CUDA Basics

Ulrich Finkler – Wei Zhang

CSCI-GA.3033-020 HPML

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, Python, Cpython

Libraries Performance

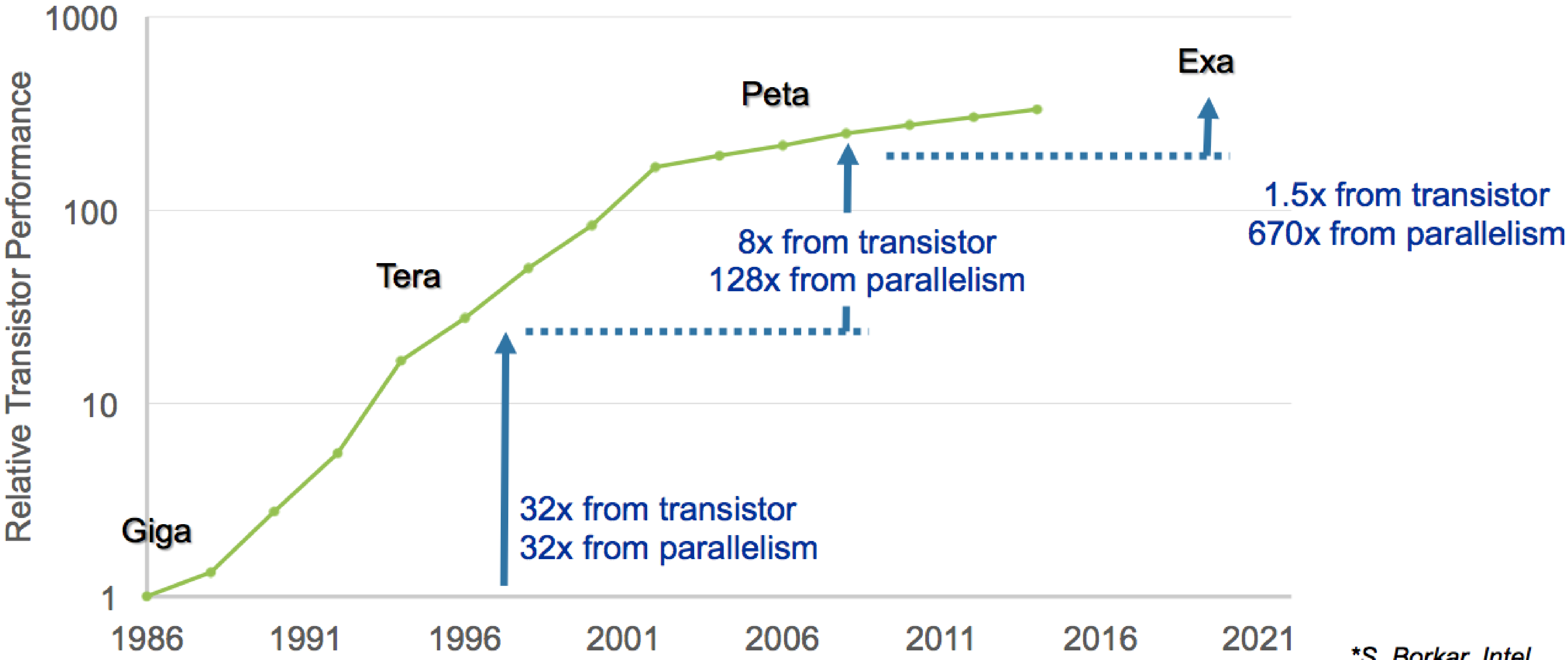
- **CUDA,** cuDNN, Communication Libraries (MPI, GLOO)

Hardware Performance

- CPU, DRAM, **GPU, HBM,** Tensor Units, Disk/Filesystem, Network

Heterogenous computing motivations

Heterogeneous Computing Motivation

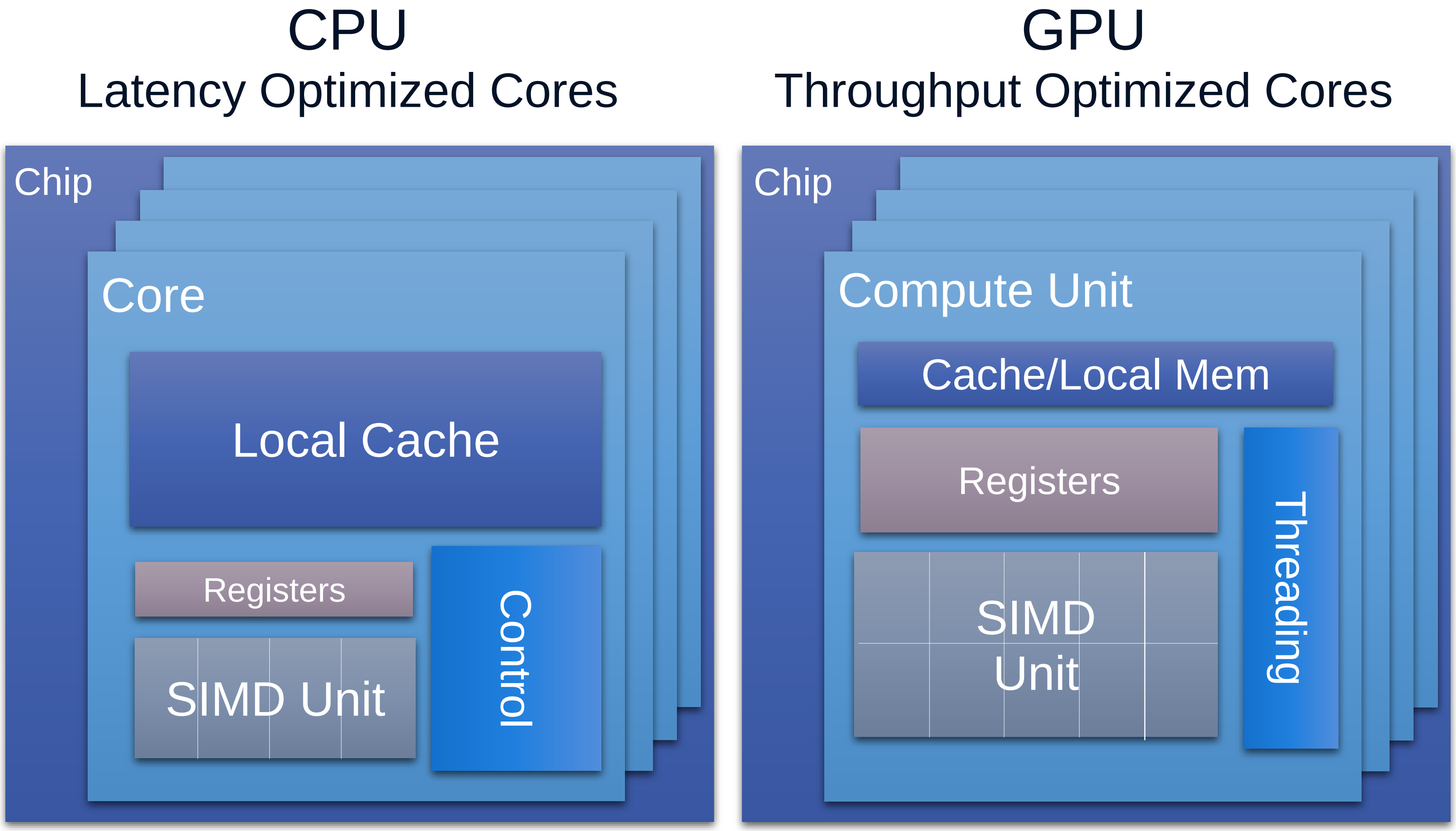


**S. Borkar, Intel*

Difference between CPU and GPU

- Latency Optimized:
  - Optimized to compute operation in a minimum time

- Throughput Optimized:
  - Optimized to compute many operations on the same time



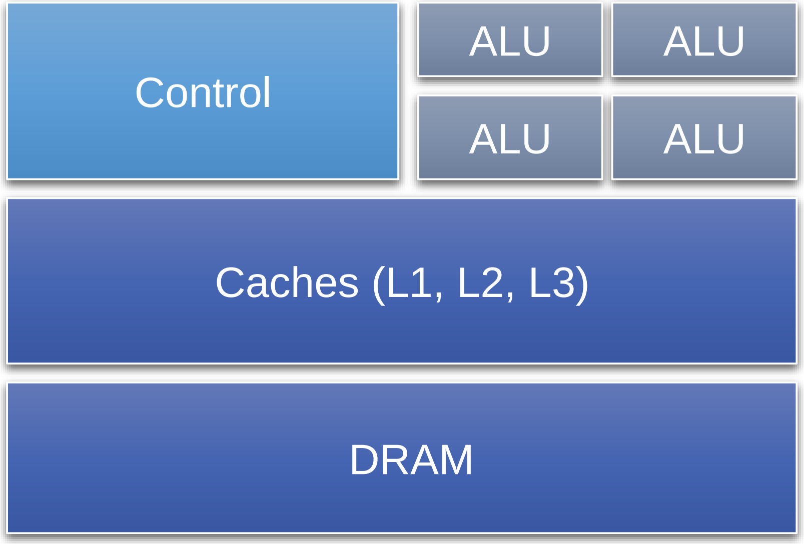
From: Nvidia – U Illinois

CPU: Latency Optimized

- Powerful **Arithmetic Logic Unit**
  - Optimized for latency
- Sophisticated **Control logic**
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Large **L1, L2, L3 Cache** Hierarchy
  - Convert long latency memory accesses to short latency cache accesses

- https://cacm.acm.org/magazines/2010/11/100622-understanding-throughput-oriented-architectures/fulltext

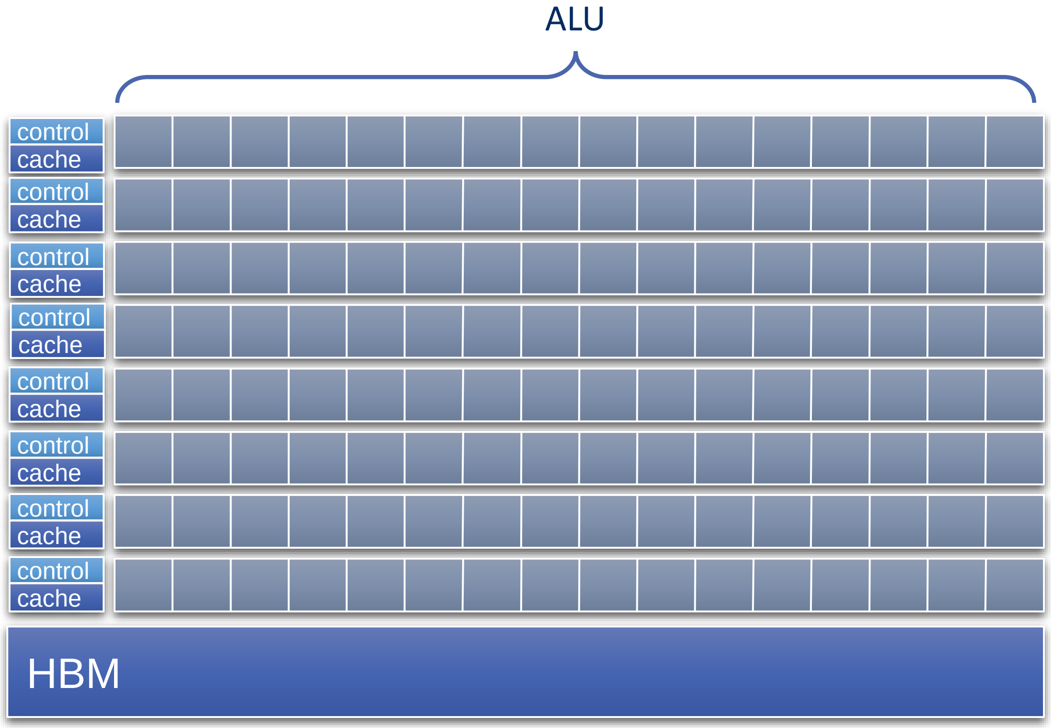
CPU Architecture



GPU: Throughput Optimized

- Many small **ALUs**
  - Many
  - Long latency
  - Parallelism
- Small **Caches (shared memory)**
  - To boost memory throughput
- Simple **Control**
  - No branch prediction
  - No data forwarding
  - Require massive number of threads to tolerate latencies
  - Threading logic and state

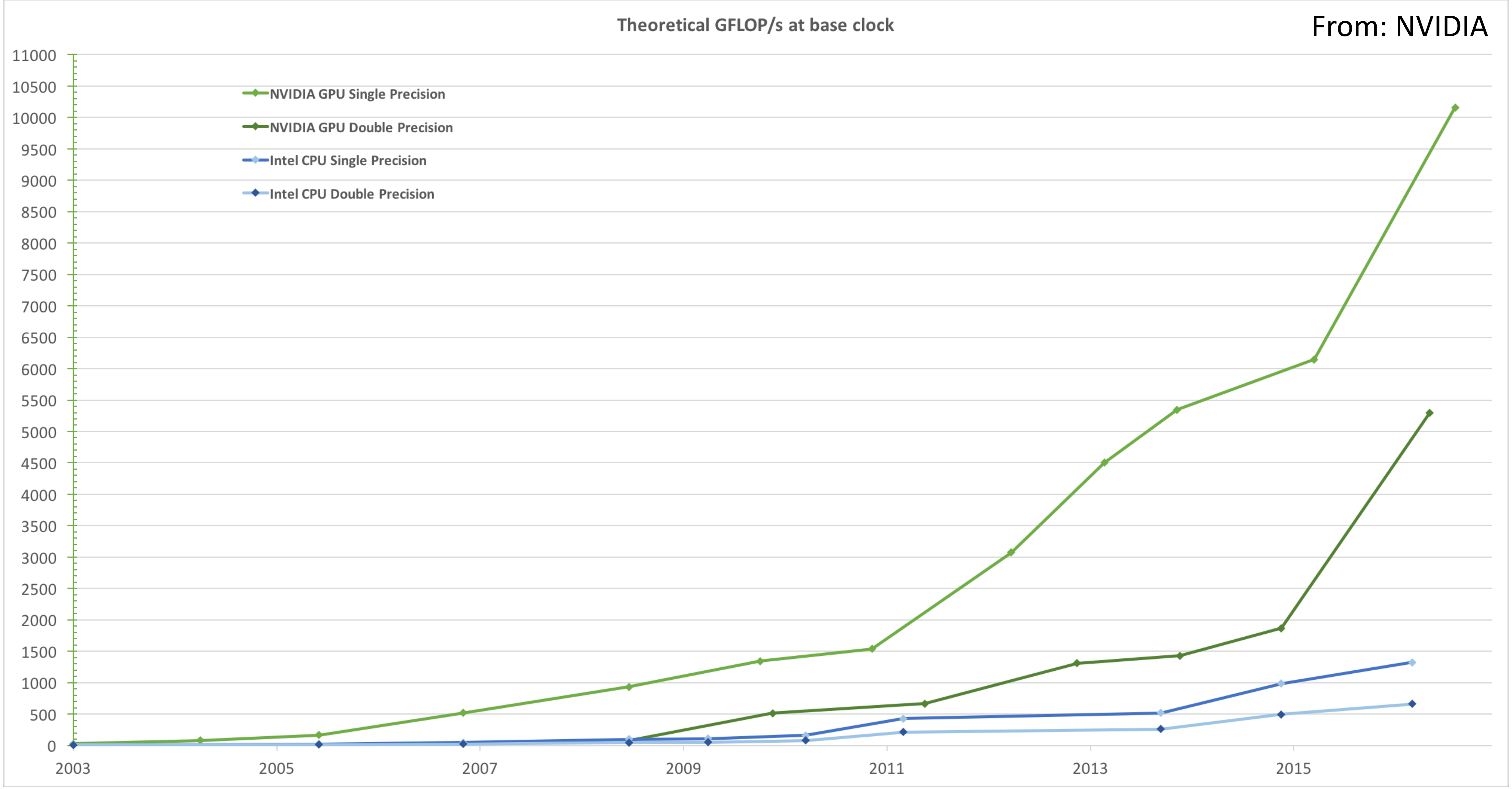
GPU Architecture



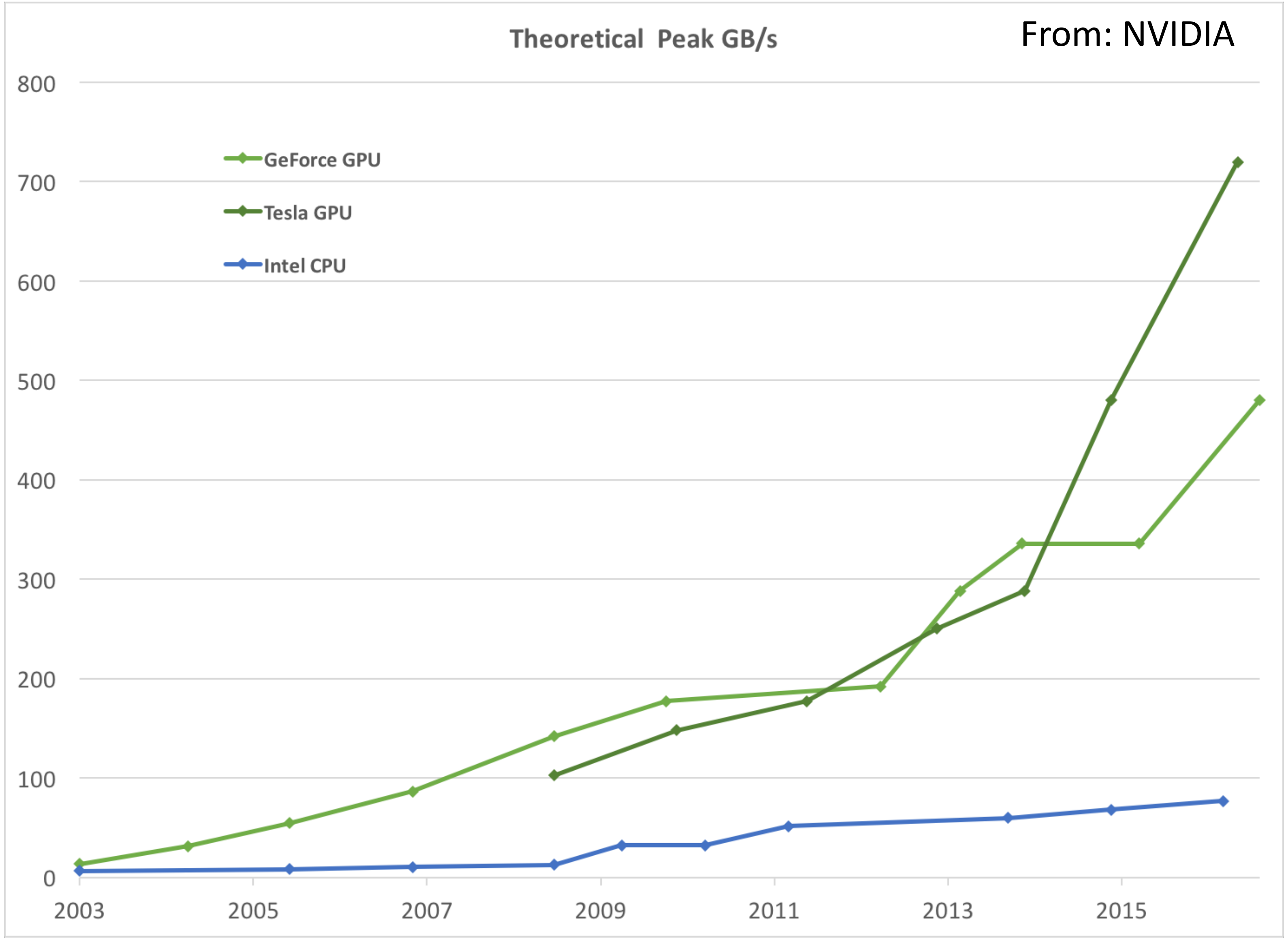
CPU/GPU Performance Comparison

- Sequential Code:
  - CPU about 10x faster than GPU
- Parallel Code:
  - GPUs can be 10X+ faster than CPUs for parallel code
- Latencies:
  - CPU: latency of operations is in **nsec**
  - GPU: launching a kernel can take 10s **micro-sec** or more

GPU Performance Advantage 1: FLOPS



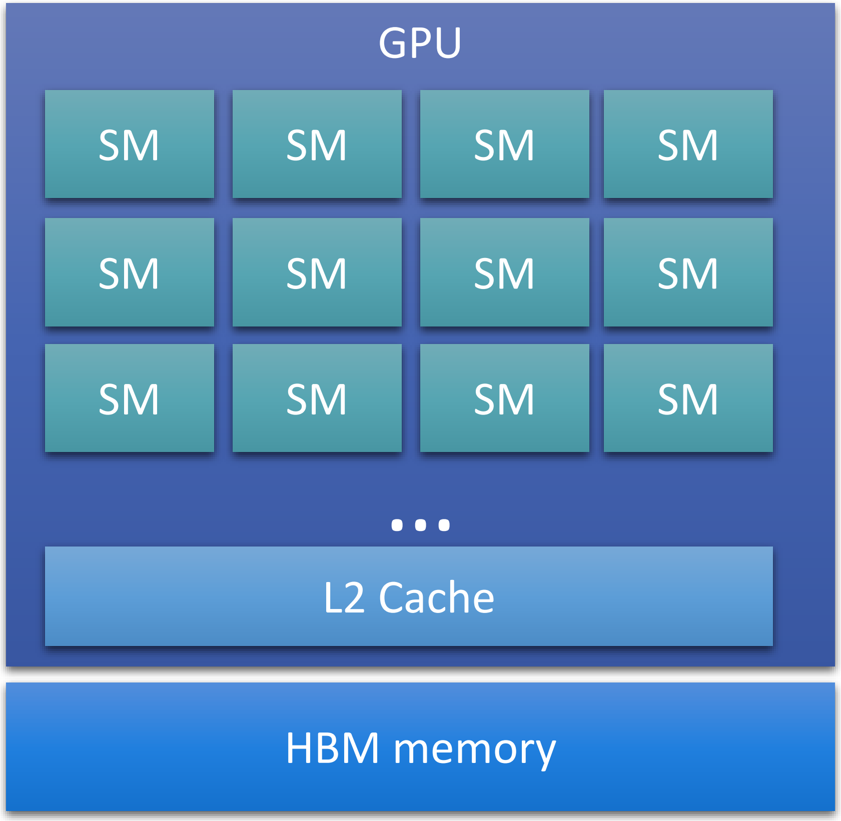
GPU Performance Advantage 2: Memory BW



NVIDIA GPUs and CUDA

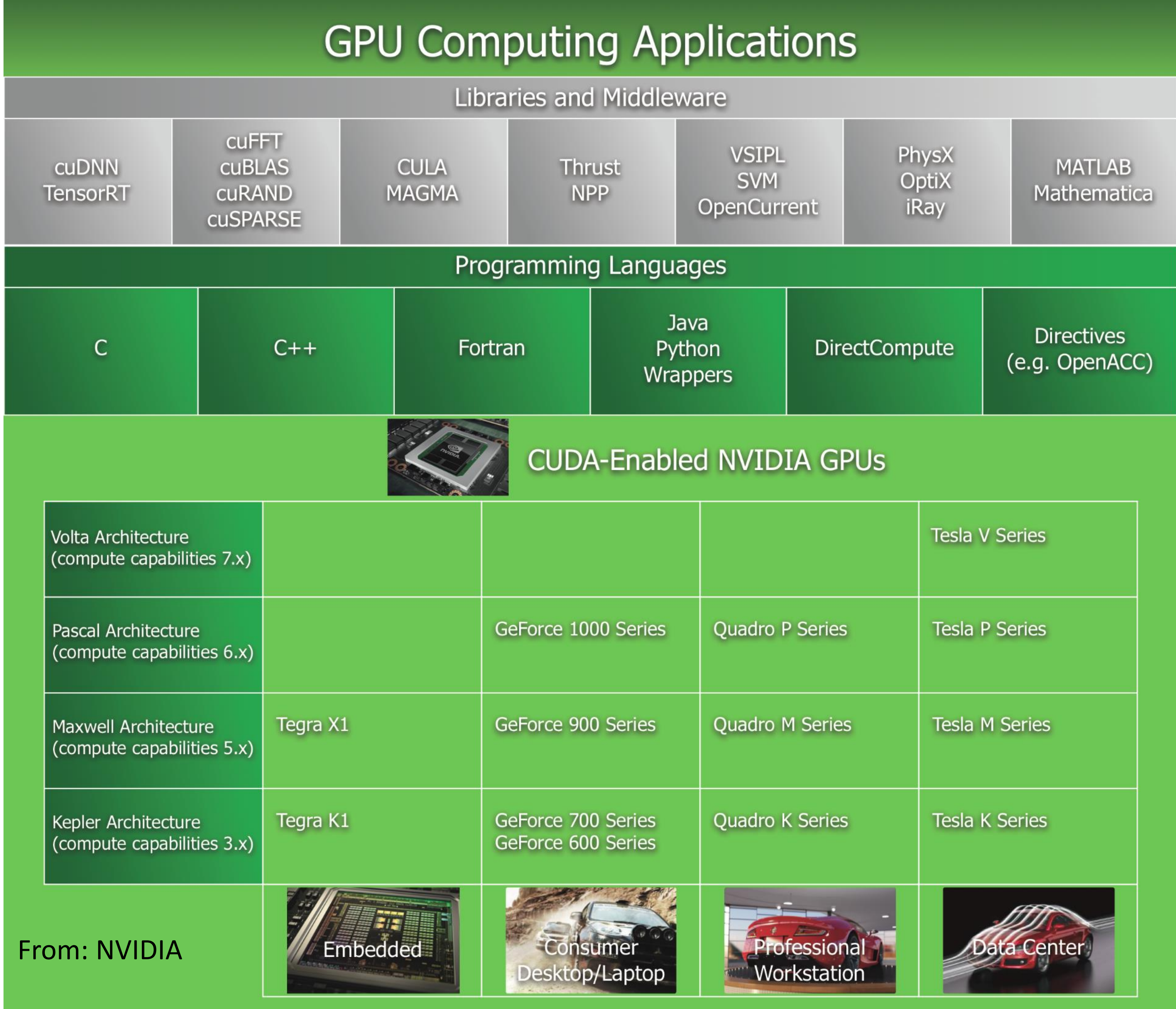
NVIDIA Volta Architecture (Tesla V100)

- Streaming Multiprocessors
    - 32 hardware threads DP
    - or 64 hardware threads SP
- DP FLOPS: 7,000 GFLOPS
- L2 size: 6MB
- High Bandwidth Memory
    - Size: 16 GB
    - Bandwidth: 900 GB/s



CUDA

- CUDA®: *A General-Purpose Parallel Computing Platform and Programming Model*
- Key objectives:
  - **Scalability**
  - **Portability**
- CUDA has different compute capabilities for each architecture
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction



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
Volta Architecture (compute capabilities 7.x)				Tesla V Series
Pascal Architecture (compute capabilities 6.x)		GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series

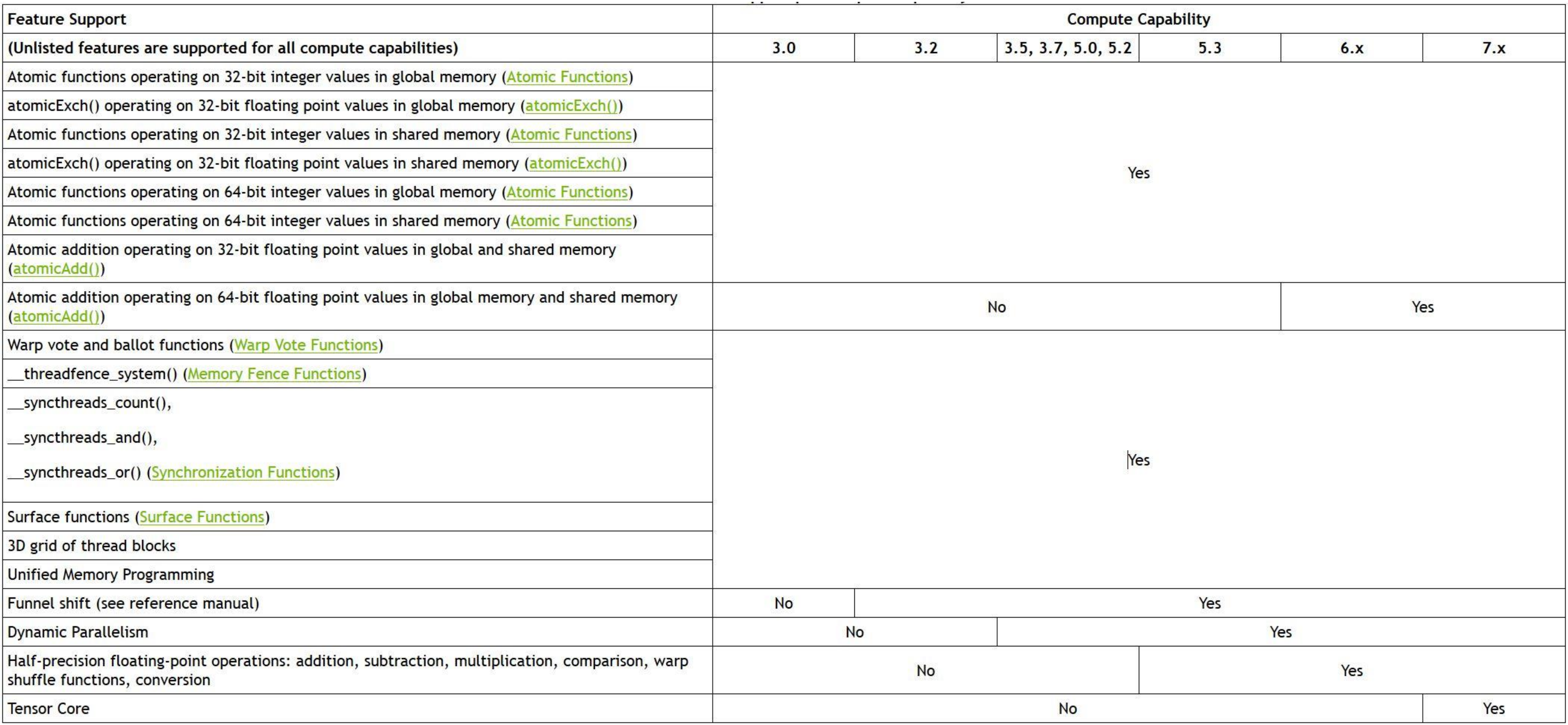
From: NVIDIA

CUDA – Compute Capability

- Compute Capability:
  - represented by a version number (ex. 6.2)
    - Major revision number (ex. 6) identifies the core architecture
    - Minor revision number identifies incremental improvements (ex. .2)
  - Identifies the features supported by the GPU hardware
  - Used by applications at runtime to identify available features
- Do not confuse with CUDA version
  - Tesla and Fermi not supported on CUDA 7.0 and 9.0 respectively

Compute Capability Major revision	NVIDIA GPU Architecture
1	Tesla
2	Fermi
3	Kepler
5	Maxwell
6	Pascal
7	Volta

NVIDIA Hardware and Compute Capabilities



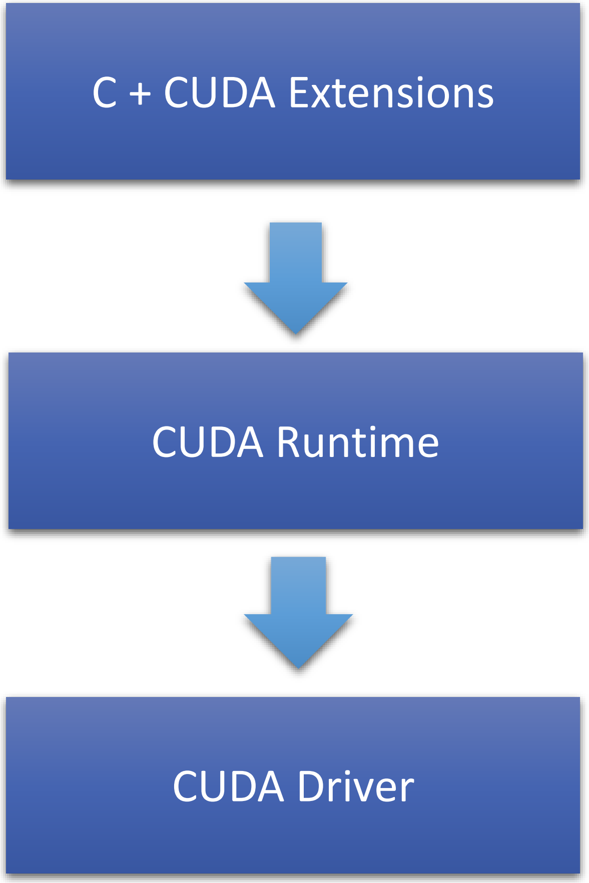
Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)						
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)						
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())						
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No				Yes	
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming						
Funnel shift (see reference manual)	No	Yes				
Dynamic Parallelism	No		Yes			
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No			Yes		
Tensor Core	No					Yes

From: NVIDIA

CUDA Compilation and Runtime

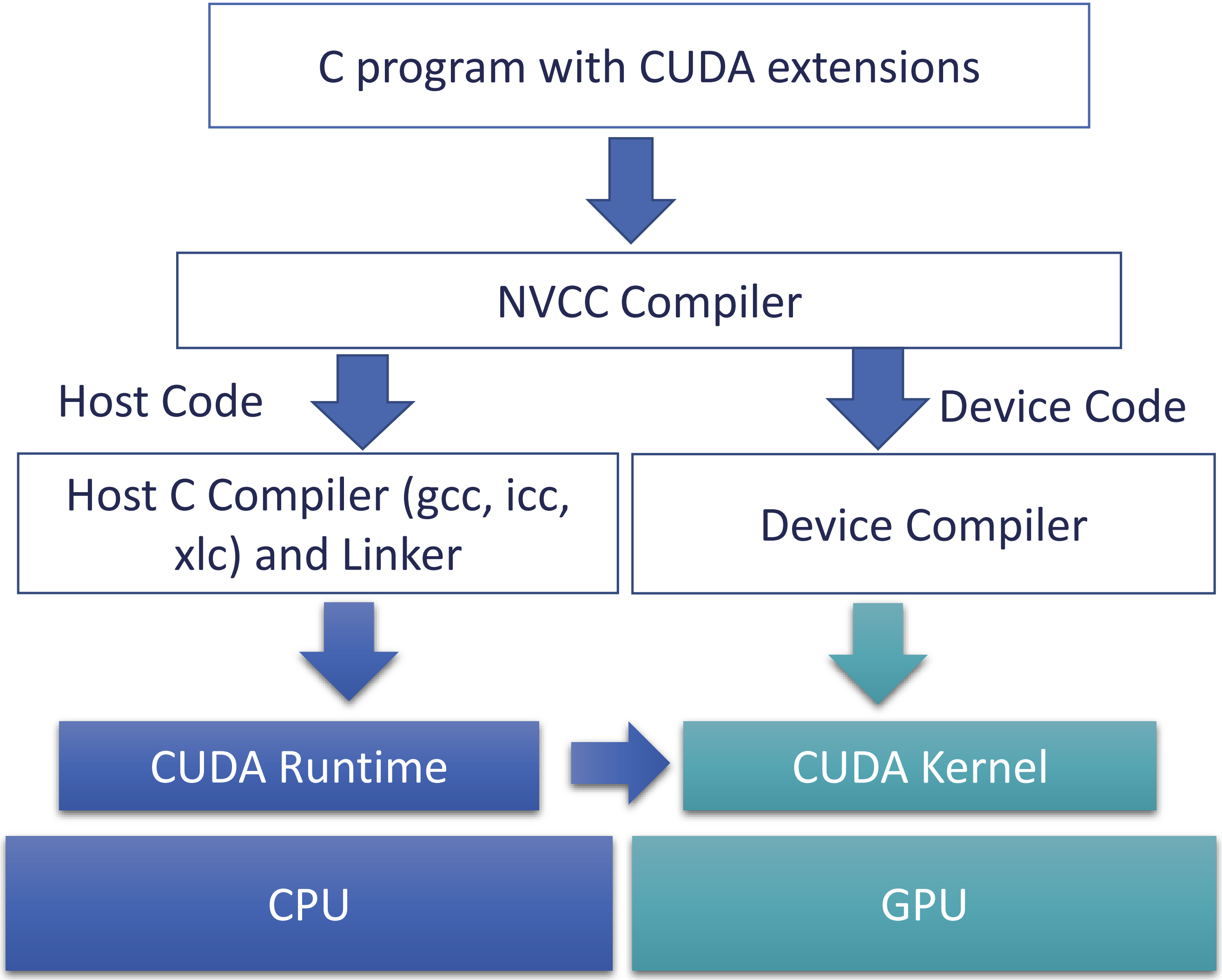
CUDA C Extension Runtime and Driver

- There are three layers for a CUDA program:
  1. CUDA C extension:
     - Developers write code using this extension
     - Ex. The <<<>>> kernel construct or the threadIdx variable
  2. CUDA Runtime API
     - NVCC compiler produces code that calls the runtime
     - Functions preceded by *cudaXXX*
     - Library: *cudart.dll or cudart.so*
  - 3 CUDA Driver API
     - The runtime uses the CUDA driver APO
     - Library: *cuda.dll or coda.so*
     - Functions preceded by cuXXX
     - Uses PTX code to execute on the GPU



CUDA Compilation and Runtime

- NVCC compiler divides code in two parts:
  - Host C/C++ compiler
  - Device compiler
- Host C/C++ code is passed to the host compiler
- Host C code:
  - Contains calls to **CUDA runtime** and pass the **kernel code** as argument
- Kernel code is executed on the GPU



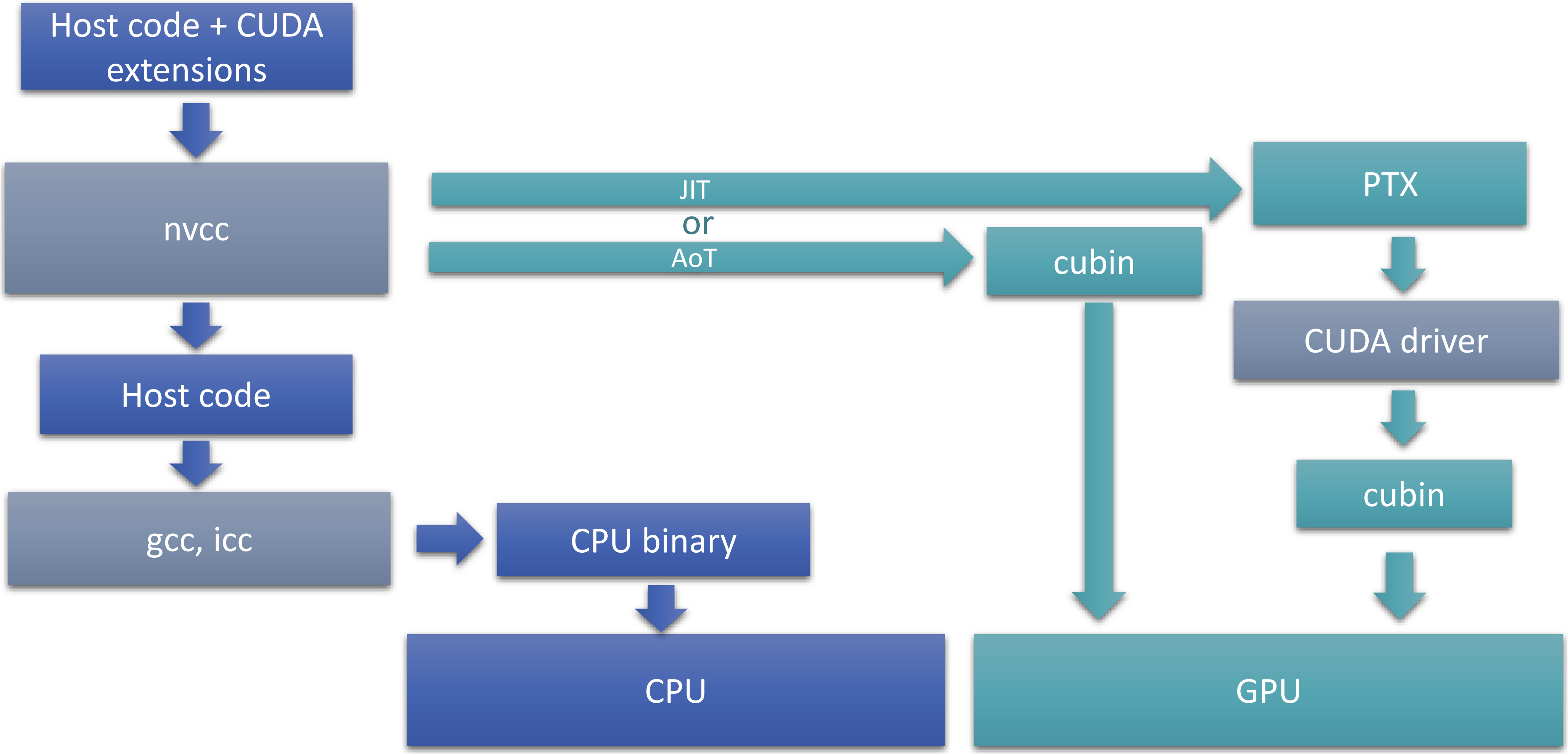
CUDA Just-in-time compilation

- Two approaches to compiling CUDA
- CUDA Ahead of Time/Offline Compilation:
  - Compilation flag code (example: --code=sm_60 for comp. cap. 6.0)
  - Directly compile into CUDA binary: **cubin**
- CUDA JIT:
  - Compilation flag arch (example: --arch=compute_60 for comp. cap. 6.0)

  1. Compile kernels in **PTX** (CUDA assembly)
  2. PTX code is compiled by the **CUDA Driver**

What is the advantage of JIT for CUDA applications like games?

CUDA Compilation Schema

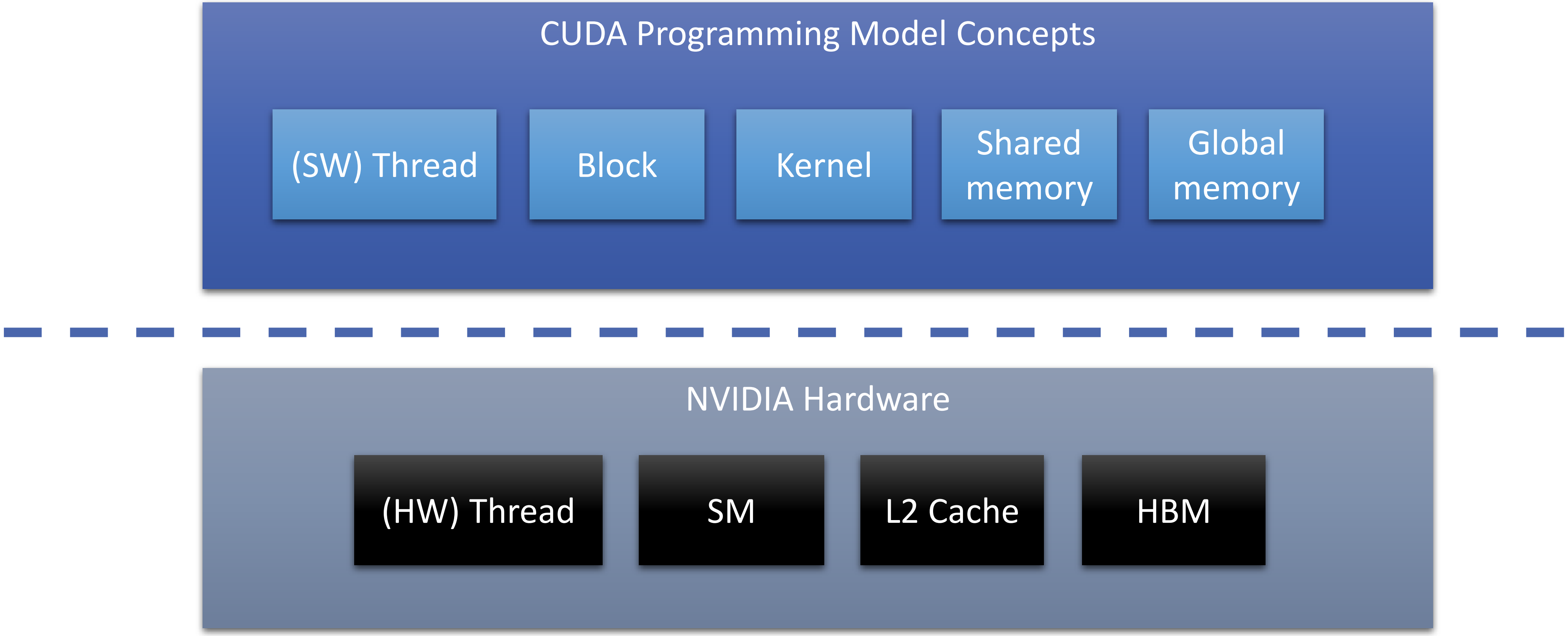


http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc

CUDA Programming Model

CUDA Programming model

- Programming model tries to **abstract** hardware architecture



CUDA Programming Model - Definitions

- ***Kernel:***
  - C function that will execute in parallel on *N* hardware threads in the GPU
- ***(Software) Thread***:
  - Smallest unit of execution
  - ***threadIdx****:* 1,2, or 3 dimensions vector (x,y,z)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

CUDA functions definition

	Executed on:	**Callable from:**
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- Keywords for CUDA functions:
  - **__global__** defines a kernel function
    - always returns **void**
  - **__device__** and **__host__** can be used together
  - **__host__** is optional if used alone

CUDA – Hello World

- Build instructions:
    - C code: main.c
    - CUDA code: **main.cu**
        - **nvcc** only parses .cu (or use –x)
    - First make sure to have nvcc and cuda libraries (module load…)

```
$ nvcc ./main.cu –o main
$ ./main
```

- C code

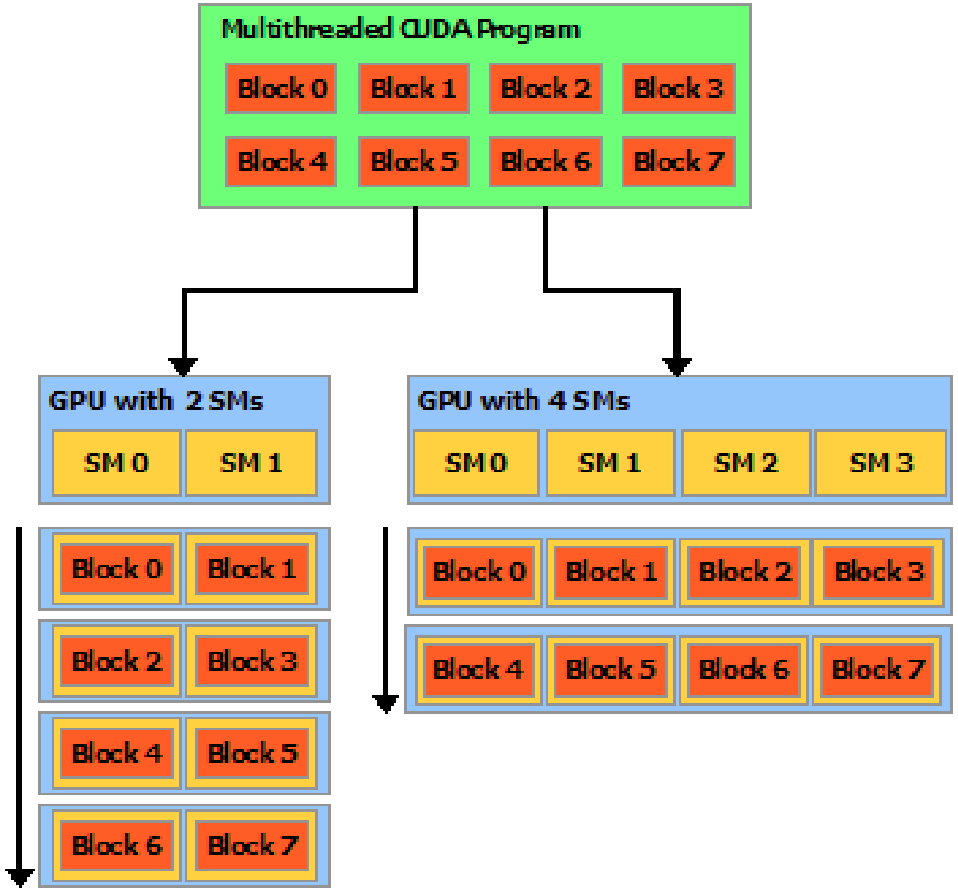
```
intmain() {
printf("Hello World!\n");
return0;
}
```

- CUDA code

```
__global__ void mykernel(void) {
printf("Hello World!\n");
}
intmain(void) {
mykernel<<<1,1>>>();
return 0;
}
```

CUDA Programming Model - Definitions

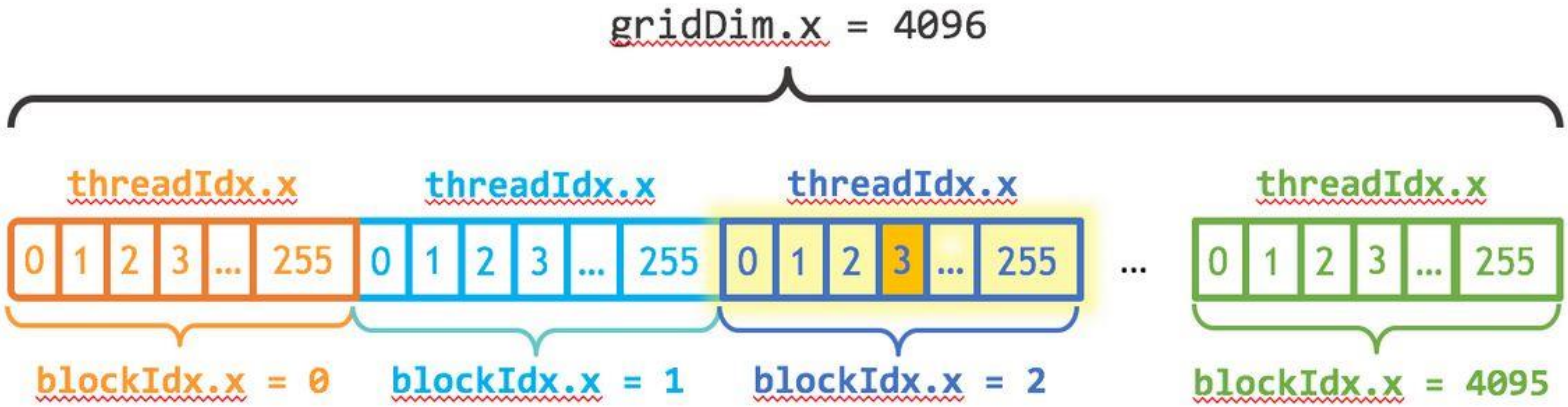
- ***Grid:***
  - *A group* of blocks executing a **kernel**
- ***Block***:
  - A group of threads that can be scheduled independently on a SM
  - Max threads in a block: 1024
  - Blocks can be executed in any order by the SMs
- ***Thread:*** a single context of execution



From: NVIDIA

CUDA – Grid Block and Thread

- One Grid per CUDA Kernel
- Multiple Blocks per Grid
- Multiple Threads per Block

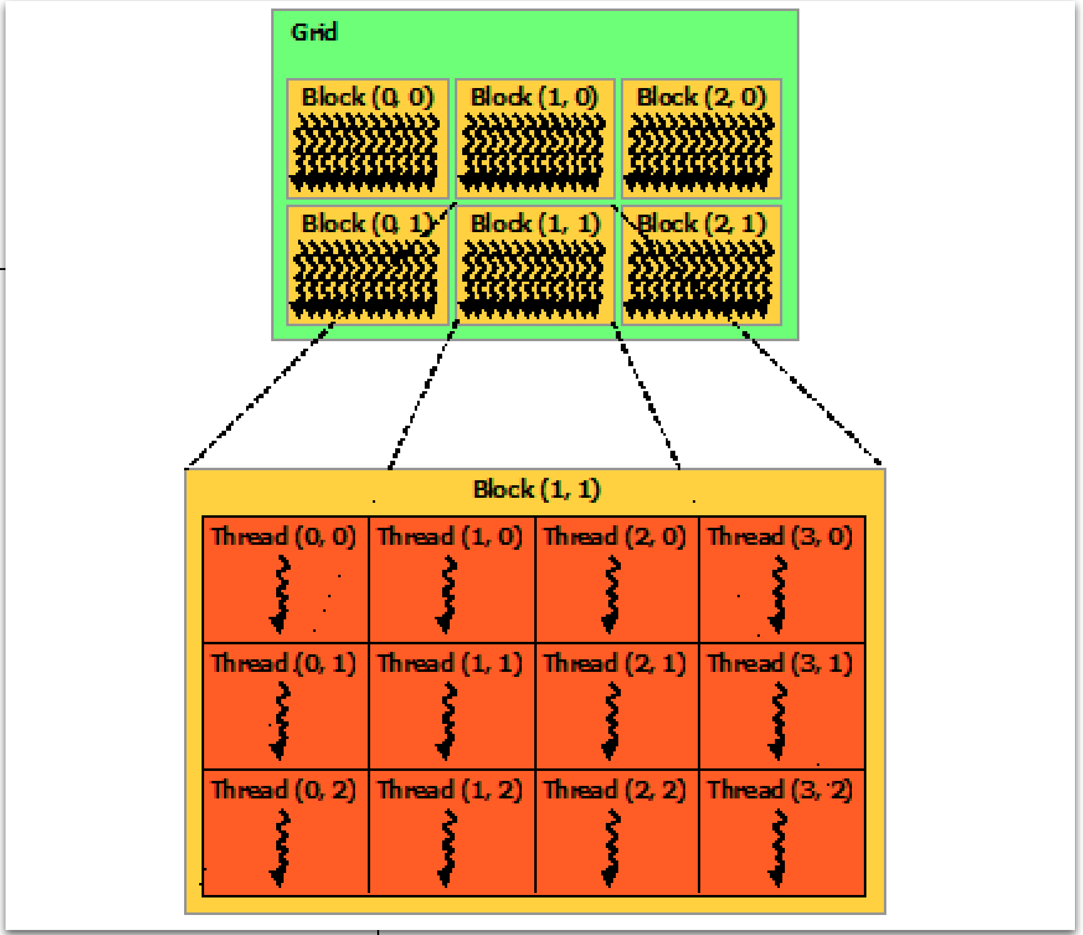


From: NVIDIA

CUDA - Example

- Block and Threads index

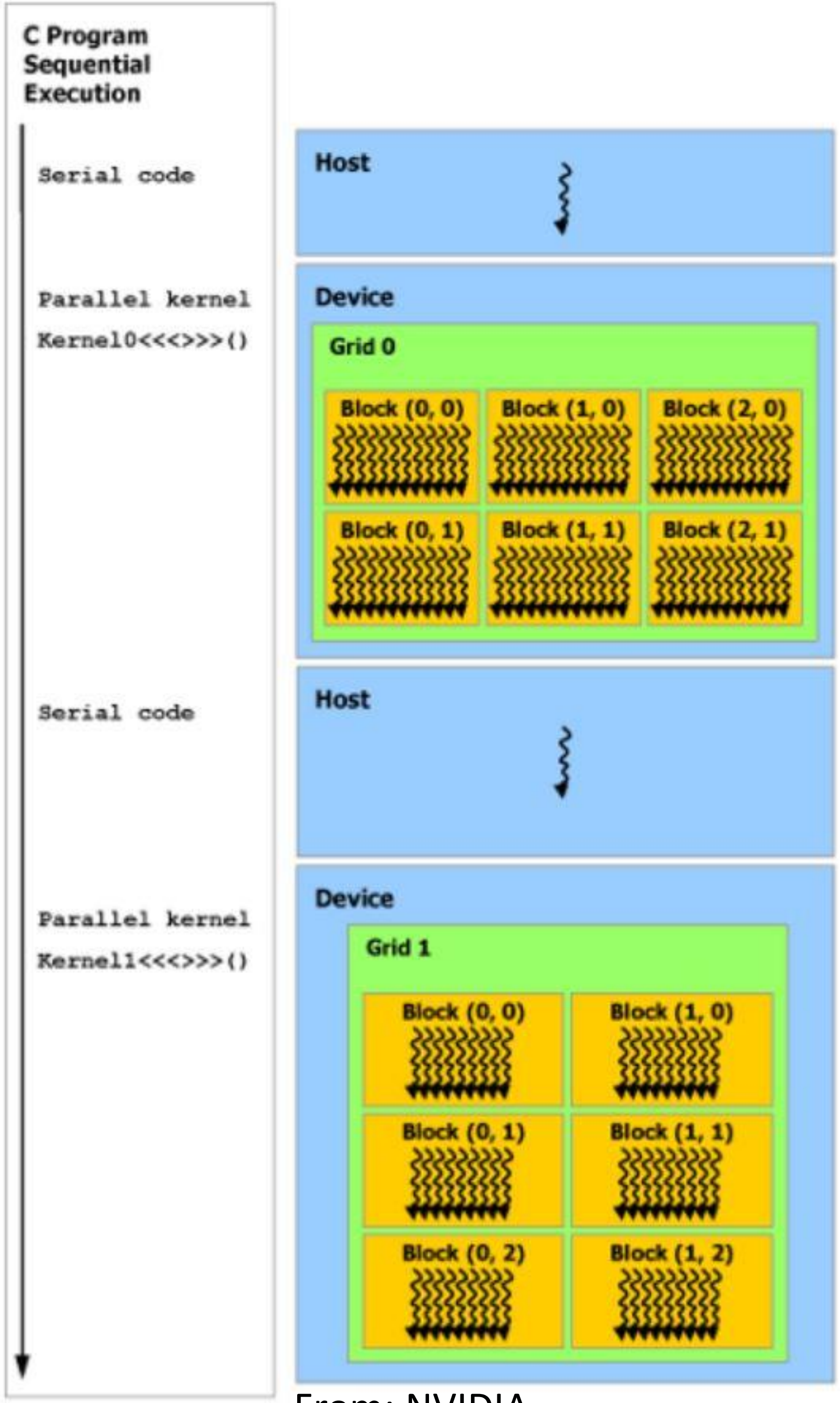
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



From: NVIDIA

CUDA – Serial and Parallel code

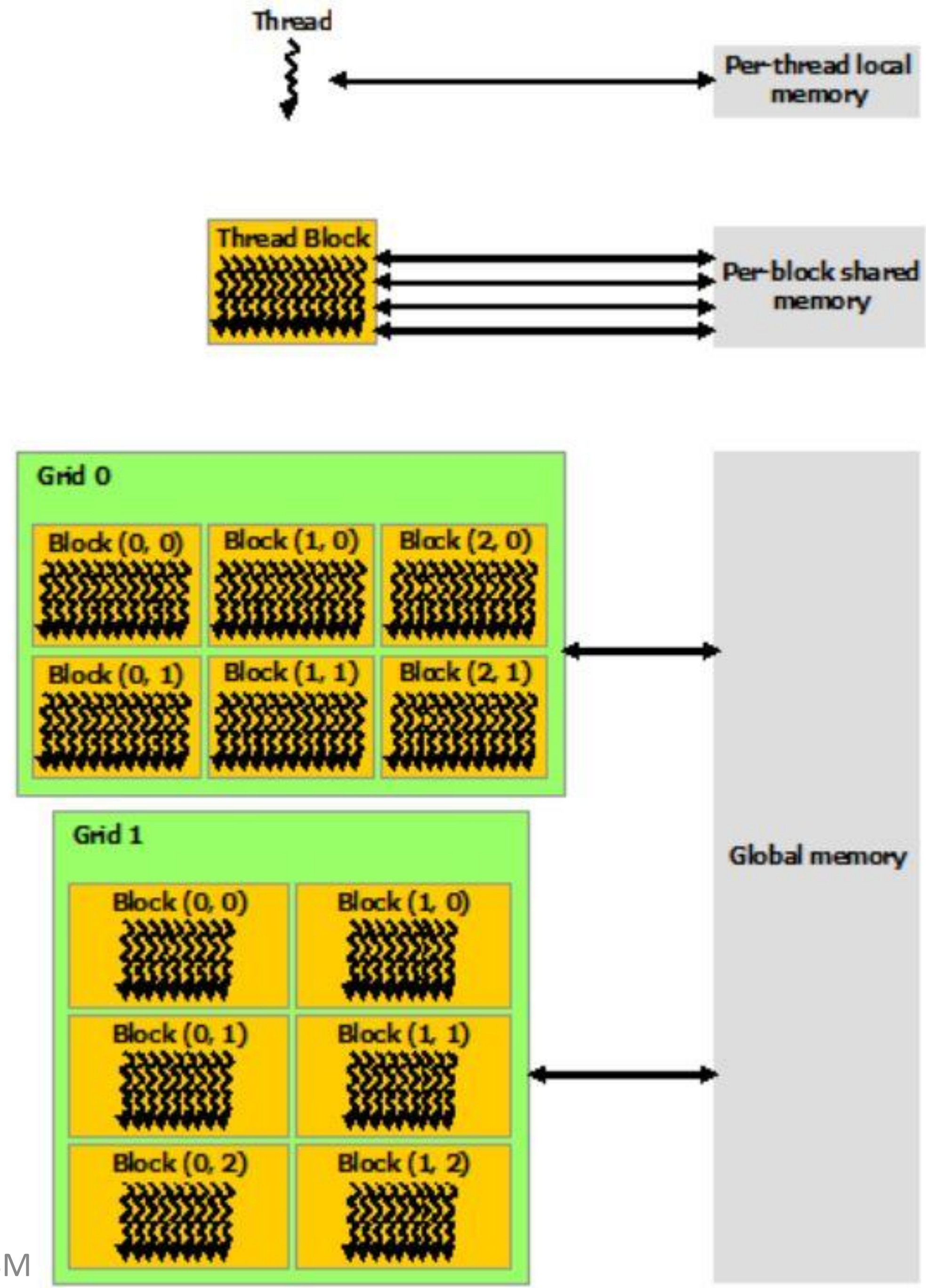
- A CUDA program starts as a sequential C/C++/Fortran process in the Host
- With a kernel launch:
  - Parallel code is executed inside the device (GPU)
  - SIMT Parallelism
    - Single Instruction Multiple Threads
- At the exit of a kernel launch the becomes sequential again



From: NVIDIA

CUDA Memory Hierarchy

- There are 3 levels of CUDA memory
- Per-thread local memory:
  - Available only the single thread
- Block-shared memory:
  - Shared by all the threads in a block
- Global memory:
  - Shared among all blocks and all grids



From: NVIDIA

C++ Example

- Sum two arrays in C++
- C++ code:
  - Allocate arrays
  - Add elements of two arrays
  - Check for errors

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

```
int main(void) {
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

// Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

From: NVIDIA

CUDA Example

- Sum two arrays with C++ and CUDA

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride) // bit funny, why?
    y[i] = x[i] + y[i];
}
int main(void) {
  int N = 1<<20;
  size_t size = N*sizeof(float);
  float *x, *y;

  // Allocate input vectors h_A and h_B in host memory
  float* hx = (float*)malloc(size);
  float* hy = (float*)malloc(size);

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
   hx[i] = 1.0f;
   hy[i] = 2.0f;
  }
```

```
  // Allocate vectors in device memory
  cudaMalloc(&x, size);
  cudaMalloc(&y, size);
  // Copy vectors from host memory to device global memory
  cudaMemcpy(x, hx, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y, hy, size, cudaMemcpyHostToDevice);

  // Run kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);
  cudaMemcpy(hy, y, size, cudaMemcpyDeviceToHost);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(hy[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);  cudaFree(y);
  free(hx); free(hy);
  return 0;
}
```

From: NVIDIA

CUDA Vector Add – Vector size exceeds grid

```
#include <iostream>
#include <math.h>

__global__ void vector_add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride)  // Needed !!!
    y[i] = x[i] + y[i];
}
int main(void) {
  int N = 1000;
  size_t size = N*sizeof(float);
  float* hx = (float*)malloc(size);      // cpu buffers
  float* hy = (float*)malloc(size);
  for (int i = 0; i < N; i++) {
    hx[i] = 1.0f;    hy[i] = 2.0f;
  }
float *x, *y;                      // gpu buffers
  cudaMalloc(&x, size);
  cudaMalloc(&y, size);
  cudaMemcpy(x, hx, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y, hy, size, cudaMemcpyHostToDevice);

 int blockSize = 8;
 int numBlocks = 8;                 // only 64 'tasks'
 vector_add<<<numBlocks, blockSize>>>(N, x, y);
 cudaMemcpy(hy, y, size, cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(hy[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  cudaFree(x);  cudaFree(y);
  free(hx); free(hy);
  return 0;
}
```

CUDA Host and Device Memory

- In normal conditions Host and Device memory do no share an address space:
  - A pointer from *malloc()* or *cudaMallocHost()* (page-locked) cannot be read from GPU
  - A pointer from *cudaMalloc()* is on the device and cannot be hosted by the host
- Implications of not having a shared address space:
  - Explicit *cudaMemcpy()*
  - Pointers cannot be used: each data-structure using pointers needs to be serialized, copied and then recomposed

CUDA Unified Virtual Memory

- Unified Virtual Memory: Defines a *managed* memory space with **single coherent global address space**
  - Pointers work on CPU and GPU => No need for explicit *cudaMemcpy()*
  - Two ways: use ***__managed__*** keyword or use ***cudaMallocManaged()***

```
__device__ __managed__  int  ret[1000];
__global__ void AplusB(int a, int b) {
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
   AplusB<<< 1, 1000 >>>(10, 100);
   cudaDeviceSynchronize();
   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);
   return  0;
}
```

```
__global__ void AplusB(int *ret, int a, int b) {
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
   int *ret;
   cudaMallocManaged(&ret, 1000 * sizeof(int));
   AplusB<<< 1, 1000 >>>(ret, 10, 100);
      cudaDeviceSynchronize();
   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);
   cudaFree(ret);
   return  0;
}
```

CUDA Example with UVM

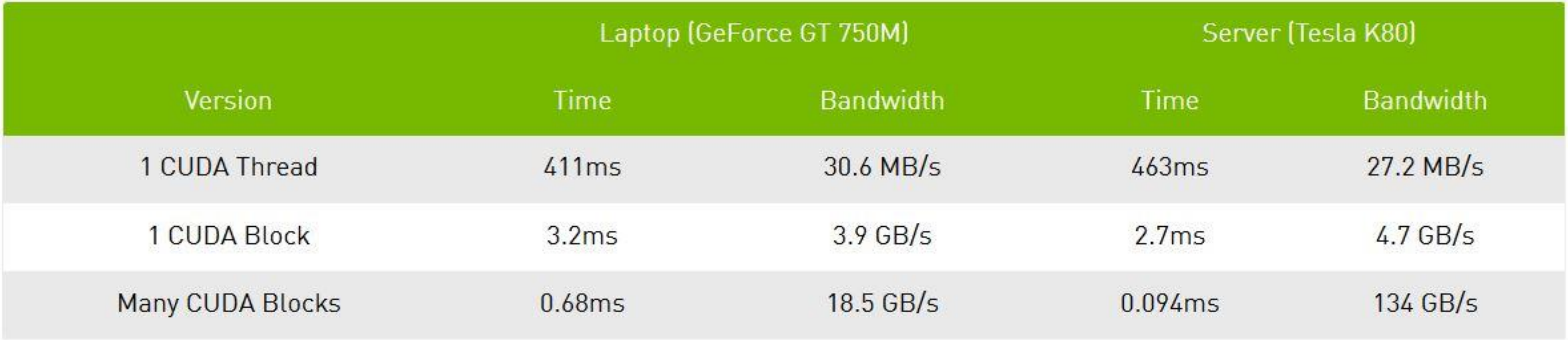
```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void) {
 int N = 1<<20;
 float *x, *y;

 // Allocate Unified Memory – accessible from CPU or GPU
 cudaMallocManaged(&x, N*sizeof(float));
 cudaMallocManaged(&y, N*sizeof(float));

 // initialize x and y arrays on the host
 for (int i = 0; i < N; i++) {
  x[i] = 1.0f;
  y[i] = 2.0f;
 }
```

```
// Run kernel on 1M elements on the GPU
 int blockSize = 256;
 int numBlocks = (N + blockSize - 1) / blockSize;
 add<<<numBlocks, blockSize>>>(N, x, y);

 // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
   maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

// Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

From: NVIDIA

CUDA Example Performance Results

- Latency and Bandwidth for the *add()* kernel that uses UVM



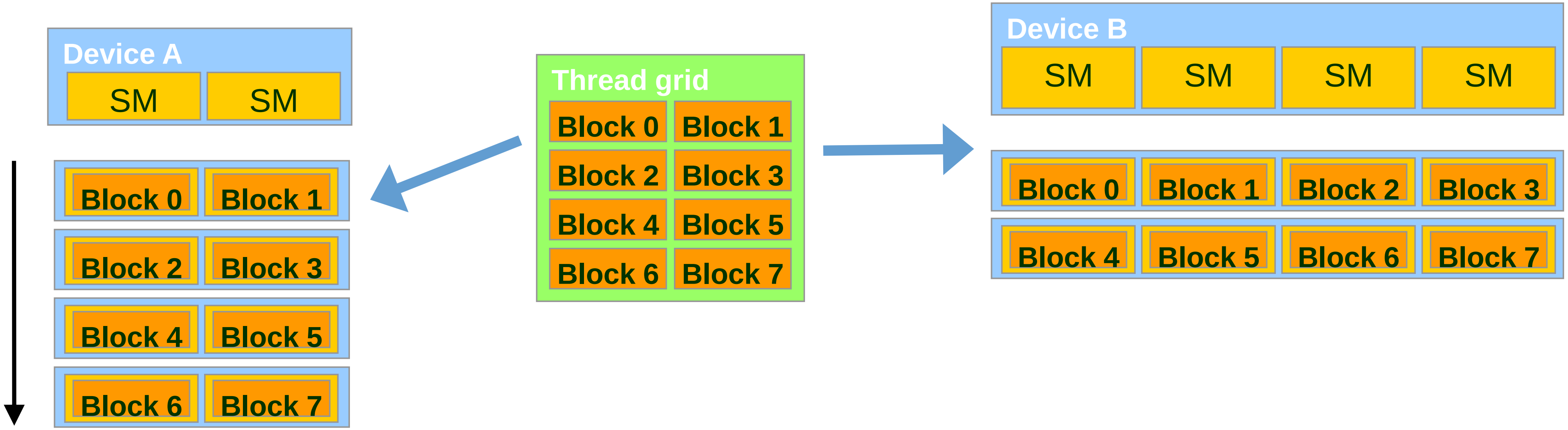
	Laptop (GeForce GT 750M)		Server (Tesla K80)	
Version	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s

From: NVIDIA

- Results from: https://devblogs.nvidia.com/even-easier-introduction-cuda/
- UVM can be much slower than explicit memcpy, or requires prefetching etc.!

CUDA Block and Warp Scheduling

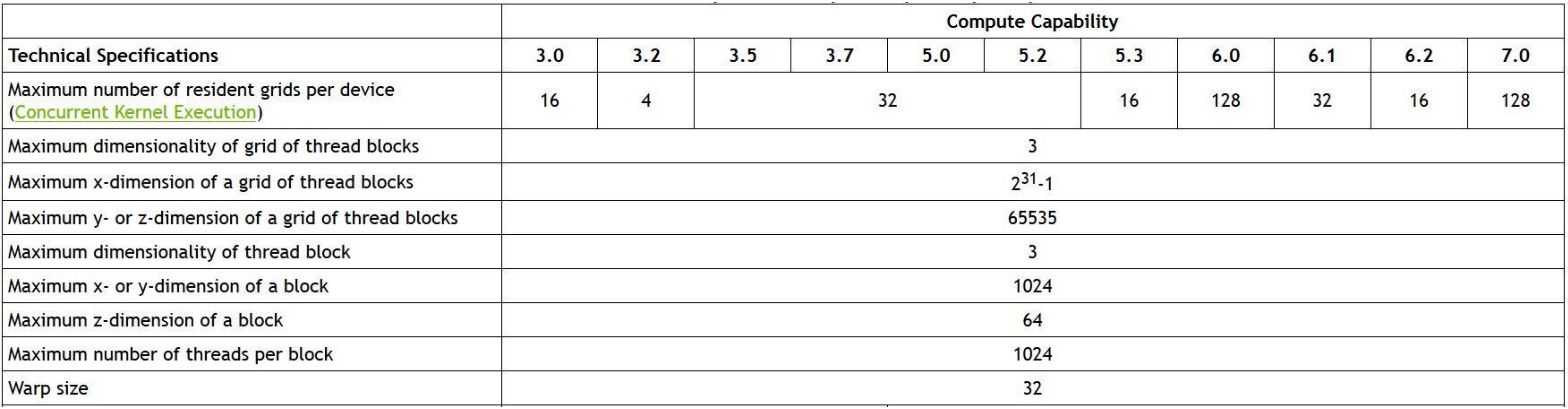
CUDA Transparent Scalability



- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of SMs

CUDA – Grids and Blocks maximum sizes

- Grid, and Block sizes have been the same for all Compute capabilities
- Newer GPUs can have more SMs



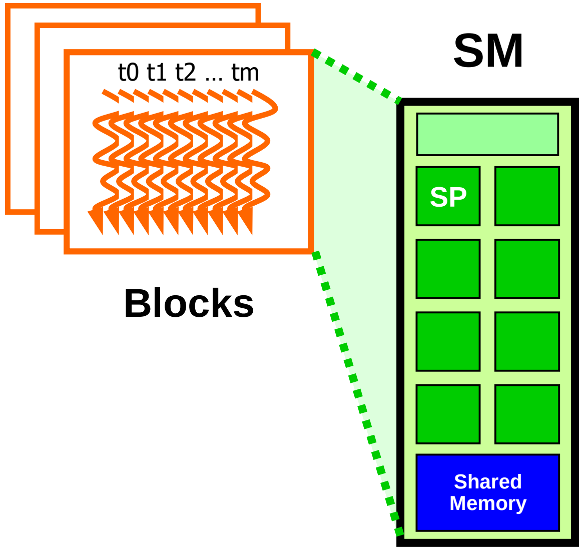
	Compute Capability										
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	2^{31}-1										
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of thread block	3										
Maximum x- or y-dimension of a block	1024										
Maximum z-dimension of a block	64										
Maximum number of threads per block	1024										
Warp size	32										

From: NVIDIA

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

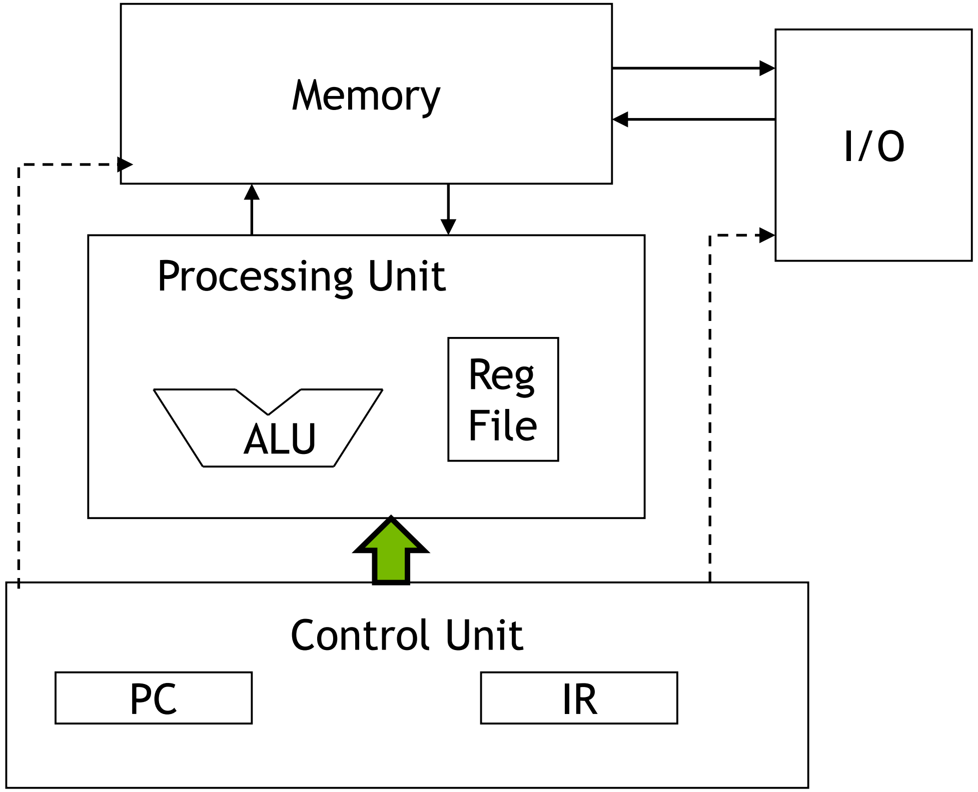
Example: Executing Thread Blocks on Fermi

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Fermi’s compute capability:
    - See: https://en.wikipedia.org/wiki/CUDA
    - Up to **8** blocks to each SM as resource allows
  - Example: A Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
  - Recent GPUs (Maxwell, Pascal, Volta)
    - resident blocks per SM: 32
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



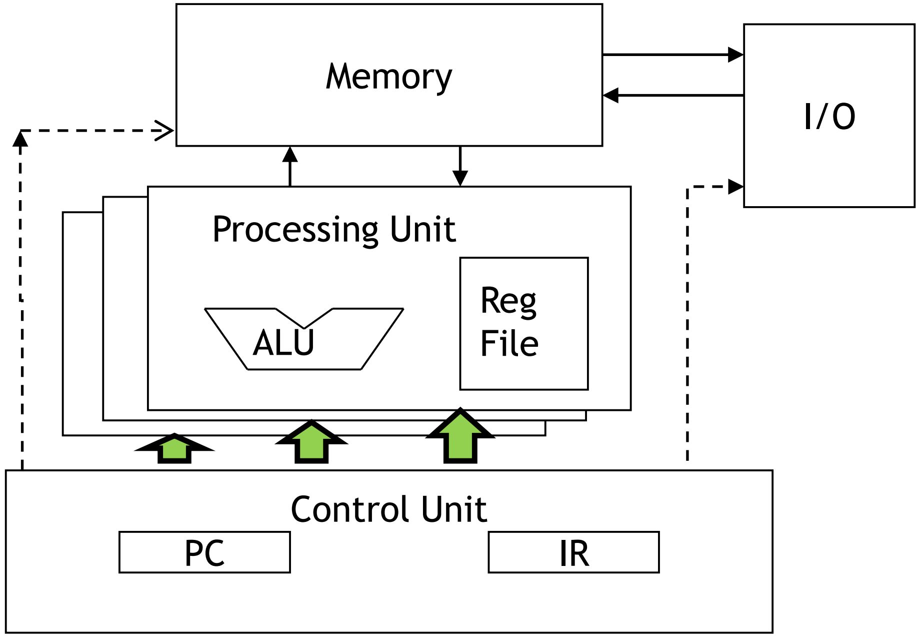
The Von-Neumann Model

- Fundamental Architecture of computing system:
  - Instruction Register
  - Program Counter
  - ALU
  - Register File
  - Memory
  - I/O



The Von-Neumann Model with SIMD units

- Single Instruction Multiple Data:
  - 1 instruction used to compute more operations:
    - Example 8/16 DP FLOP



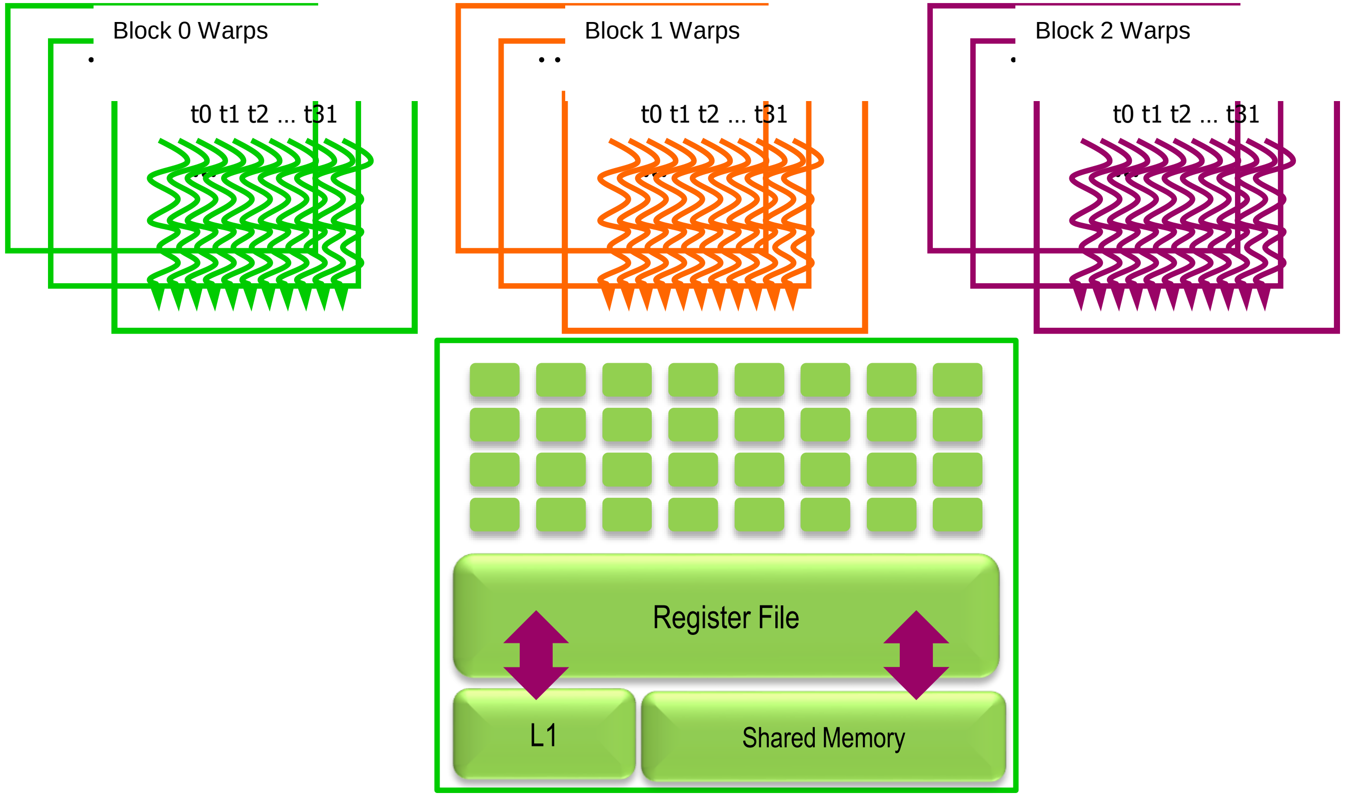
Single Instruction Multiple Data (SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

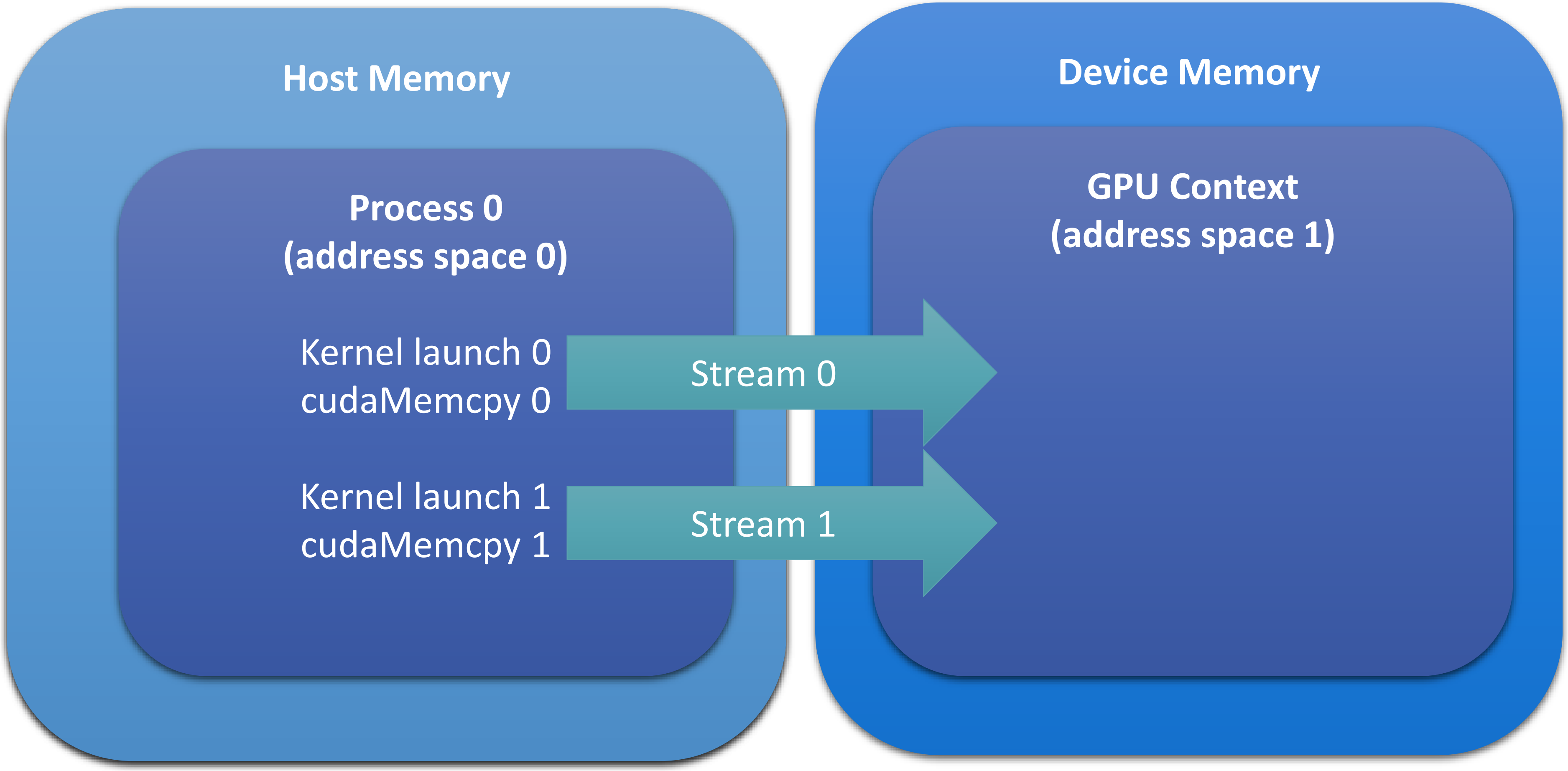
- SM implements zero-overhead warp scheduling
  - Warps whose **next instruction has its operands ready** for consumption are eligible for execution
  - Eligible Warps are selected for execution based on an optimized scheduling policy
  - All threads in a warp execute the **same instruction** when selected

SM Occupancy for performance

- Always try to achieve maximum **SM occupancy:**
  - #Active-Warps / #Max-Active-Warps
  - Approach: increase block size until it can fit the SM
- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.
- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

CUDA Context and Streams

CUDA Constext and Streams



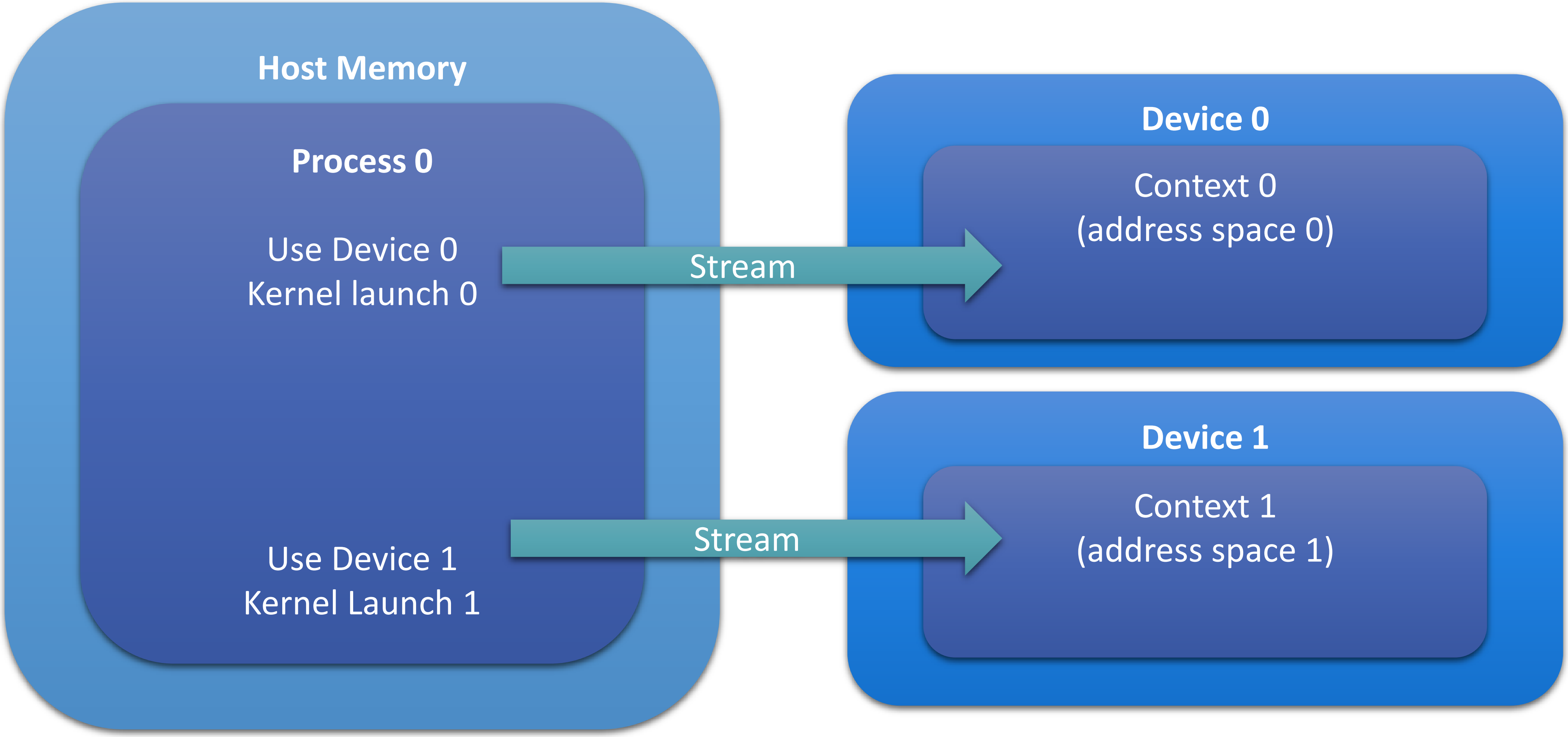
torch.cuda.Stream

- *torch.cuda.Stream()* creates a stream
  - i.e. a linear sequence of execution that belongs to a specific device
- Completion can be verified with *.query()*
- Without explicit stream default stream is used: pytorch handles all synchronization (such as *synchronize()* or *wait_stream()*) with data
- **With explicit streams: user is expected to do synchronization**

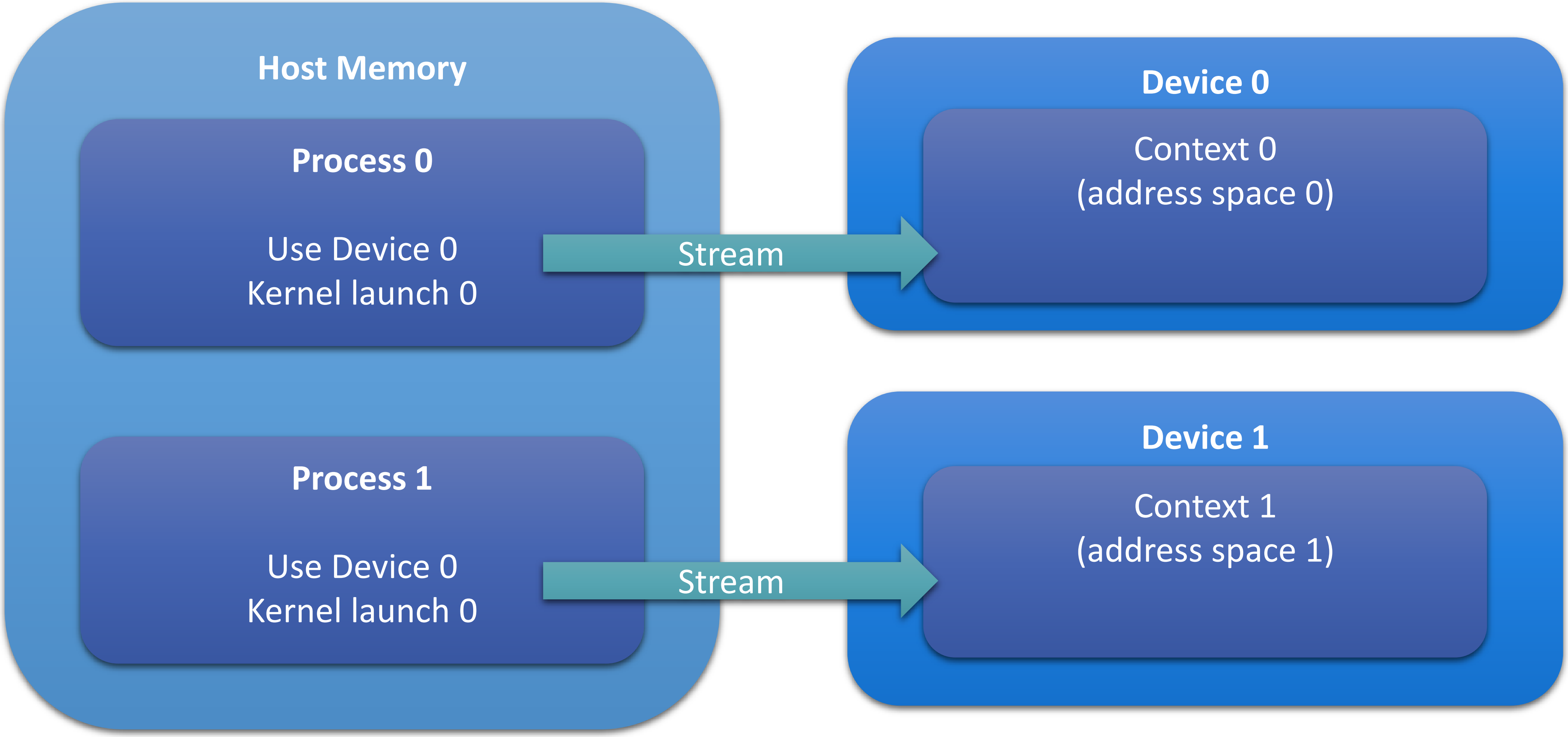
```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
with torch.cuda.stream(s):
    # sum() may start execution before normal_() finishes!!!
    B = torch.sum(A)
```

Missing a synchronization!

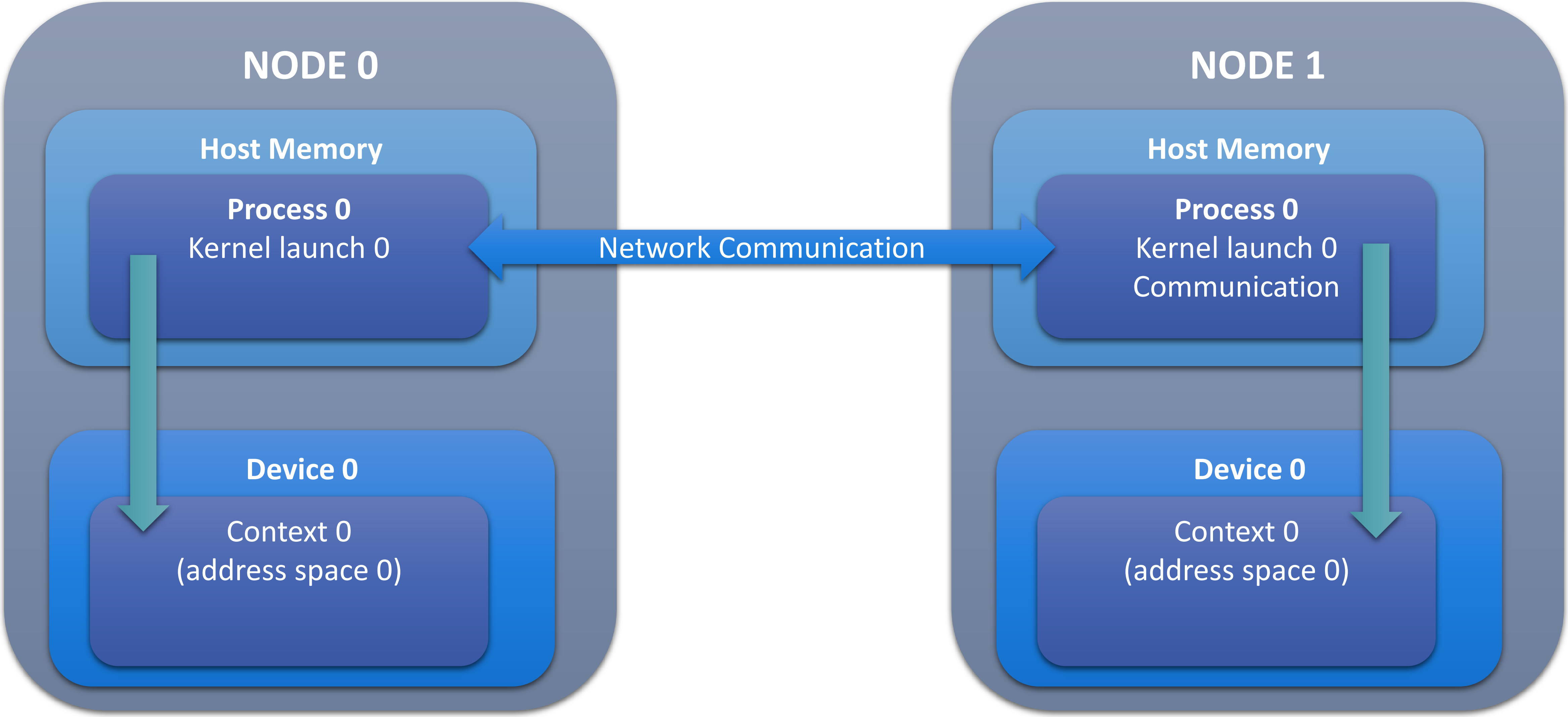
Multi-GPUs with 1 Process



Multi-GPUs with Multiple Processes



Multi-GPU – Multiple processes - Distributed



NODE 0

Host Memory

Process 0
Kernel launch 0

Device 0

Context 0
(address space 0)

Network Communication

NODE 1

Host Memory

Process 0
Kernel launch 0
Communication

Device 0

Context 0
(address space 0)

Lesson Key Points

- Heterogenous architectures motivations
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling