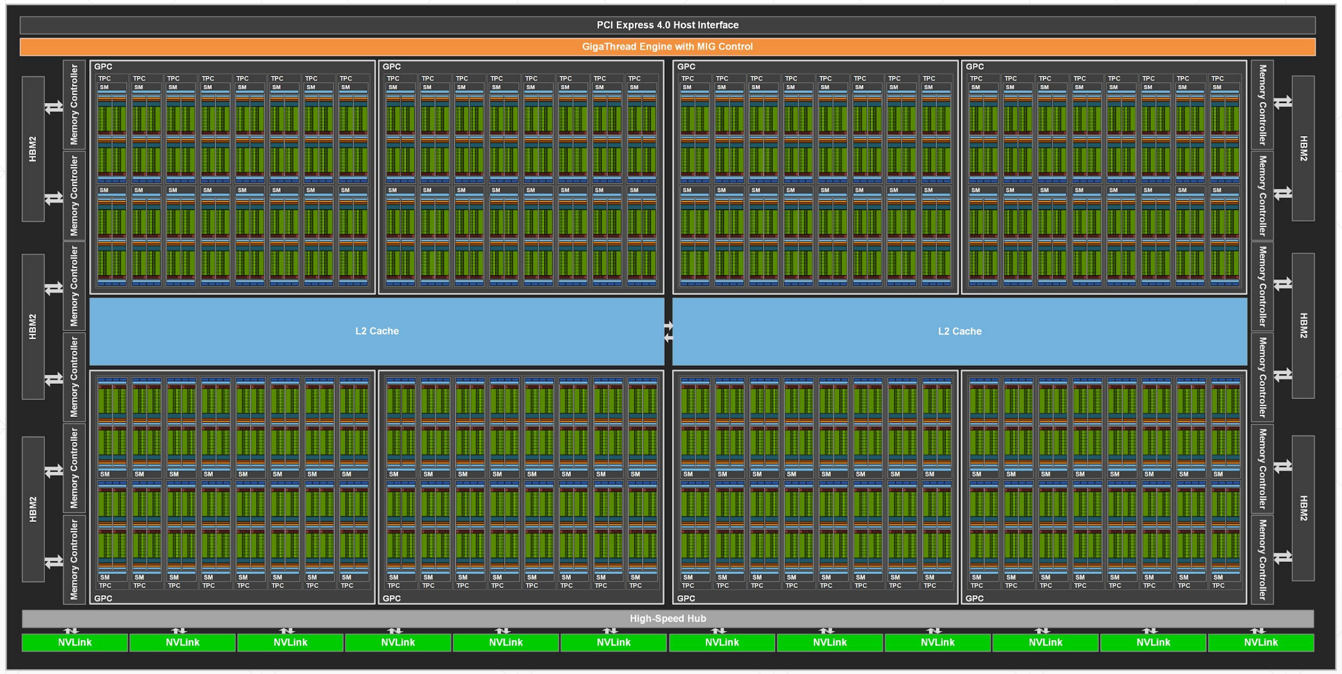
Source: https://docs.nvidia.com/cuda/

NVIDIA Ampere Architecture

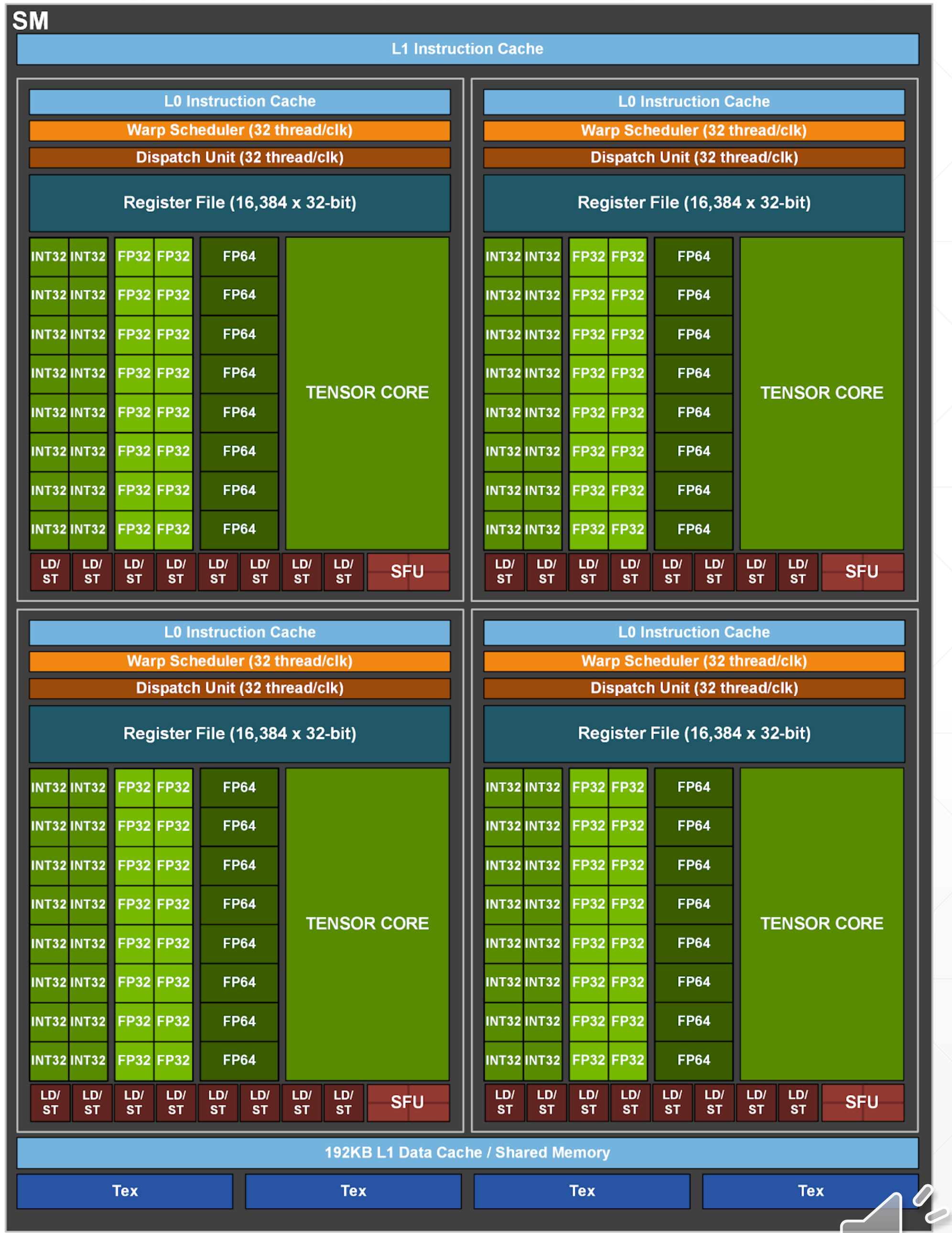


Source: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

NVIDIA Streaming Multiprocessor (SM)



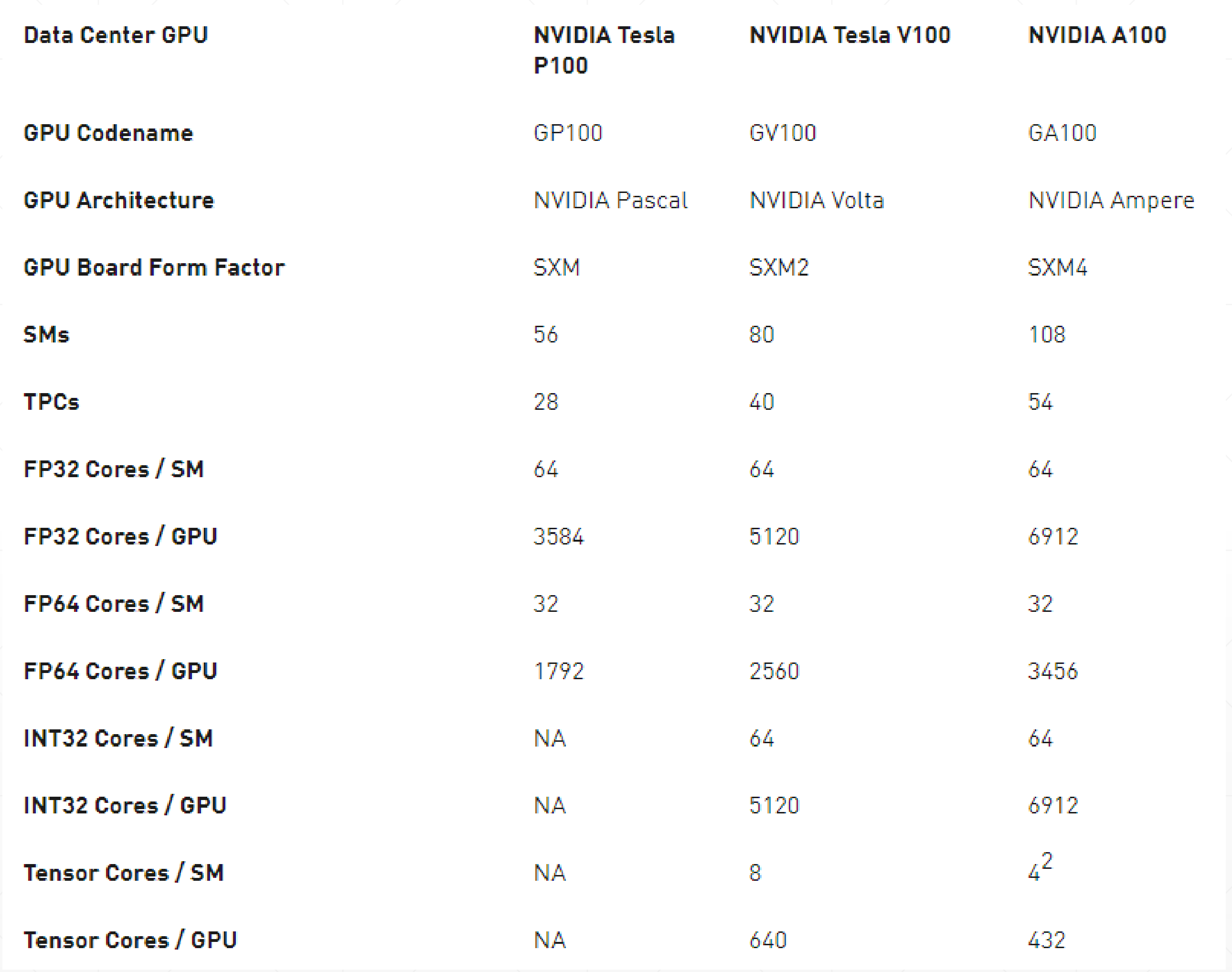
PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink



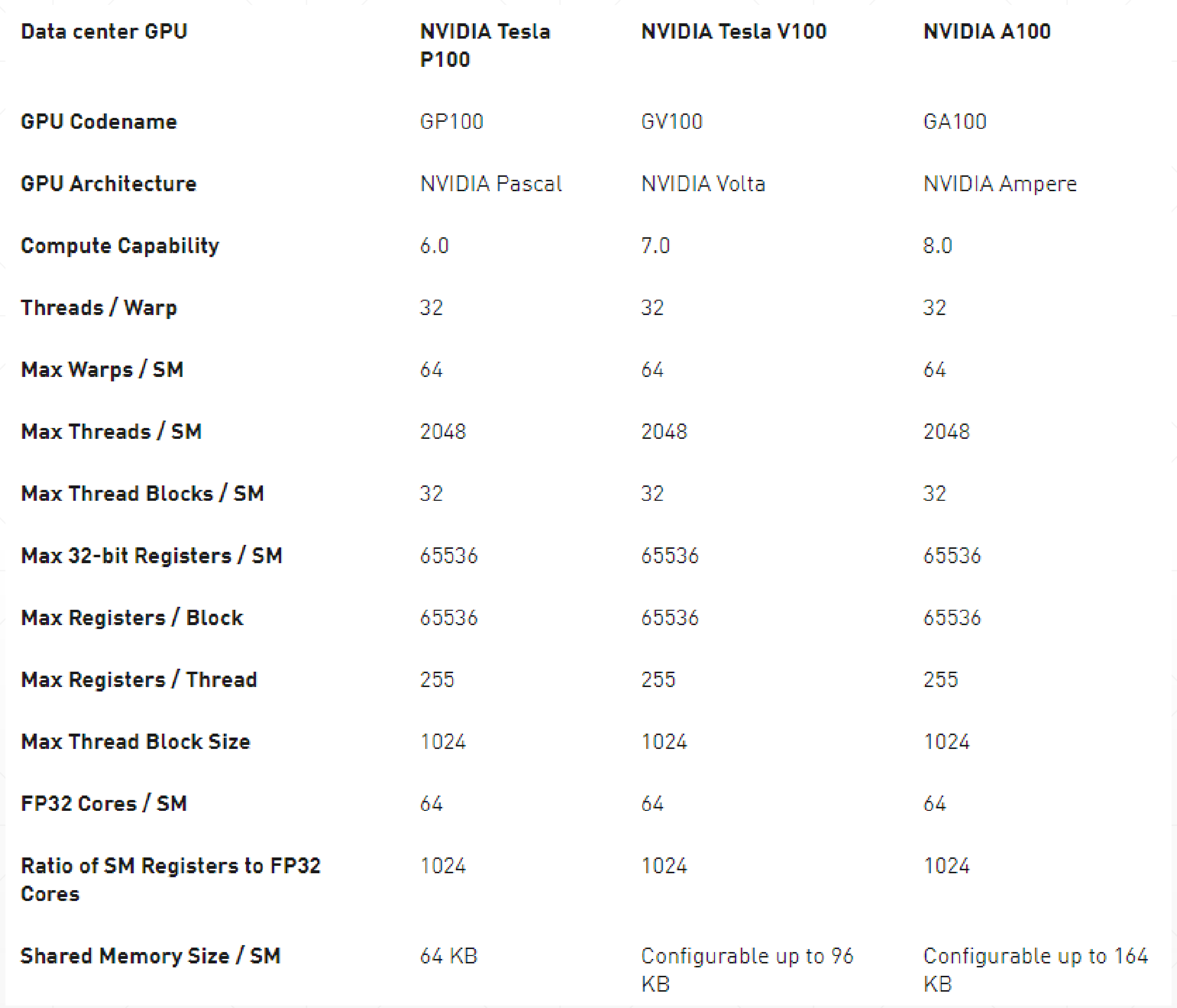
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Source: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Just a few numbers …



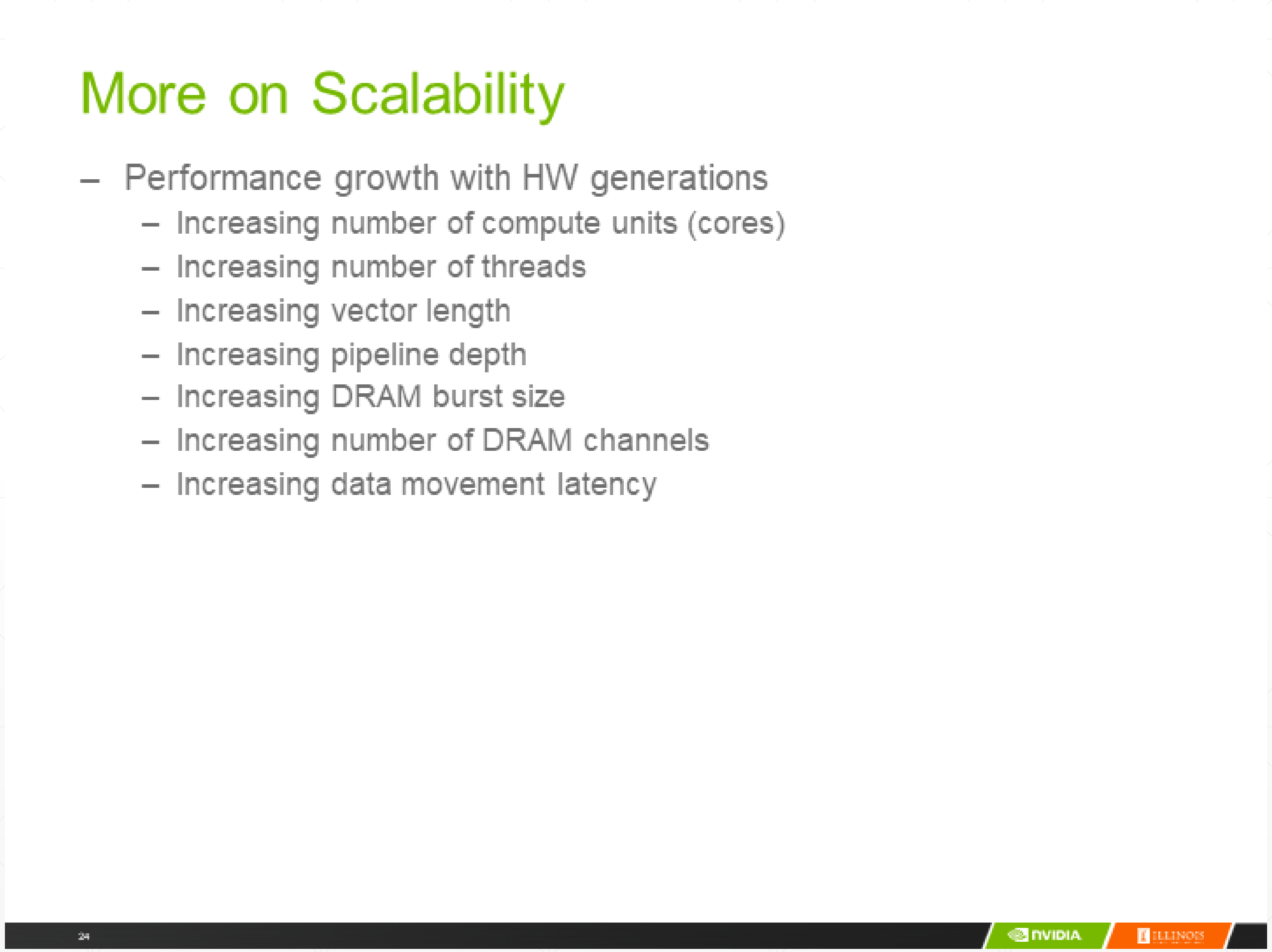
Data Center GPU	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM	32	32	32
FP64 Cores / GPU	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	4[2]
Tensor Cores / GPU	NA	640	432



Data center GPU	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
Compute Capability	6.0	7.0	8.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks / SM	32	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB

Source: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Generation to generation …



More on Scalability

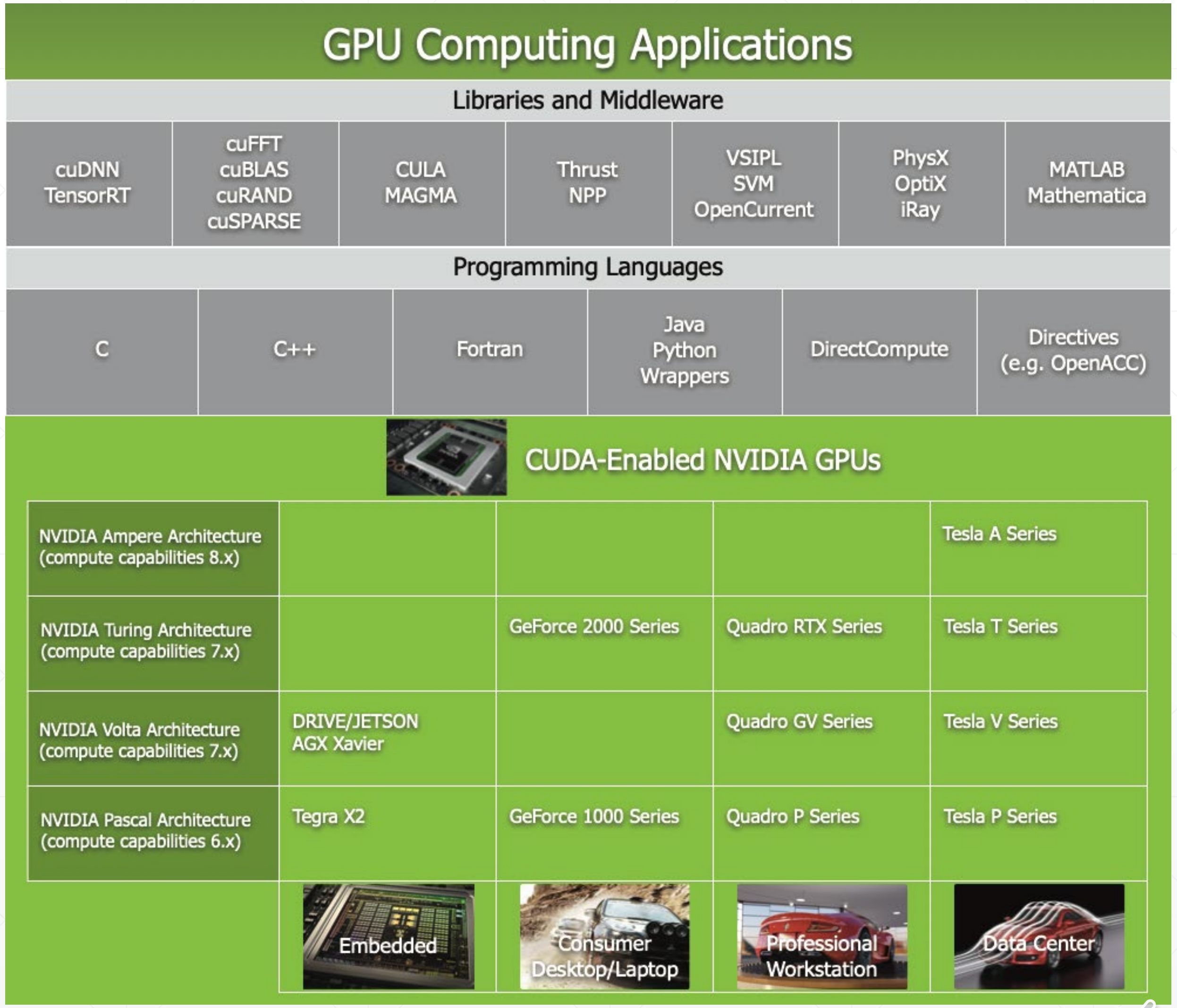
– Performance growth with HW generations
  – Increasing number of compute units (cores)
  – Increasing number of threads
  – Increasing vector length
  – Increasing pipeline depth
  – Increasing DRAM burst size
  – Increasing number of DRAM channels
  – Increasing data movement latency

24

CUDA Architecture

What is CUDA® (Compute Unified Device Architecture)

- NVIDIA CUDA is a General-Purpose Parallel Computing Platform and Programming Model
- Introduced back in 2006 by NVIDIA® to leverage the parallel compute engine in NVIDIA GPUs.
- CUDA is designed to support various languages and application programming interfaces.
- Now, other hardware platforms support CUDA programming (e.g. FPGA, Intel® OneAPI).



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

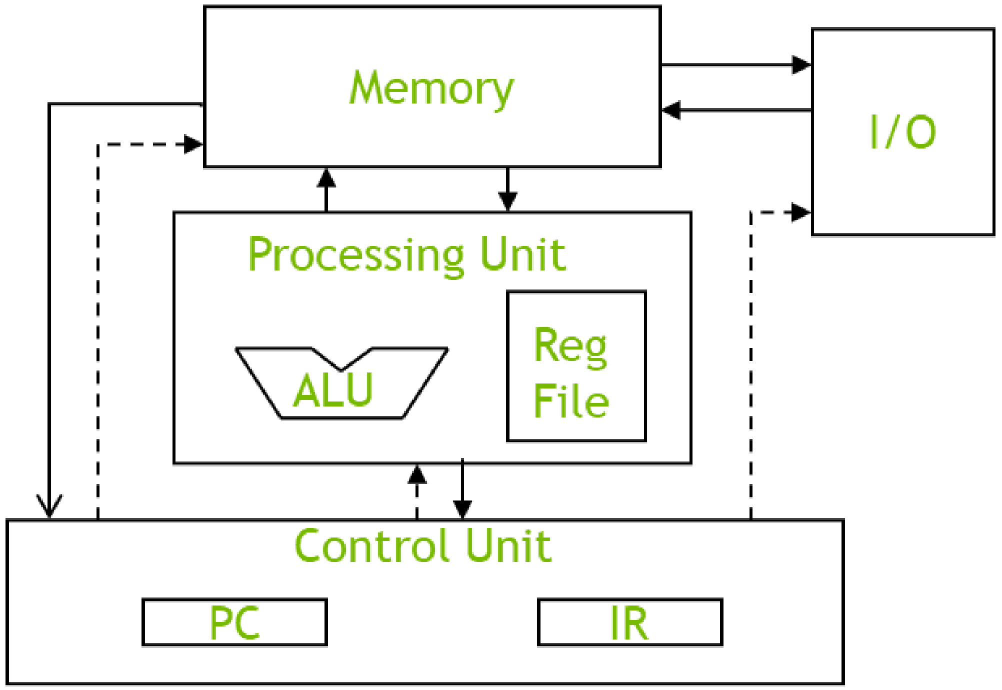
CUDA-Enabled NVIDIA GPUs

	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center
NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series

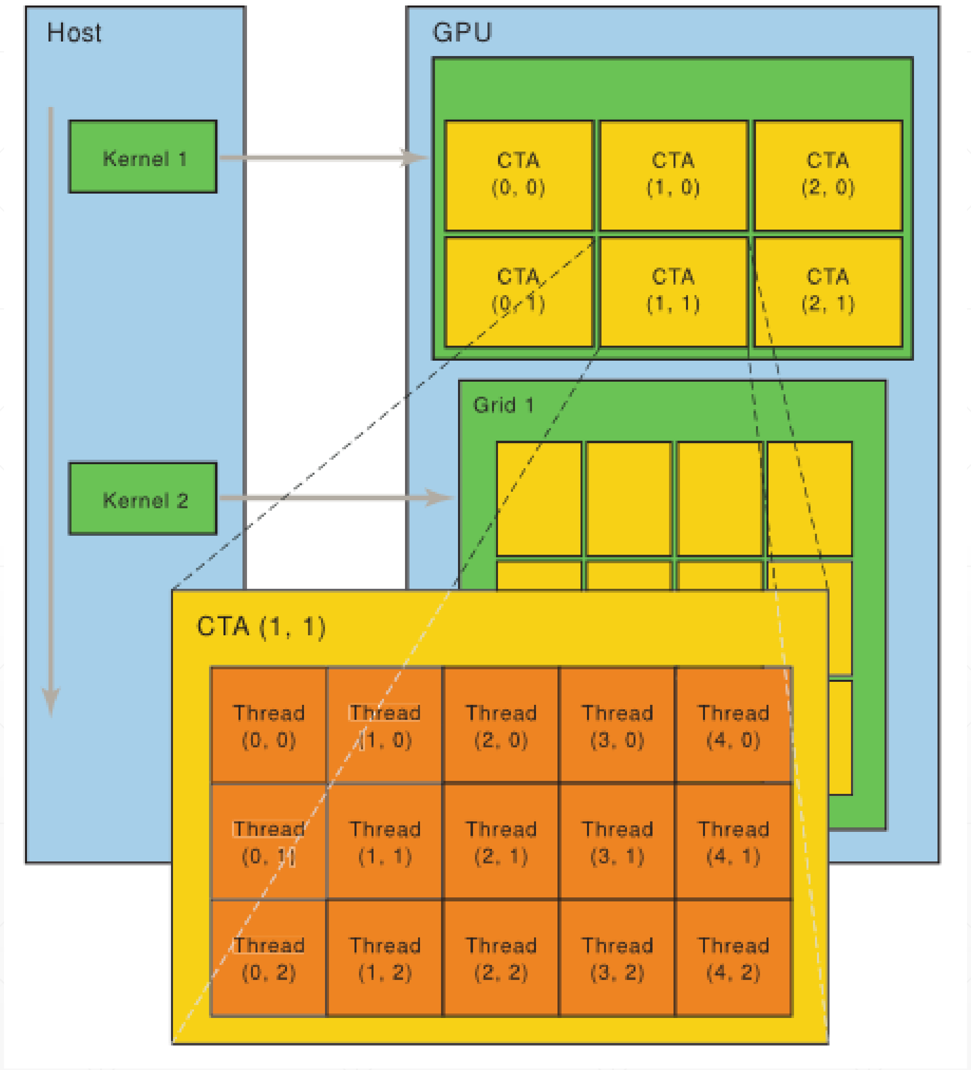
Source: https://docs.nvidia.com/cuda/

A Thread as a Von-Neumann Processor

A thread is a "virtualized" or
"abstracted"
Von-Neumann Processor



CUDA – Grid of Cooperative Thread Arrays

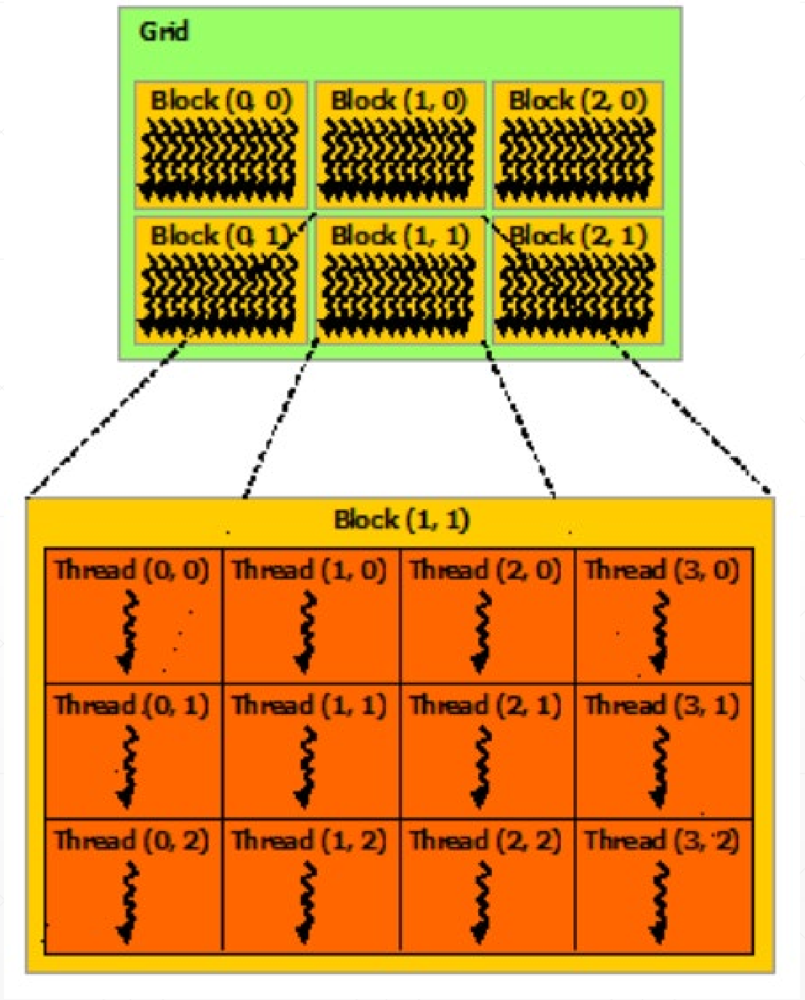


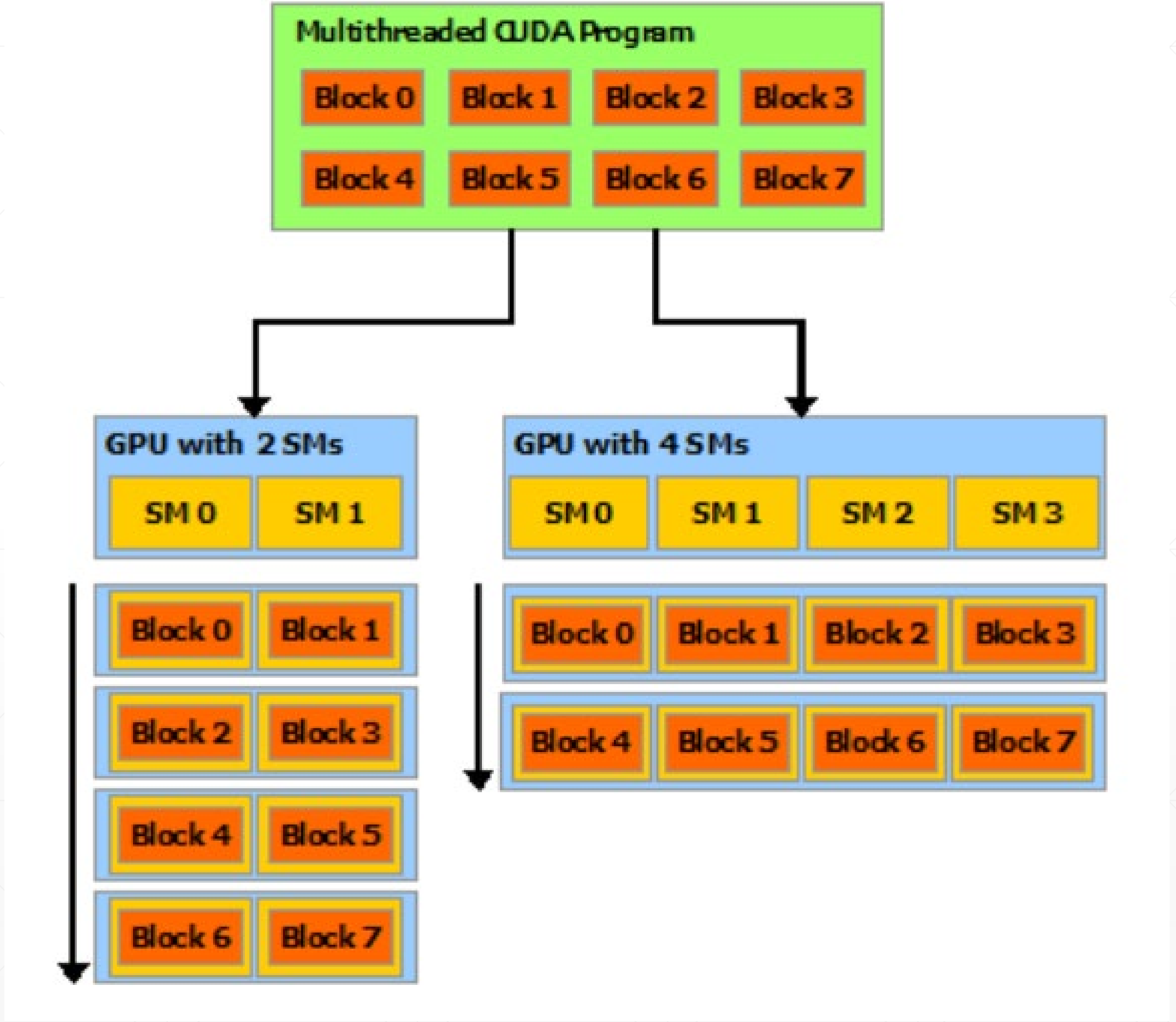
Source: https://docs.nvidia.com/cuda/

Source: https://docs.nvidia.com/cuda/

CUDA / GPU Automatic Scalability

Grid of Thread Blocks



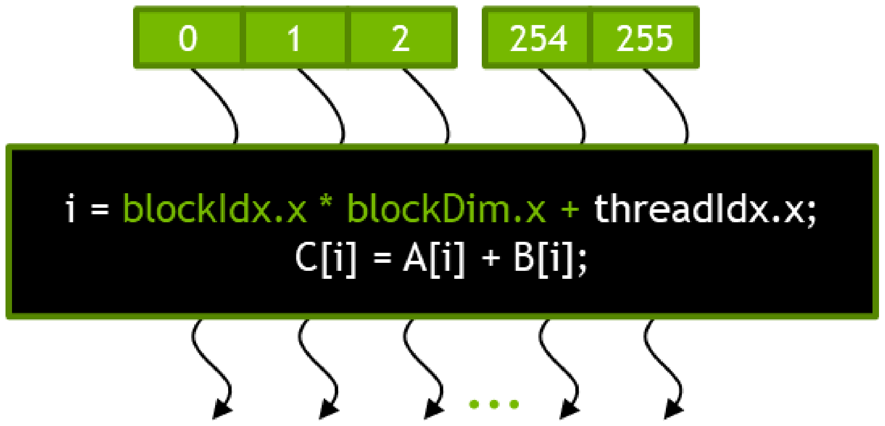


Source: https://docs.nvidia.com/cuda/

Source: https://docs.nvidia.com/cuda/

Arrays of Parallel Threads

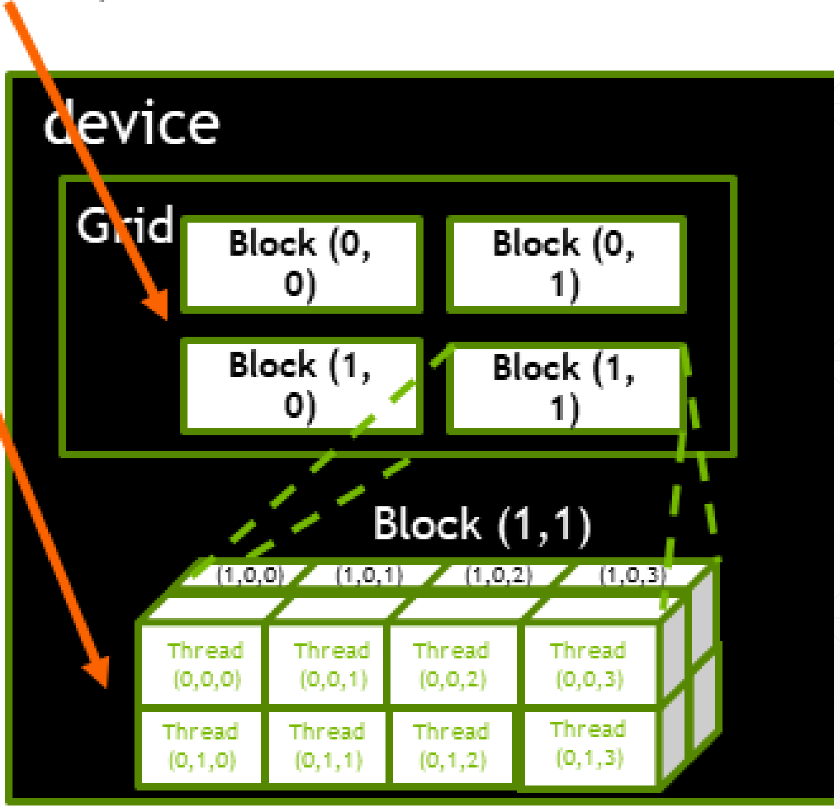
- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



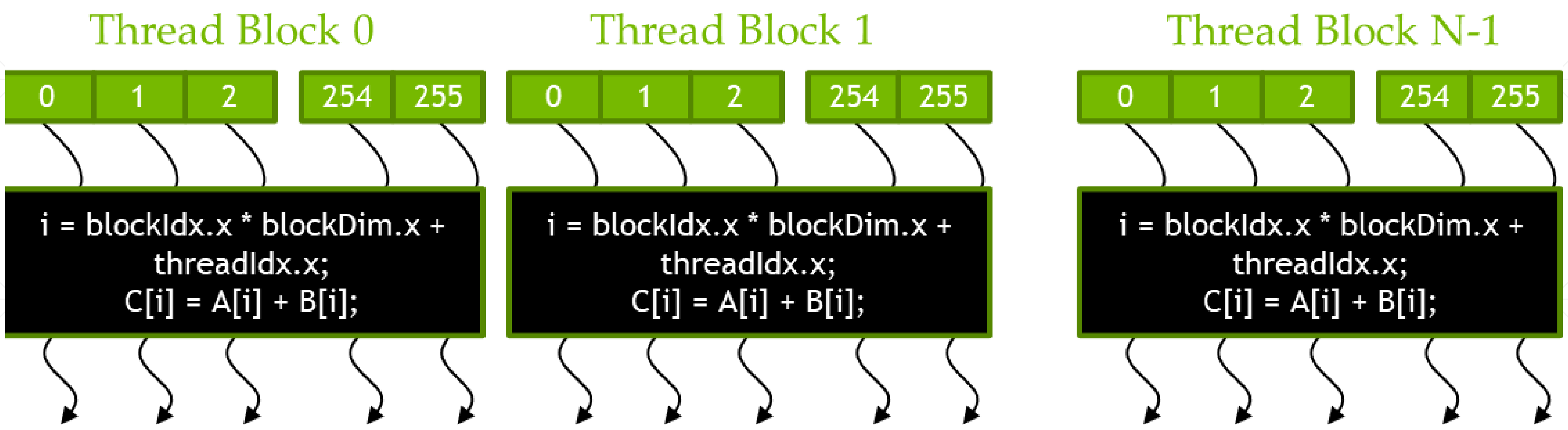
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Thread Blocks: Scalable Cooperation

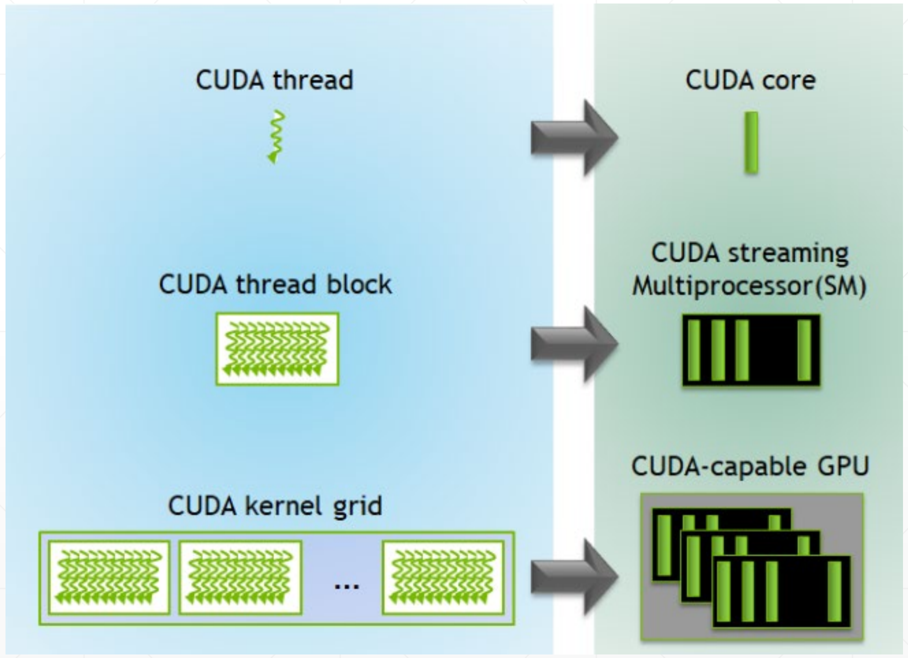


```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

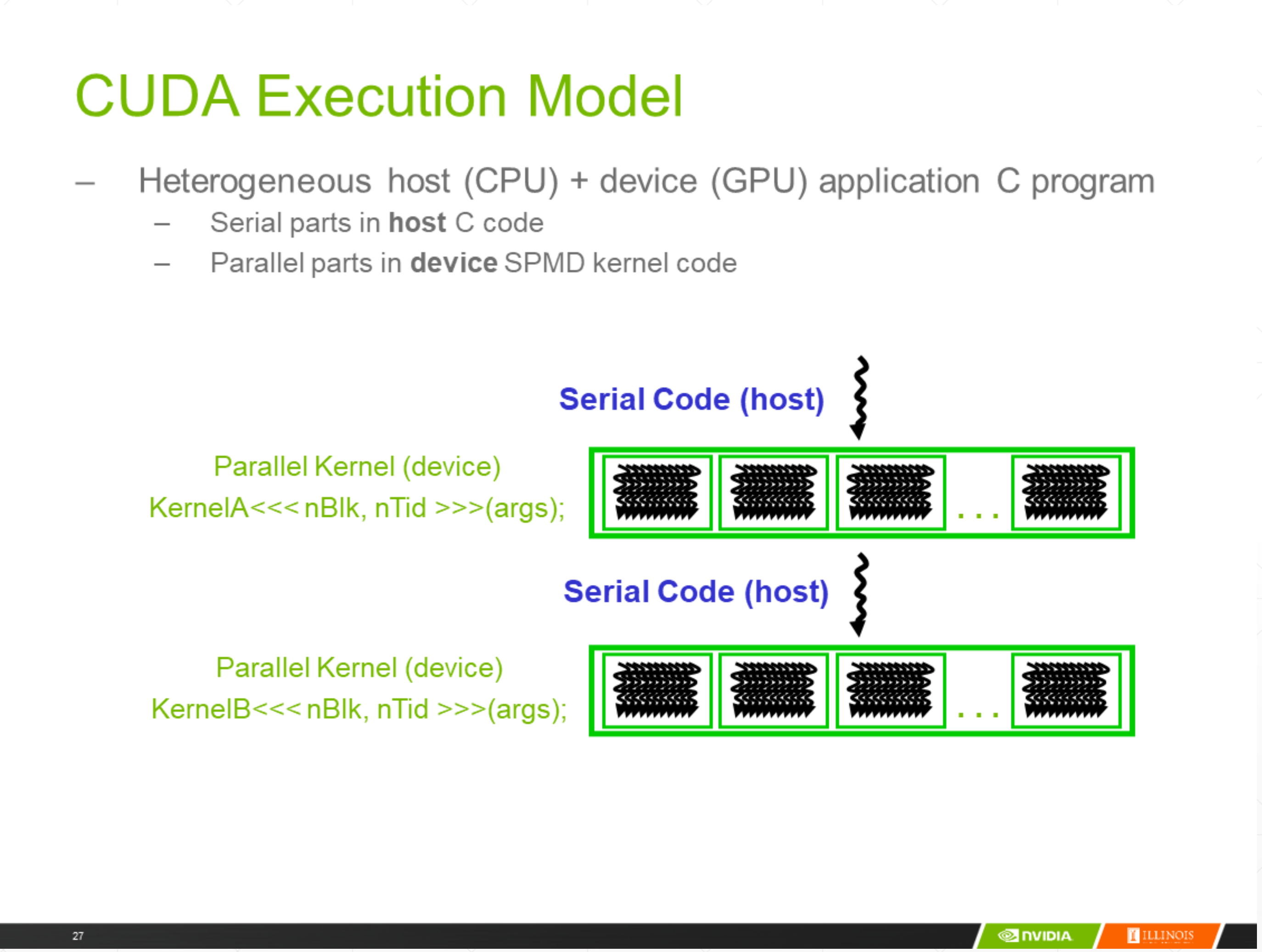
- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

Thread Blocks / Grids

- Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU.
- One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks.
- Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.
- CUDA defines built-in 3D variables for threads and blocks
- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).



Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/



CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

. . .

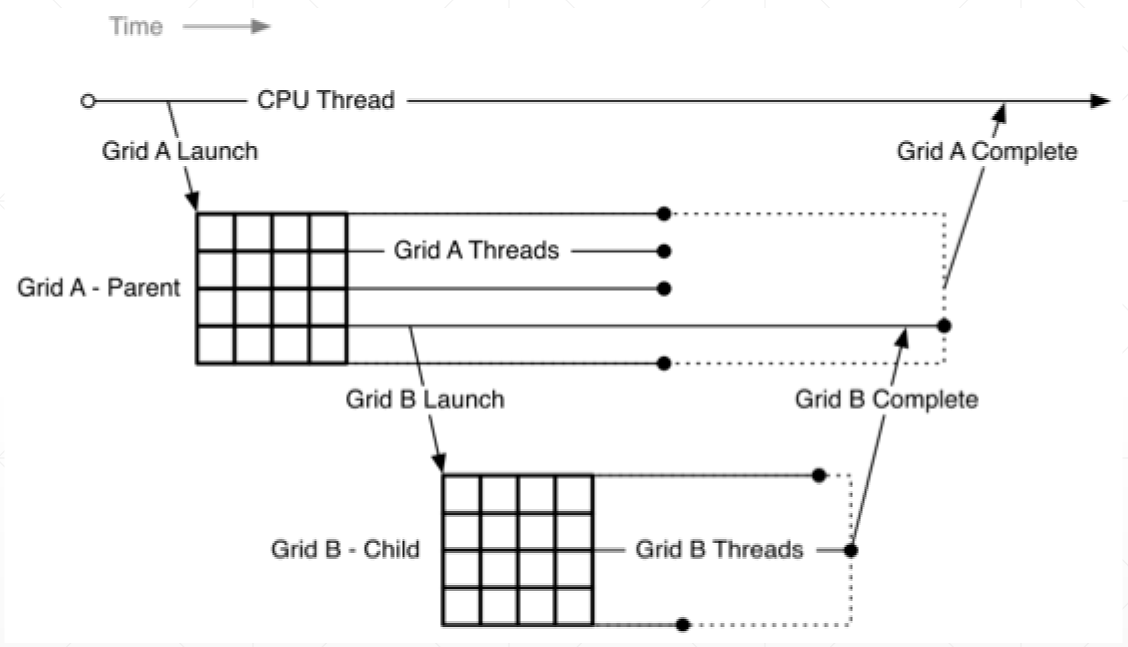
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

. . .

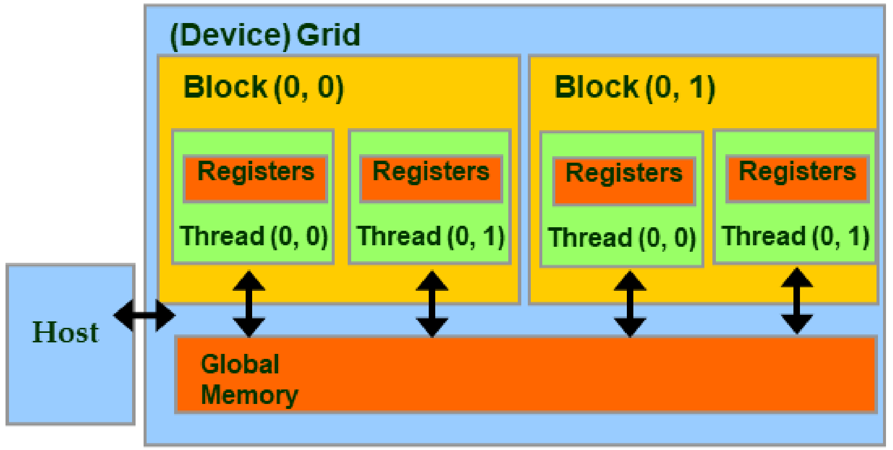
27

Dynamic Parallelism



Source: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/

Partial Overview of CUDA Memories

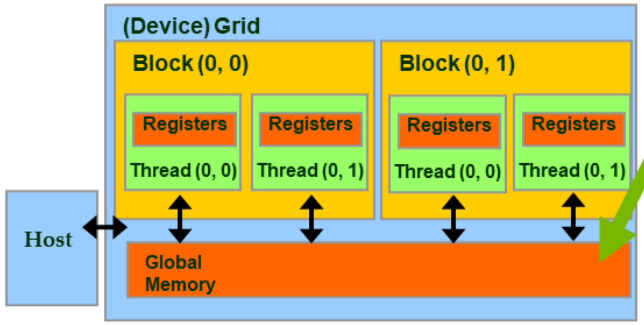


- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

We will cover more memory types and more sophisticated memory models later.

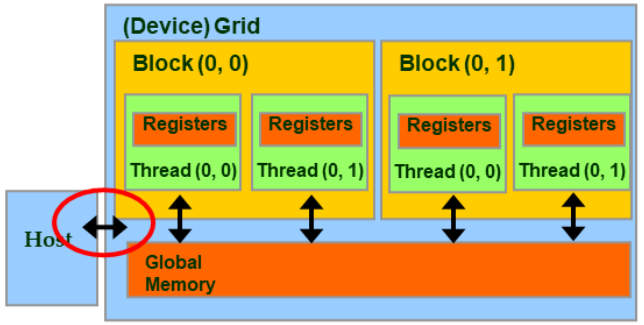
Host/Device Memory

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

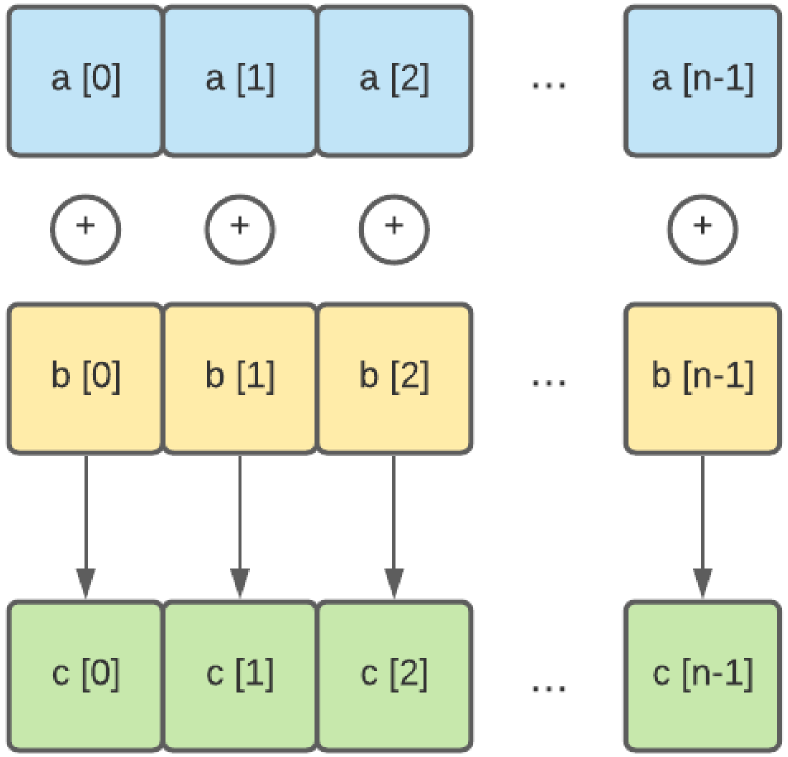
Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

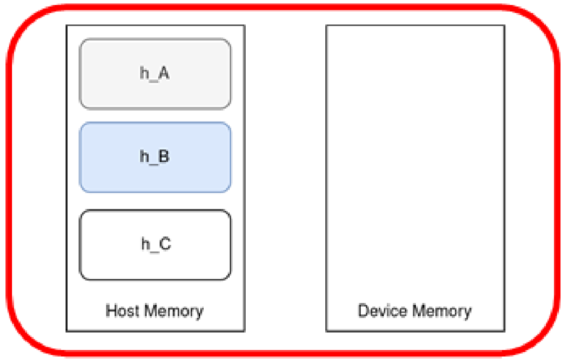
Vector Addition – Traditional C Code

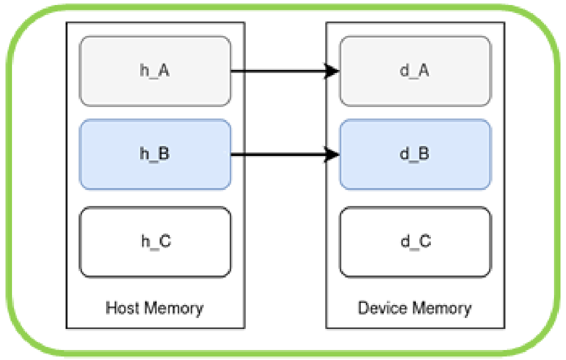
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

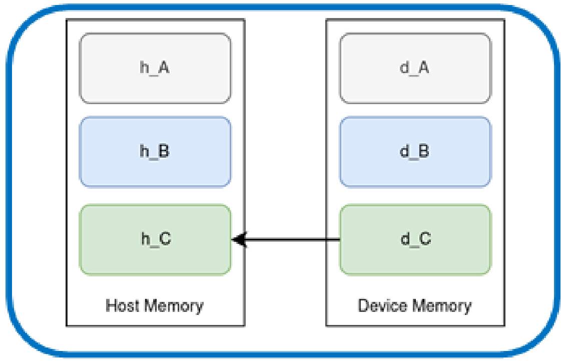


Vector Addition, Explicit Memory Management

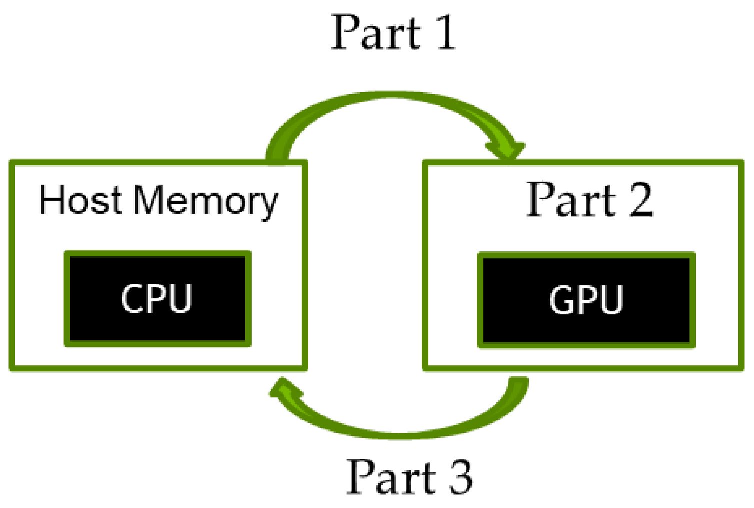
```
… Allocate h_A, h_B, h_C …

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);

   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}

… Free h_A, h_B, h_C …
```







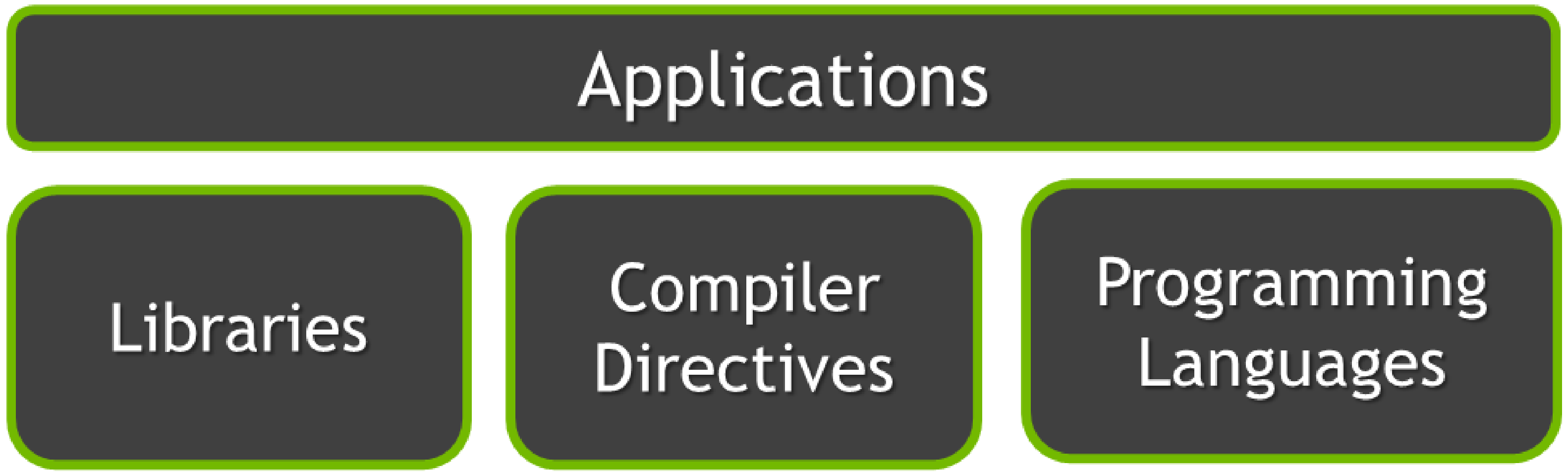
Heterogeneous Computing vecAdd CUDA Host Code



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
  float *d_A, *d_B, *d_C;
  // Part 1
  // Allocate device memory for A, B, and C
  // copy A and B to device memory

  // Part 2
  // Kernel launch  code – the device performs the actual
vector addition

  // Part 3
  // copy C from the device memory
  // Free device vectors
}
```

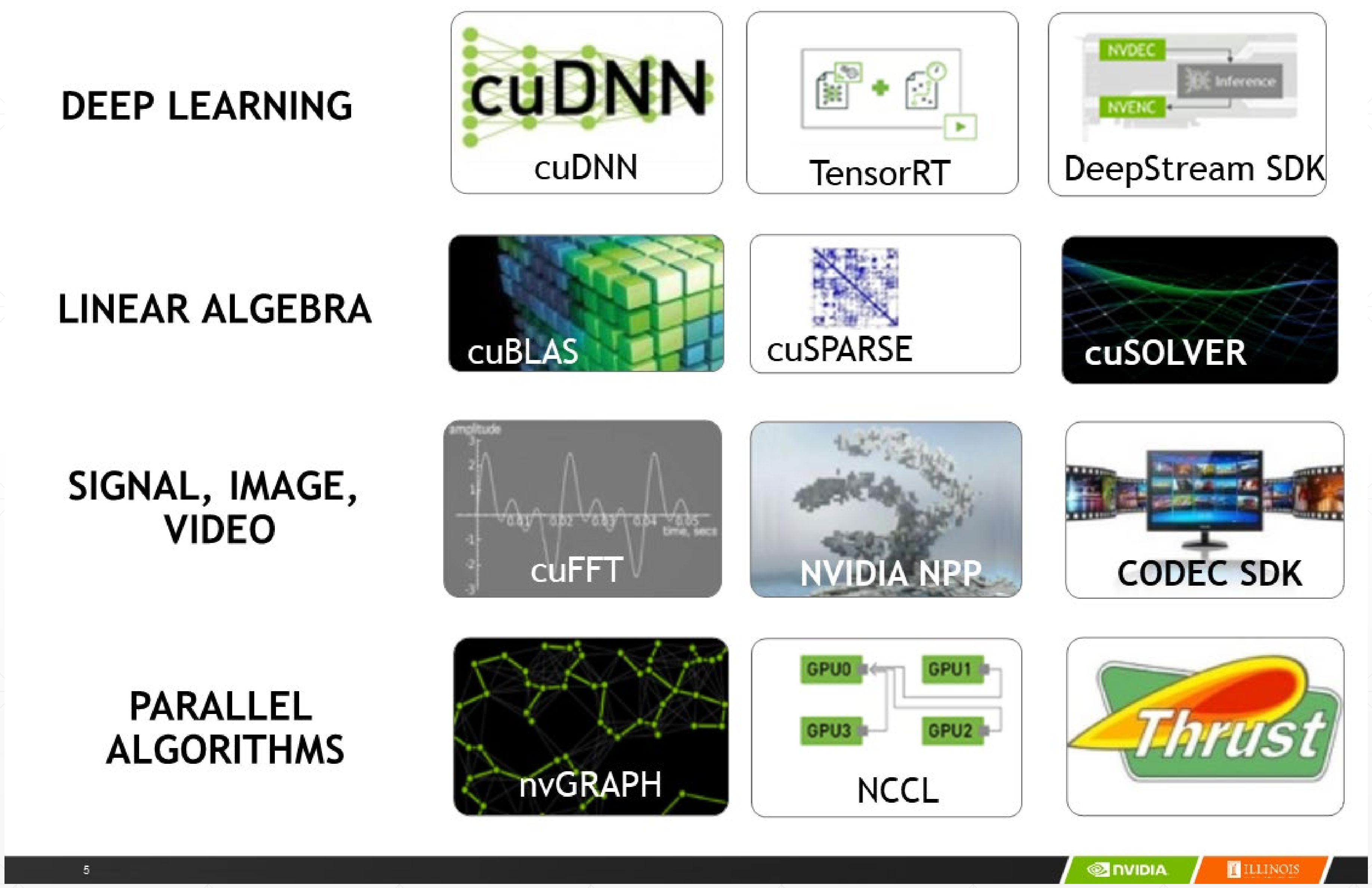
GPU Acceleration

3 Ways to Accelerate Applications



Easy to use	Easy to use	Most Performance
Most Performance	Portable code	Most Flexibility

NVIDIA GPU Accelerated Libraries



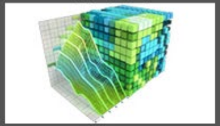
DEEP LEARNING	cuDNN	TensorRT	DeepStream SDK
LINEAR ALGEBRA	cuBLAS	cuSPARSE	cuSOLVER
SIGNAL, IMAGE, VIDEO	cuFFT	NVIDIA NPP	CODEC SDK
PARALLEL ALGORITHMS	nvGRAPH	NCCL	Thrust

NVDEC
Inference
NVENC
GPU0
GPU1
GPU3
GPU2

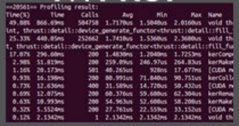
Developer Tools - Profilers

NSIGHT

NVVP



NVPROF



```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252662  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2040ms  1.7253ms  kerCompu
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerRenam
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  void th
```

NVIDIA Provided

TAU

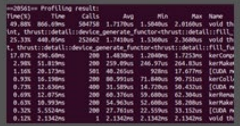
VampirTrace

3rd Party

https://developer.nvidia.com/performance-analysis-tools

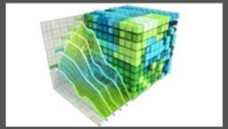
Profiling Tools

nvprof



```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252662  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2040ms  1.7253ms  kerCompu
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerRenam
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  void th
```

NVVP



Phasing out

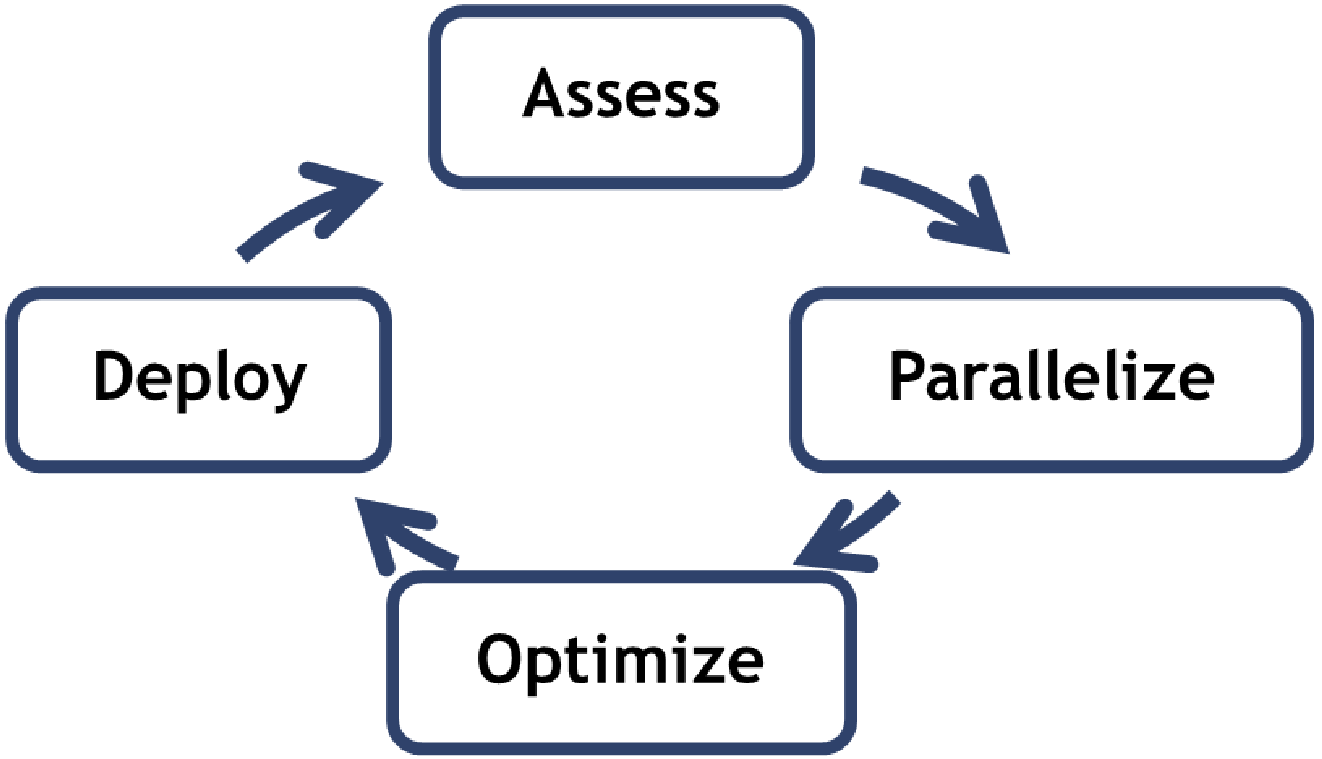
Nsight Compute

Nsight Systems

Current

See lecture 2-4 for an overview of all tools

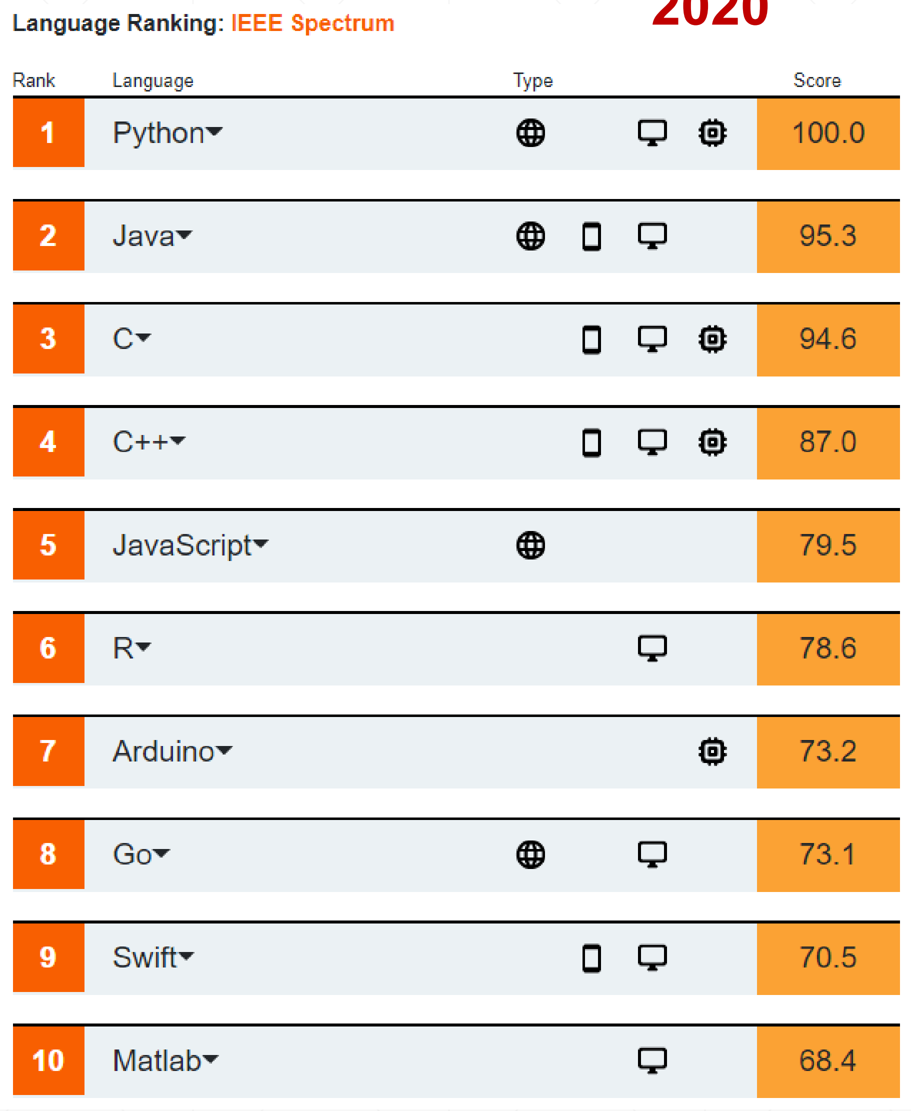
Optimization



Why Python!?

Why is Python so Popular?

1. Easy to learn and use.
2. Mature and supportive Python Community.
3. Support from Big-Players (Corporate Sponsors).
4. “Batteries Included“ – hundreds of libraries and frameworks available.
5. Versatility, efficiency, reliability, and speed. 🤞
6. Big data, machine learning and Cloud computing.
7. First-choice Language (see the ranking!).
8. The flexibility of Python language.
9. Use of Python in academics.
10. Automation



2020

Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	🌐 🖥 ⚙	100.0
2	Java	🌐 📱 🖥	95.3
3	C	📱 🖥 ⚙	94.6
4	C++	📱 🖥 ⚙	87.0
5	JavaScript	🌐	79.5
6	R	🖥	78.6
7	Arduino	⚙	73.2
8	Go	🌐 🖥	73.1
9	Swift	📱 🖥	70.5
10	Matlab	🖥	68.4

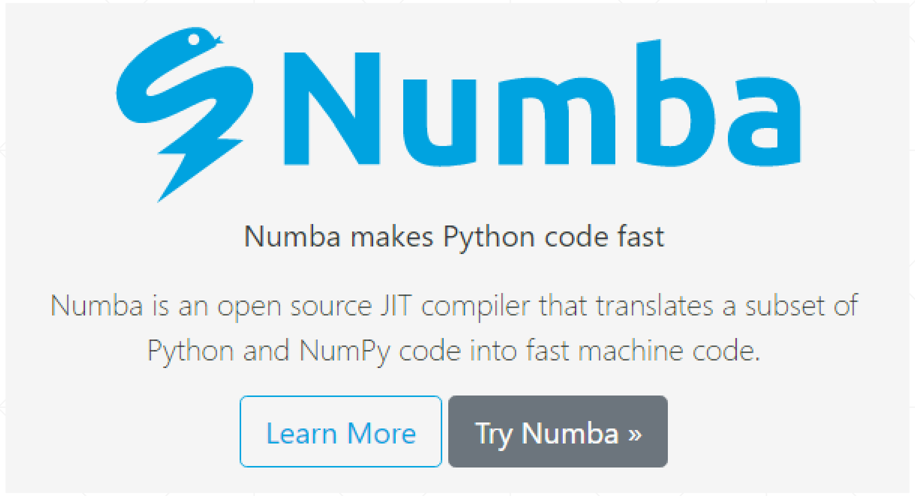
source: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

There should be one – and preferably only one – obvious way to do it (The Zen of Python) 😉

- There are many solutions for GPU acceleration in Python …
- Python GPU programming:
  - **NUMBA**, pyCUDA, pyOpenCL
- Libraries:
  - Numpy on the GPU: **CuPy**
  - Numpy on the GPU (again): Jax
  - Pandas on the GPU: RAPIDS cuDF
  - Scikit-Learn on the GPU: RAPIDS cuML
- Frameworks:
  - deep learning frameworks like PyTorch, TensorFlow, Caffe, MXNet

NUMBA

- Accelerate Python Functions
- Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.
- You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

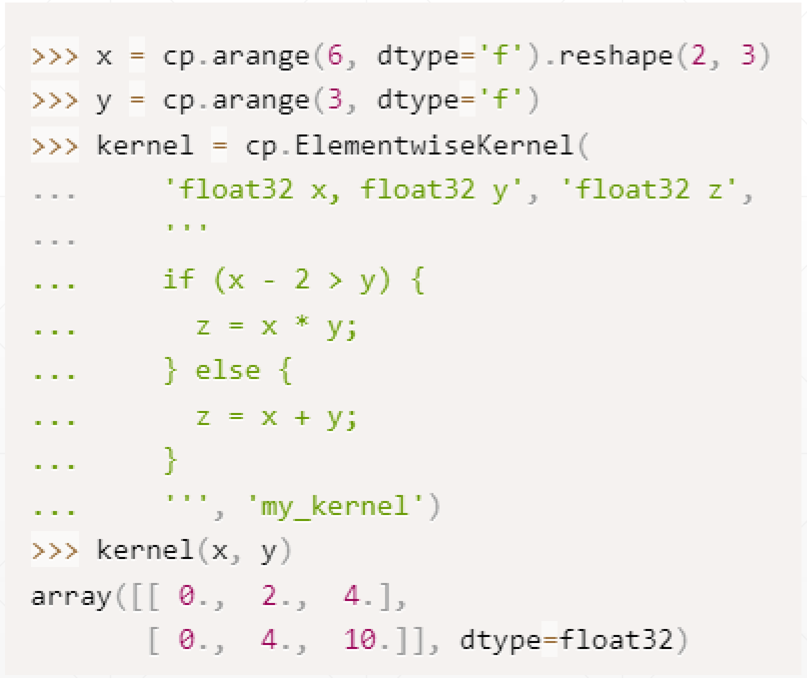

Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

```python
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

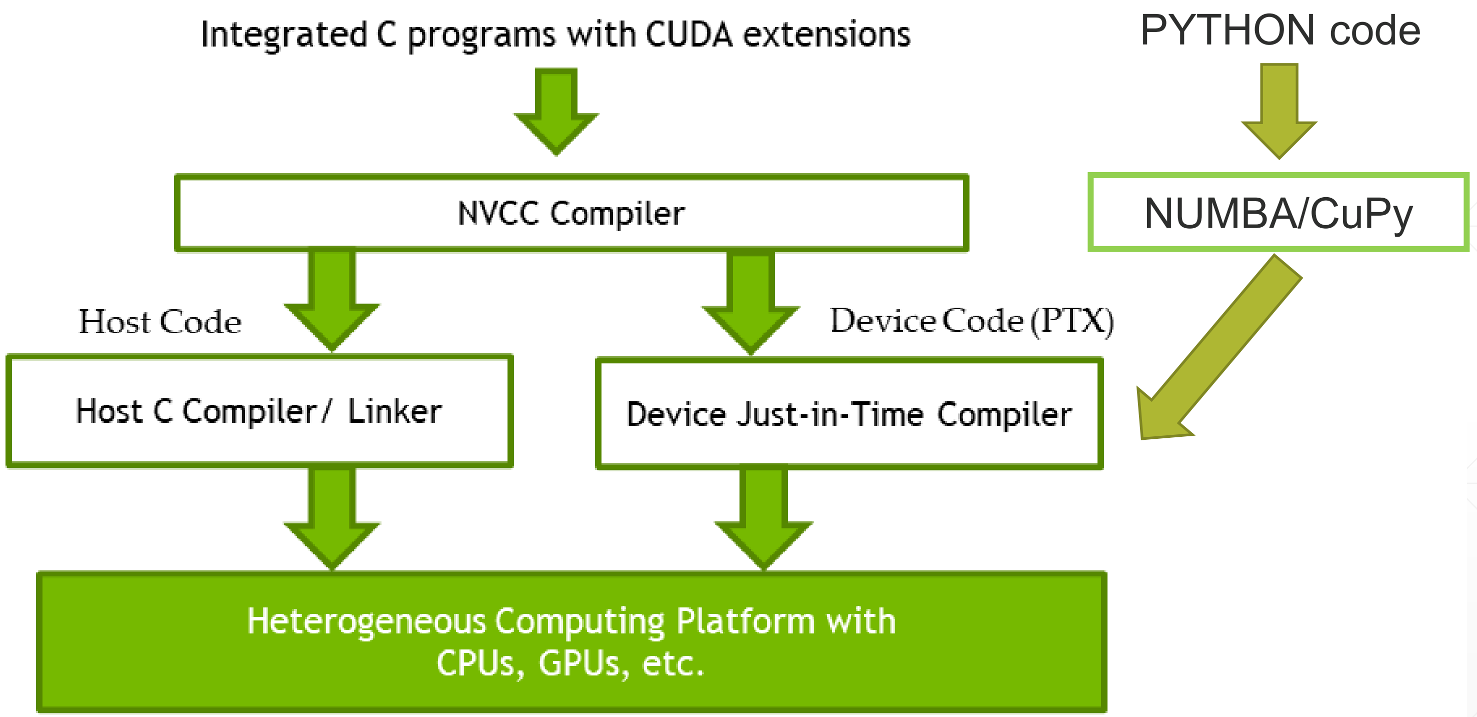
CuPy

- CuPy is an open-source array library for GPU-accelerated computing with Python. CuPy utilizes CUDA Toolkit libraries including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL to make full use of the GPU architecture.
- CuPy's interface is highly compatible with NumPy and SciPy; in most cases it can be used as a drop-in replacement. All you need to do is just replace numpy and scipy with cupy and cupyx.scipy in your Python code.
- You can easily make a custom CUDA kernel if you want to make your code run faster, requiring only a small code snippet of C++. CuPy automatically wraps and compiles it to make a CUDA binary. Compiled binaries are cached and reused in subsequent runs. Please read the User-Defined Kernels tutorial.
  And, you can also use raw CUDA kernels via Raw modules.



```
>>> x = cp.arange(6, dtype='f').reshape(2, 3)
>>> y = cp.arange(3, dtype='f')
>>> kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y', 'float32 z',
...     '''
...     if (x - 2 > y) {
...       z = x * y;
...     } else {
...       z = x + y;
...     }
...     ''', 'my_kernel')
>>> kernel(x, y)
array([[ 0.,  2.,  4.],
       [ 0.,  4.,  10.]], dtype=float32)
```

Compiling A CUDA Program


Integrated C programs with CUDA extensions
PYTHON code
NVCC Compiler
NUMBA/CuPy
Host Code
Device Code (PTX)
Host C Compiler / Linker
Device Just-in-Time Compiler
Heterogeneous Computing Platform with CPUs, GPUs, etc.

HOW-TO run Python with CUDA?

OPTIONS:

- Install Python interpreter + CUDA locally.
- Without installation, you can use Python in the CLOUD
  - For GOOGLE COLAB start here: https://colab.research.google.com/notebooks/intro.ipynb
- Other options in the Cloud: https://developer.nvidia.com/how-to-cuda-python
- Many other options and good developers’ tools are available.

And NOW …

- Take a 10 min break …
- Then start the 1st Exercise: “CUDA Programming Model”