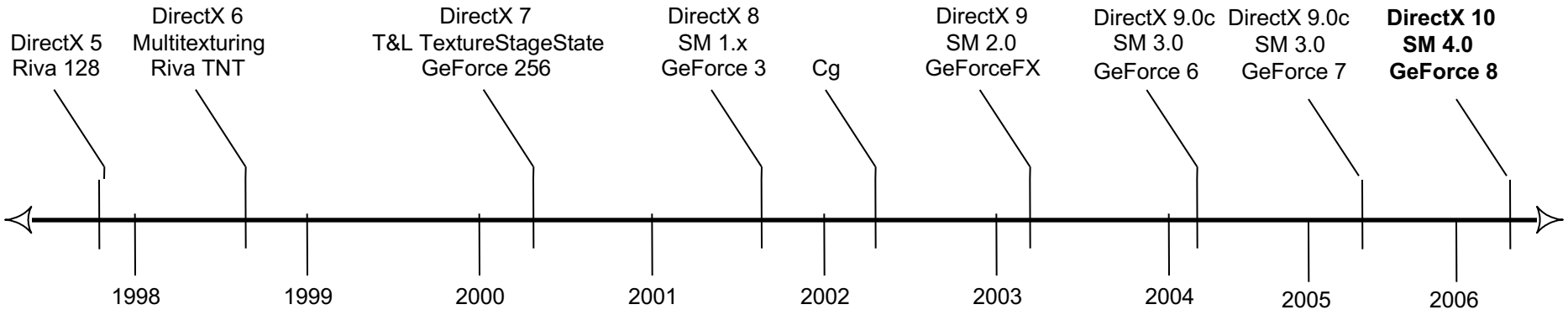


IN5050 – GPU & CUDA

Håkon Kvale Stensland
Simula Research Laboratory / Department for Informatics

PC Graphics Timeline

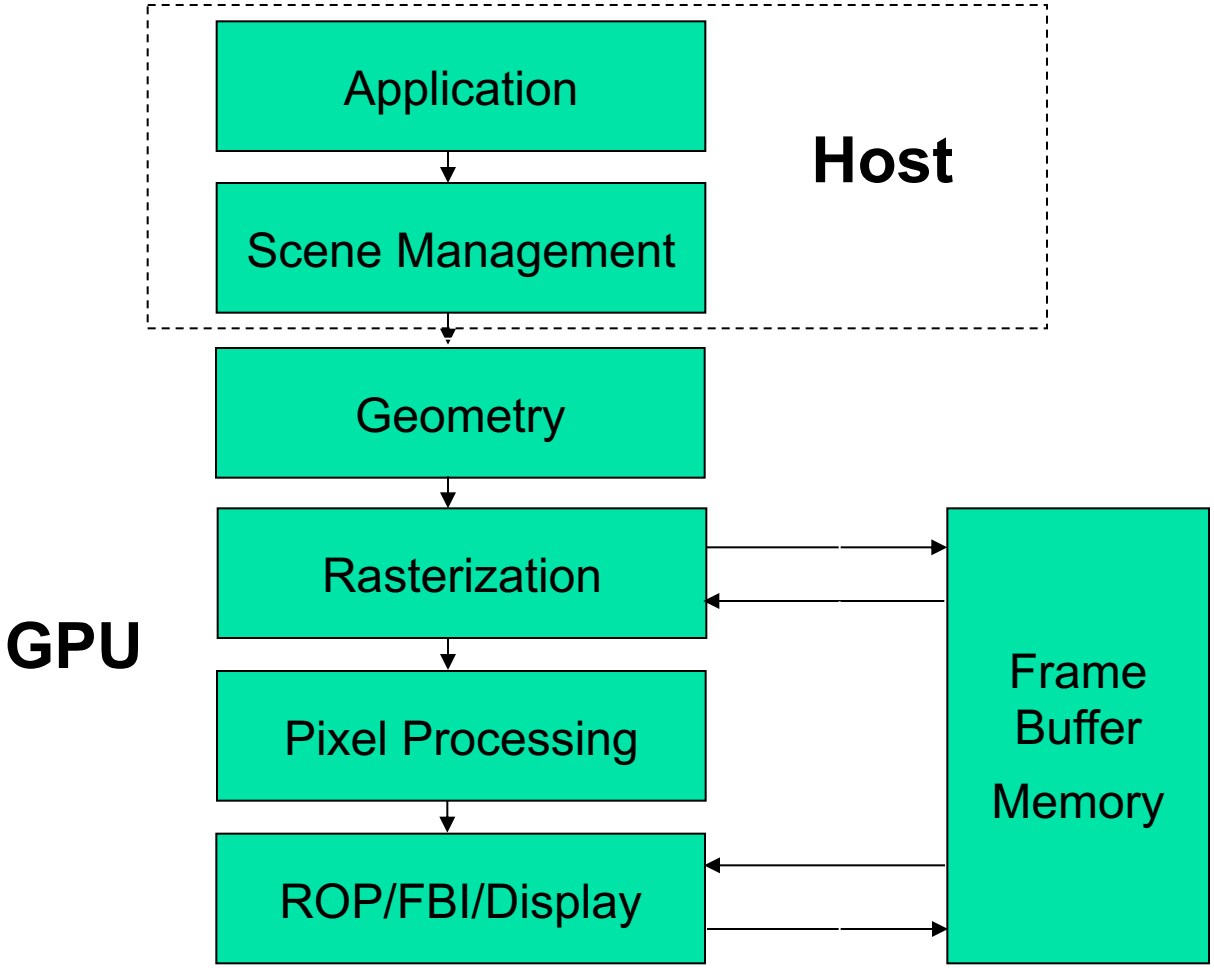
- Challenges:
  - Render infinitely complex scenes
  - And extremely high resolution
  - In 1/60th of one second (60 frames per second)
- Graphics hardware has evolved from a simple hardwired pipeline to a highly programmable multiword processor



GPU – Graphics Processing Units

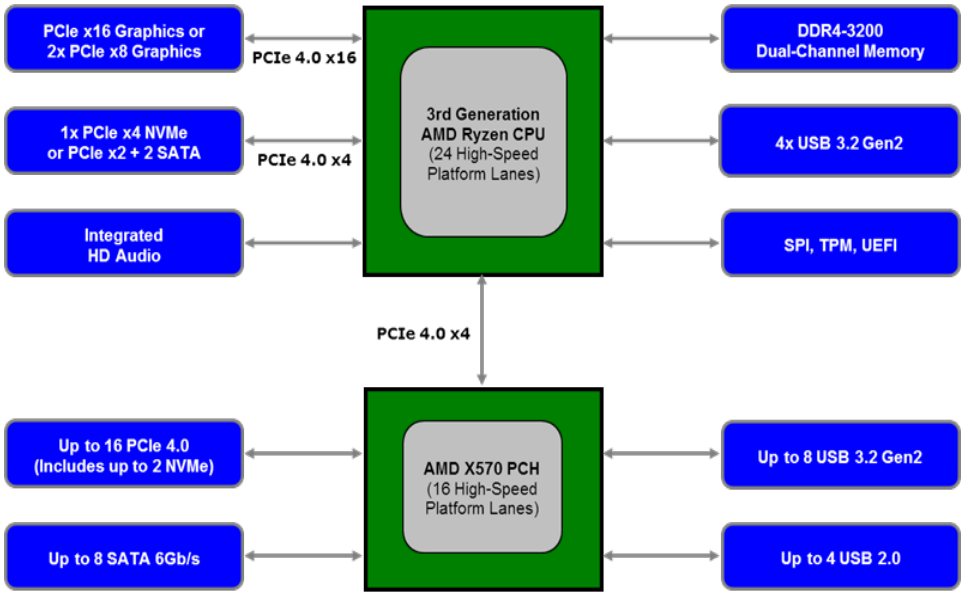


Basic 3D Graphics Pipeline

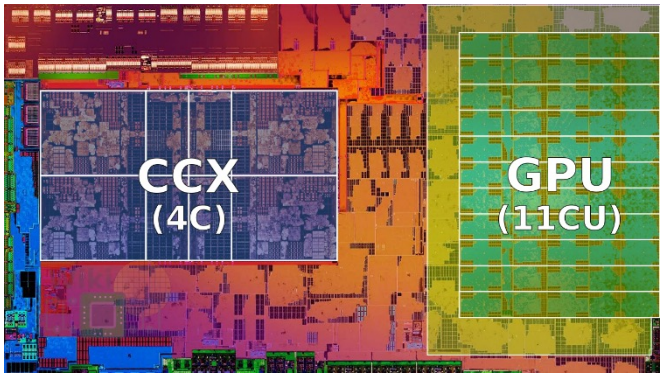


Graphics in the PC Architecture

- PCIe (PCI Express) between processor and chipset
  - Memory Control now integrated in CPU
- The old "Northbridge" integrated onto CPU
  - **PCI Express 4.0 x16 bandwidth at 64 GB/s (32 GB in each direction)**
- "Southbridge" (X570) handles all other peripherals

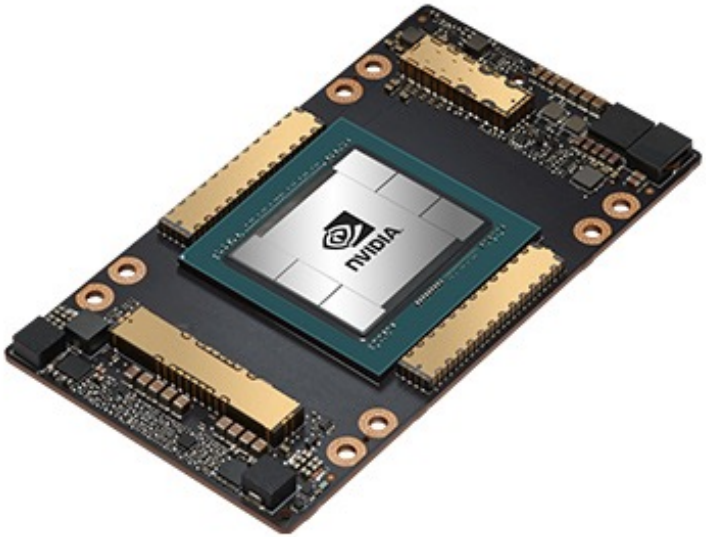


- **Most mainstream CPUs now come with integrated GPU**
  - Same capabilities as discrete GPU's
  - Less performance (limited by die space and power)



AMD «Raven Ridge» Zen+ APU

High-end «Graphics» Hardware

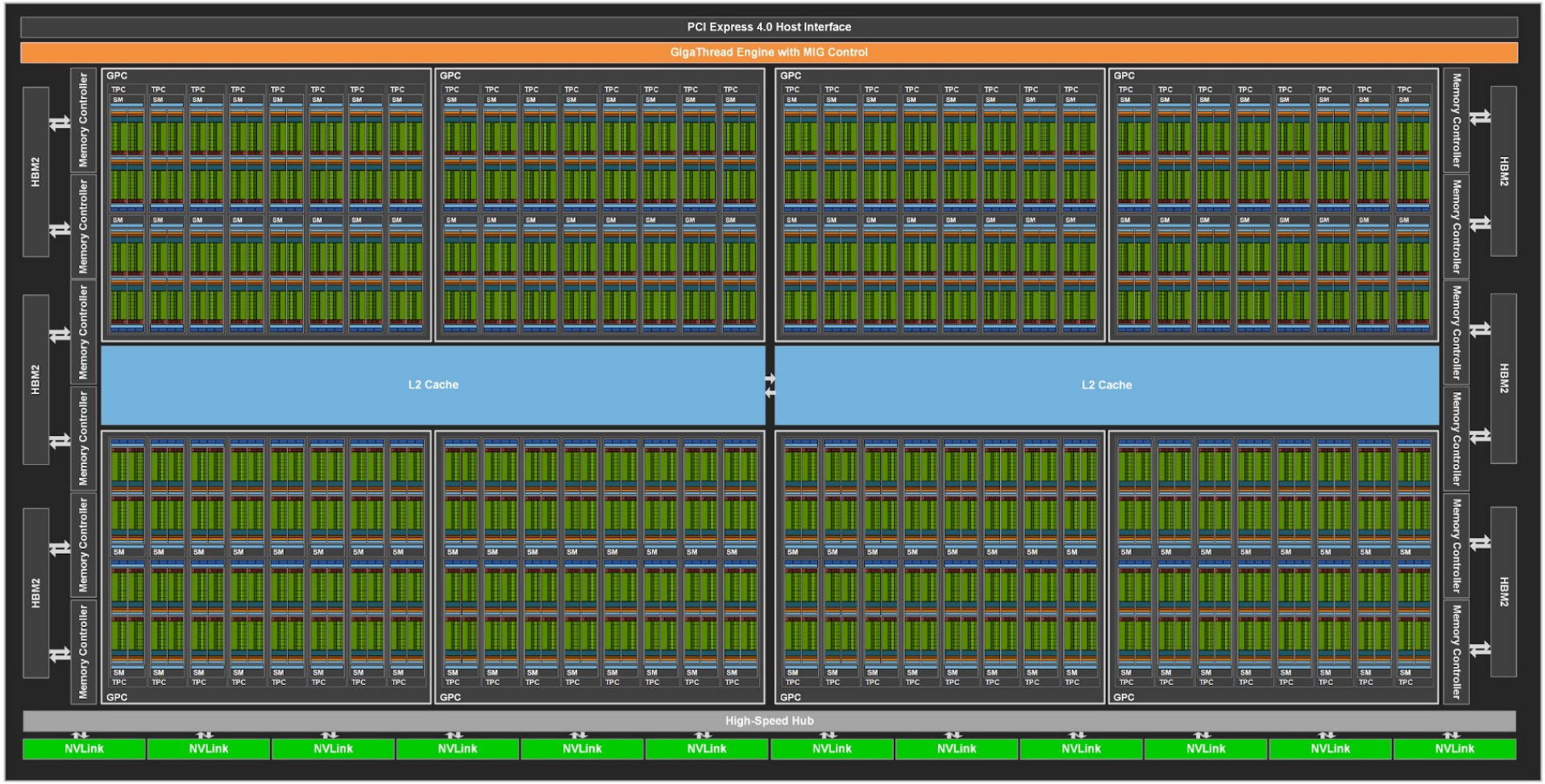


Nvidia A100

- nVIDIA Ampere Architecture
- The latest generation GPU, codenamed A100

- 54,2 **billion** transistors
- 6912 Processing cores (SP)
  - Mixed precision
  - Dedicated Tensor cores
  - PCI Express 4.0
  - NVLink interconnect
  - Hardware support for preemption.
  - Virtual memory
  - 80 GB HBM2e memory
  - Supports GPU virtualization

nVIDIA A100 Architecture



GPUs not always for Graphics

Titan X Pascal (GP102)

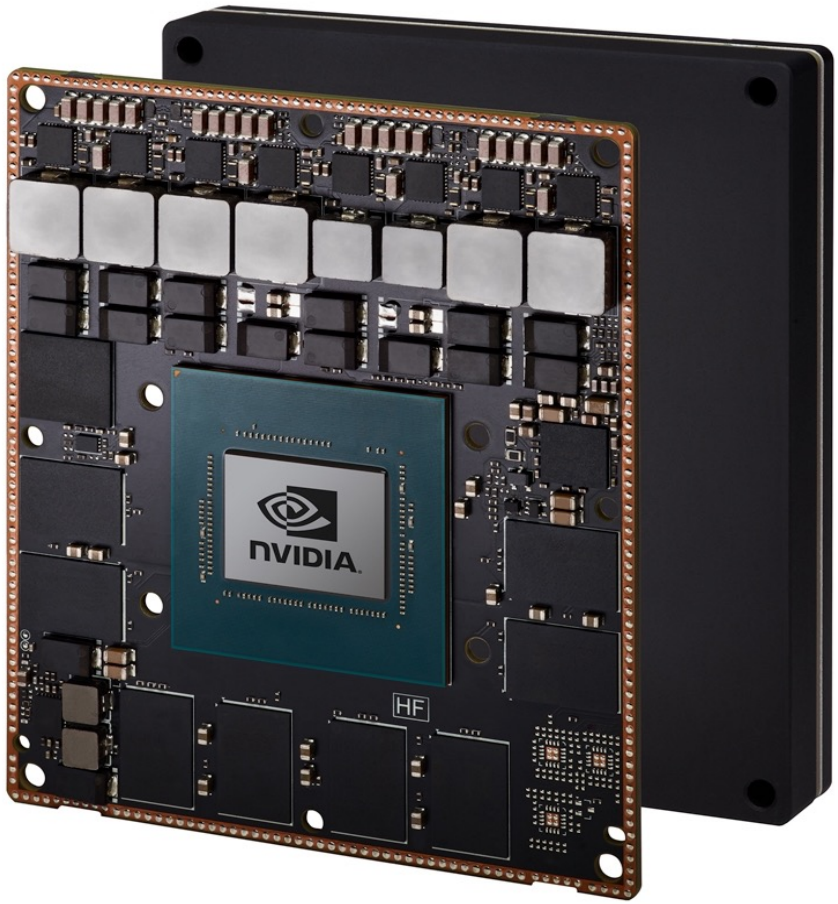


Tesla P40 (GP102)



- GPUs are now common in HPC
- Second largest supercomputer in November 2020 is the **Summit** at Oak Ridge National Laboratory
  - 4356 nodes
    - Two 22-core IBM Power9 CPU's
    - Six Nvidia Tesla V100 GPU's
  - Theoretical: ***200 petaflops***
- *Before:* Dedicated compute card released after graphics model
- *Now:* Nvidia's Ampere architecture (A100) released with a dedicated chip. Graphics variant released 6 months later, on different process node and with major changes (GA10x).

Lab Hardware

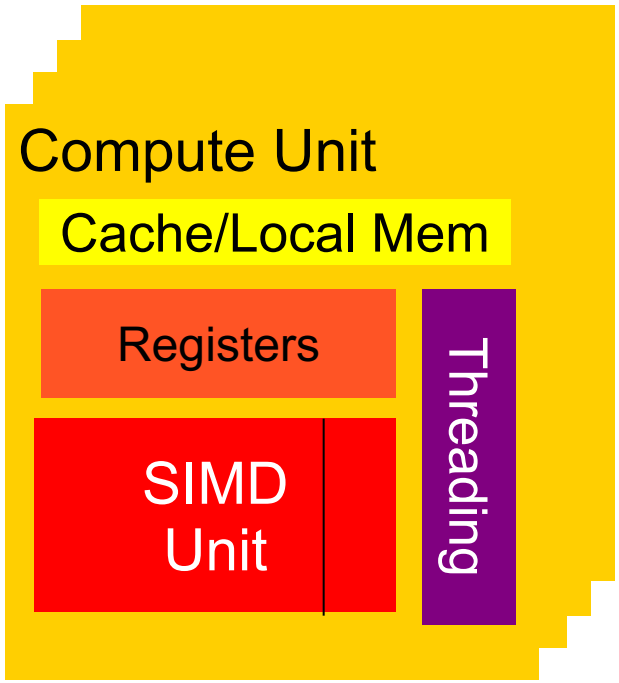


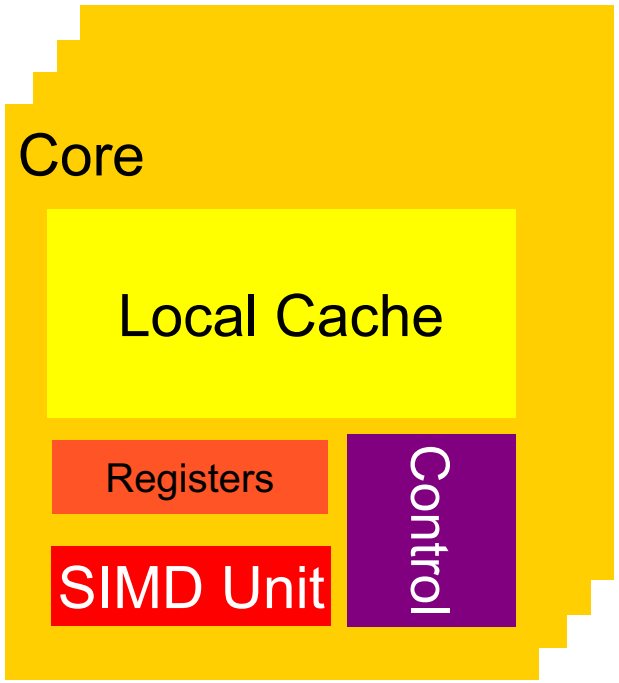
- **nVIDIA Jetson AGX Xavier**
  - **Volta GPU Architecture**
  - **Codename of GPU is GV10B**
- *No desktop or mobile counterpart, similarities with a shrunken TU117*
  - 512 Processing cores (8 Volta SM)
  - 64 Tensor Cores
  - 16/32 GB Memory with 137 GB/sec bandwidth (LPDDR4X)
  - 512 kB Level 2 cache
  - 1,4 TFLOPS theoretical FP32 performance.
  - 2,8 TFLOPS theoretical FP16 performance.
  - Compute version 7.2

CPU and GPU Design Philosophy

GPU
Throughput Oriented Cores

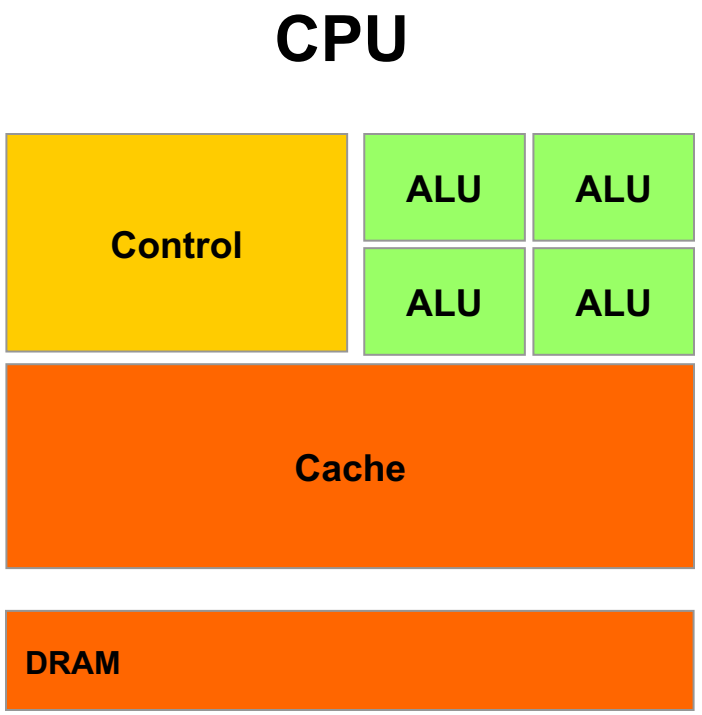
CPU
Latency Oriented Cores





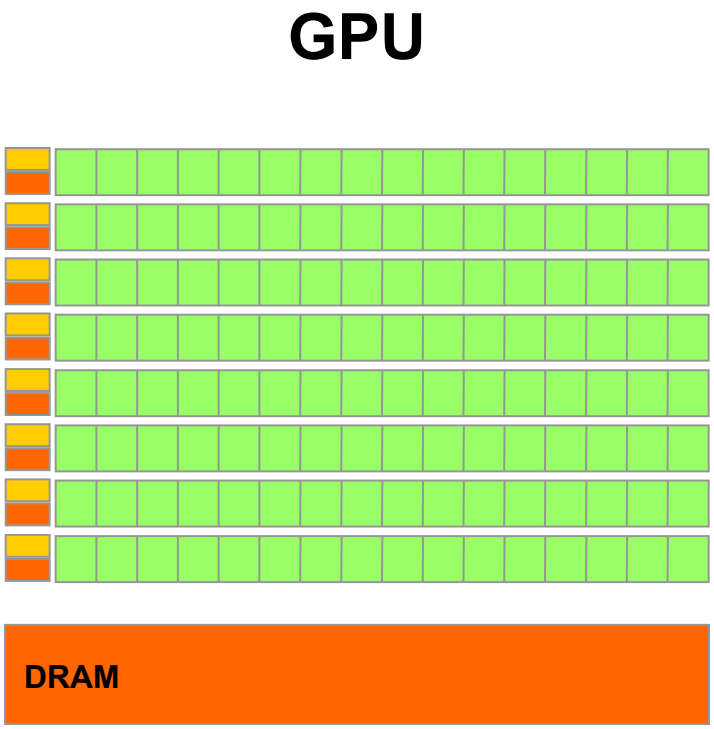
CPUs: Latency Oriented Design

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



GPUs: Throughput Oriented Design

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies

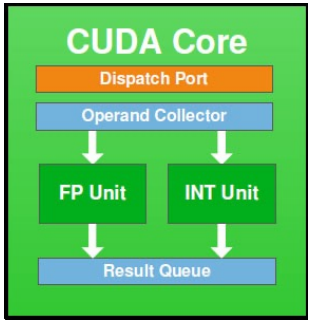


Think both about CPU and GPU…

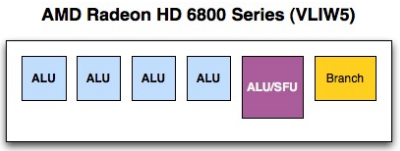
- CPUs for sequential parts where latency matters
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code

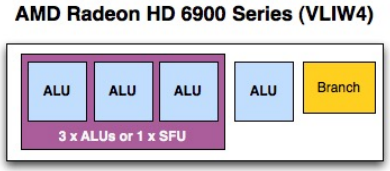
The Core: The basic processing block

- *The nVIDIA Approach:*
  - Called Stream Processor and CUDA cores. Works on a single operation.

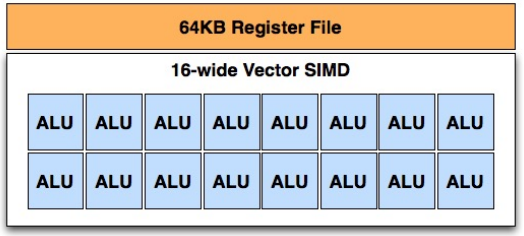


- *The AMD Approach:*
  - *VLIW5:* The GPU work on up to five operations
  - VLIW4: The GPU work on up to four operations
  - GCN: 16-wide SIMD vector unit



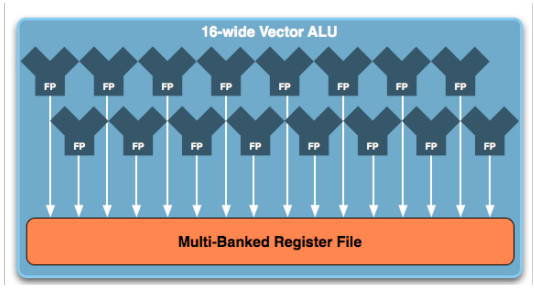


Graphics Core Next (GCN):

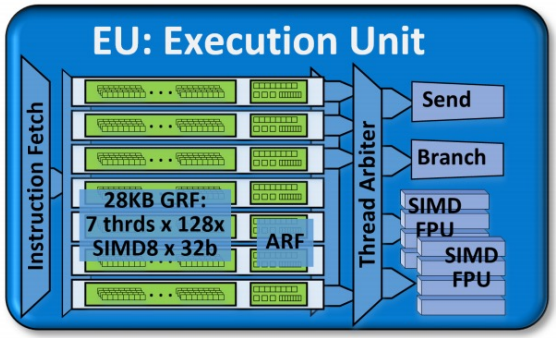


The Core: The basic processing block

- The (failed) Intel Approach:
  - 512-bit SIMD units in x86 cores
  - Failed because of complex x86 cores and software ROP pipeline
  - Used in Xeon Phi, and basis for AVX-512



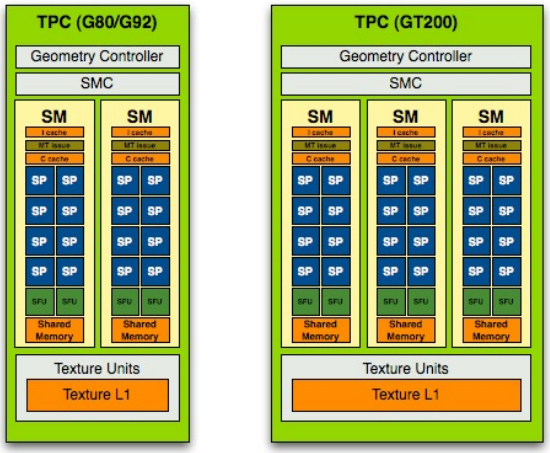
- The (new) Intel Approach:
  - Used in Sandy Bridge, Haswell, Skylake and Xe
  - 128 SIMD-8 32-bit registers



The Evolution of The «CUDA Cores»

The nVIDIA GPU Architecture Evolving

- **Streaming Multiprocessor (SM) 1.x on the Tesla Architecture**
  - 8 CUDA Cores (Core)
  - 2 Super Function Units (SFU)
- Dual schedulers and dispatch units
  - 1 to 512 or 768 threads active
- Local register (32k)
- 16 KB shared memory
- 2 operations per cycle



- **Streaming Multiprocessor (SM) 2.0 on the Fermi Architecture (GF1xx)**
  - 32 CUDA Cores (Core)
  - 4 Super Function Units (SFU)
- Dual schedulers and dispatch units
  - 1 to 1536 threads active
- Local register (32k)
- 64 KB shared memory / Level 1 cache
- 2 operations per cycle



The nVIDIA GPU Architecture Evolving

- **Streaming Multiprocessor (SMX) 3.x on the Kepler Architecture (Graphics)**
  - 192 CUDA Cores (CC)
  - 8 DP CUDA Cores (DP Core)
  - 32 Super Function Units (SFU)
- Four (simple) schedulers and eight dispatch units
  - 1 to 2048 threads active
- Local register (32k)
- 64 KB shared memory / Level 1 cahce
- 1 operation per cycle



- **Streaming Multiprocessor (SMM) on the Maxwell & Pascal Architecture**
  - 128 CUDA Cores (Core)
  - 4 DP CUDA Cores (DP Core)
  - 32 Super Function Units (SFU)
- Four schedulers and eight dispatch units
  - 1 to 2048 threads active
- Local register (64k)
- 64 KB shared memory
- 24 KB Level 1 / Texture Cache
- 1 operation per cycle



Volta Streaming Multiprocessor (Volta SM)

- **Streaming Multiprocessor (Volta SM) on Volta**
  - 64 CUDA Cores (Core)
  - 32 DP CUDA Cores (DP Core)*
  - 16 Super Function Units (SFU)
  - 8 Tensor Cores (GEMM)
- Four schedulers and eight dispatch units
  - 1 to 2048 active threads
  - Software controlled scheduling
- Local register (64k)
- 128 KB Level 1 / Shared Memory
  - Unified Data Cache
- 1 operation per cycle
- **GV100 / GV10B**





GPGPU

Foils adapted from nVIDIA

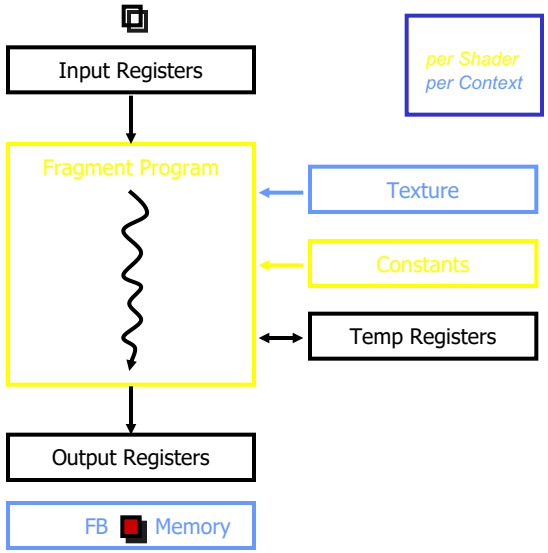
What is really GPGPU?

- Idea:
  - Potential for very high performance at low cost
  - Architecture well suited for certain kinds of parallel applications ***(data parallel)***
  - Demonstrations of ***30-100X*** speedup over CPU

- Early challenges:
  - Architectures very customized to graphics problems (e.g., vertex and fragment processors)
  - Programmed using graphics-specific programming models or libraries

Previous GPGPU use, and limitations

- Working with a Graphics API
  - Special cases with an API like Microsoft Direct3D or OpenGL
- Addressing modes
  - Limited by texture size
- Shader capabilities
  - Limited outputs of the available shader programs
- Instruction sets
  - No integer or bit operations
- Communication is limited
  - Between pixels



Heterogeneous computing is catching on...

- Financial Analysis
- Scientific Simulation
- Engineering Simulation
- Data Intensive Analytics
- Medical Imaging
- Digital Audio Processing
- Digital Video Processing
- Computer Vision
- Biomedical Informatics
- Electronic Design Automation
- Statistical Modeling
- Ray Tracing Rendering
- Interactive Physics
- Numerical Methods

nVIDIA CUDA



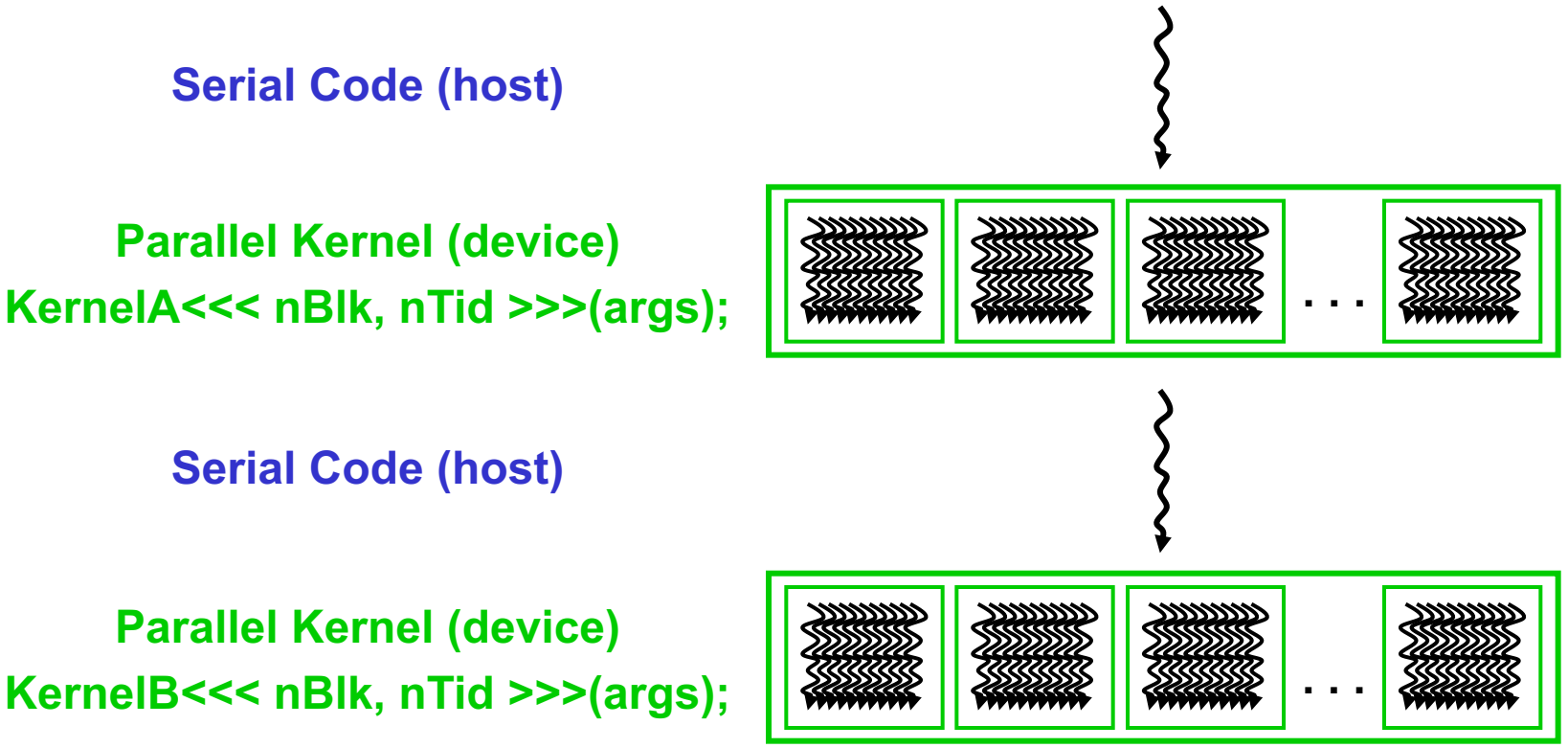
- "Compute Unified Device Architecture"
- General purpose programming model
  - User starts several batches of threads on a GPU
  - GPU is in this case a dedicated super-threaded, massively data parallel co-processor
- Software Stack
  - Graphics driver, language compilers (Toolkit), and tools (SDK)
- Graphics driver loads programs into GPU
  - All drivers from nVIDIA now support CUDA
  - Interface is designed for computing (no graphics ☺)
  - "Guaranteed" maximum download & readback speeds
  - Explicit GPU memory management

The CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU, referred to as the host
  - Has its own DRAM called device memory
  - Runs **many** threads in parallel
- Data-parallel parts of an application are executed on the device as kernels, which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

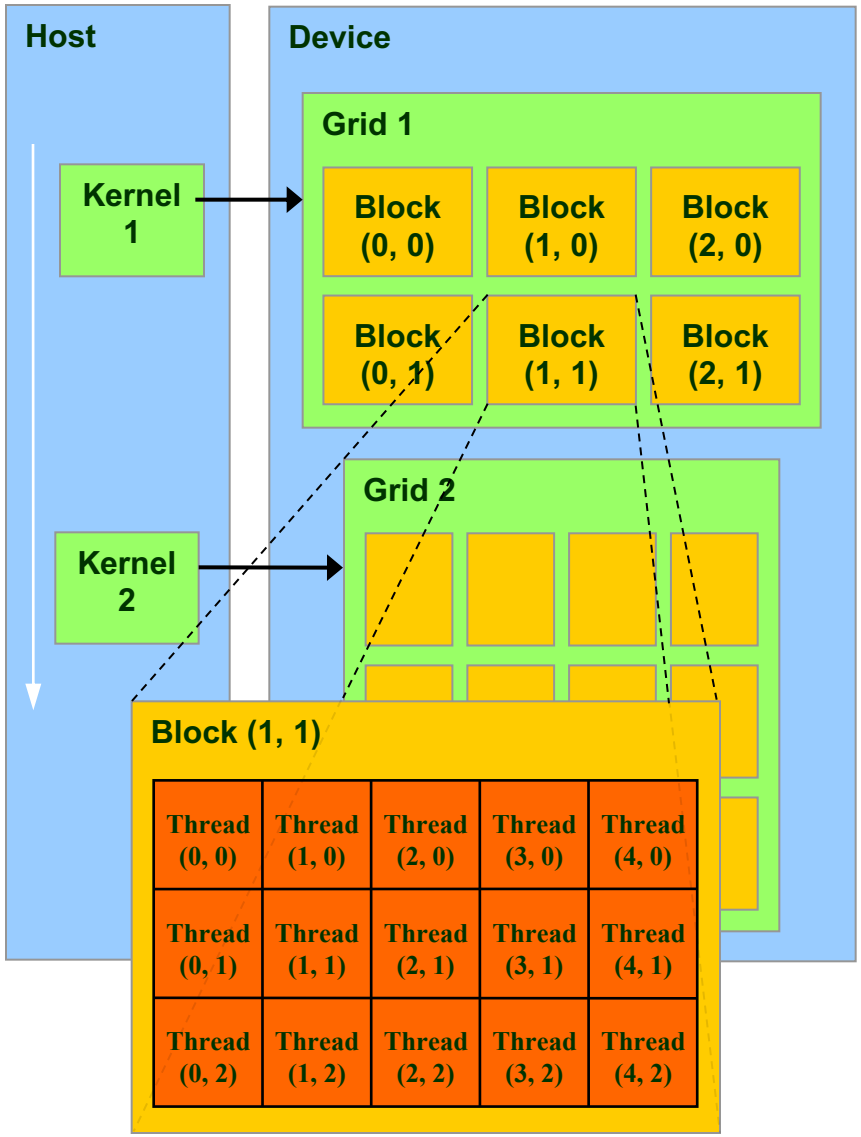
CUDA C – Execution Model

- Integrated host + device C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



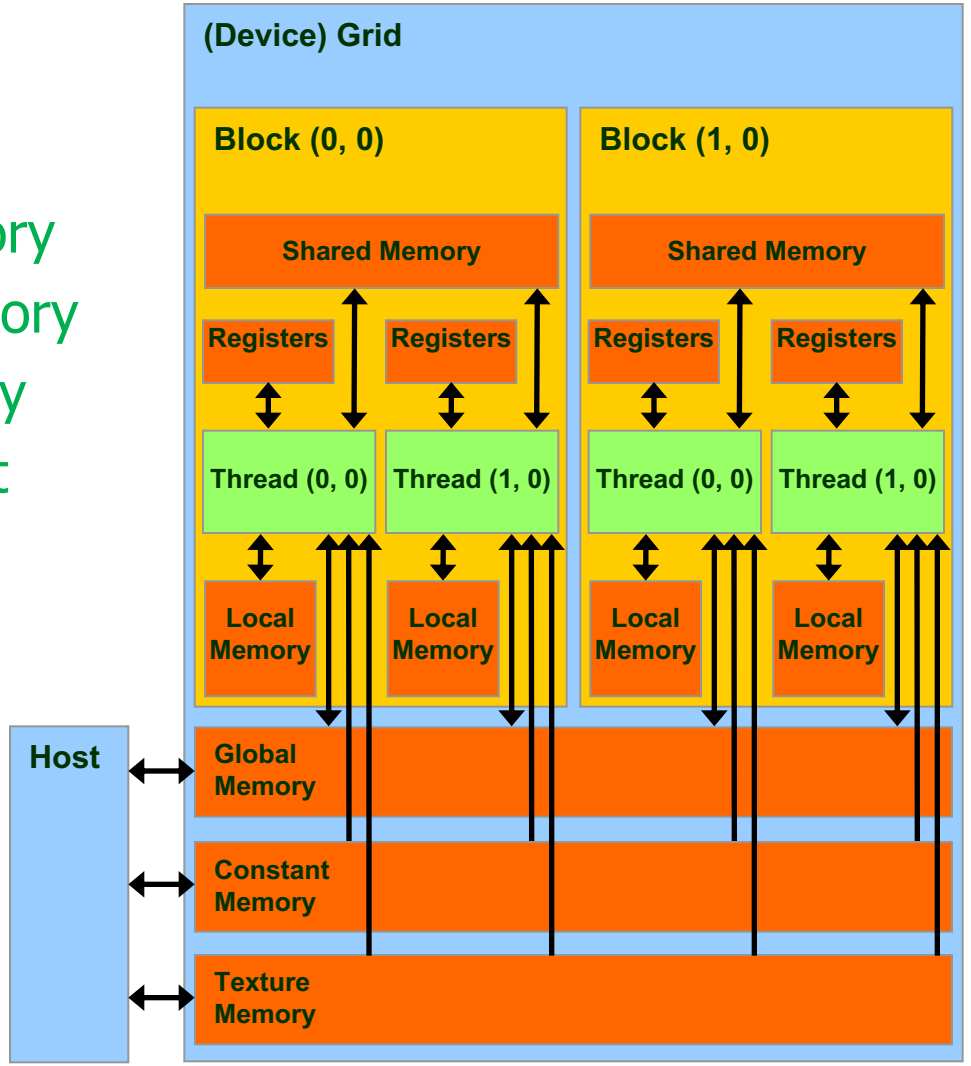
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - Non synchronous execution is very bad for performance!
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



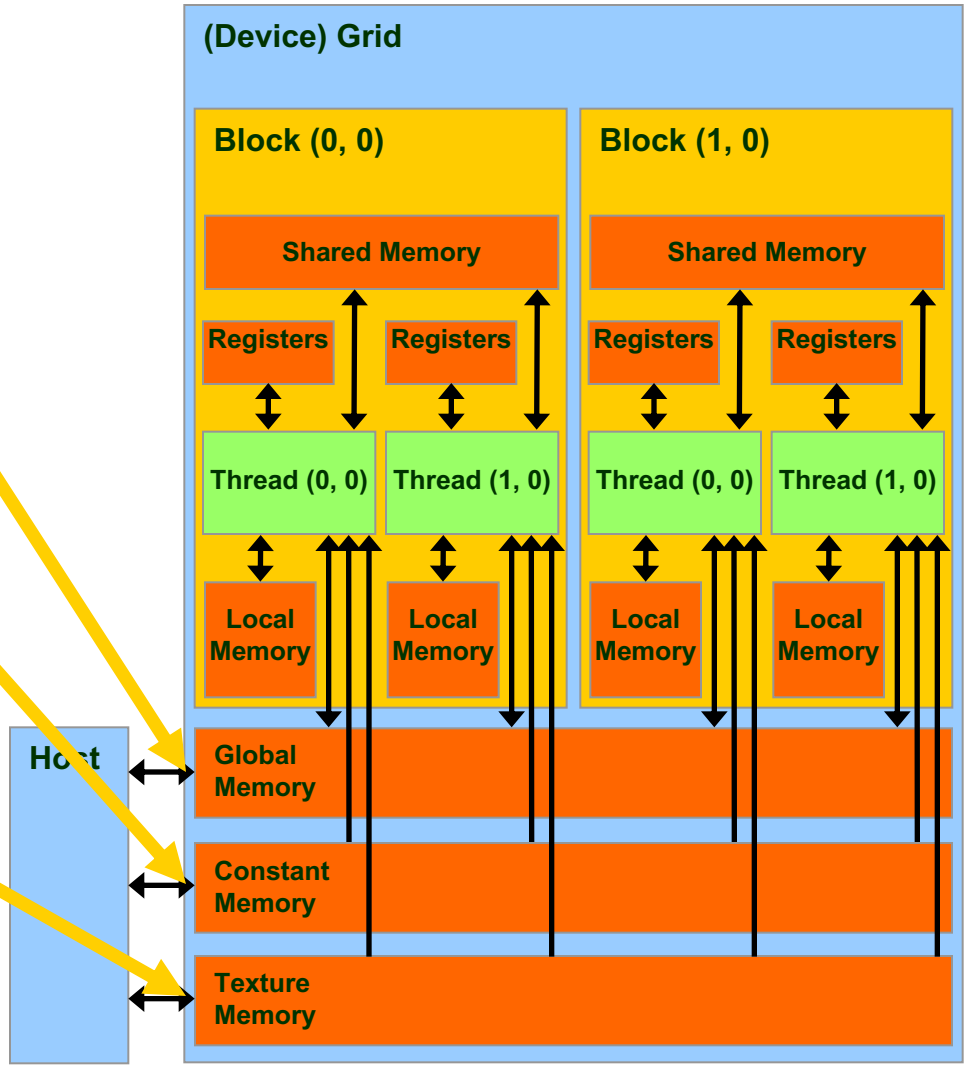
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

- Global memory:
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories:
  - Constants initialized by host
  - Contents visible to all threads



Access Times

- Register – Dedicated HW – Single cycle
- Shared Memory – Dedicated HW – Single cycle
- Local Memory – DRAM, no cache* – "Slow"
- Global Memory – DRAM, no cache* – "Slow"
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality

* Can be cached in L2 or L1 cache on GPU, however this can not be controlled by the programmer.

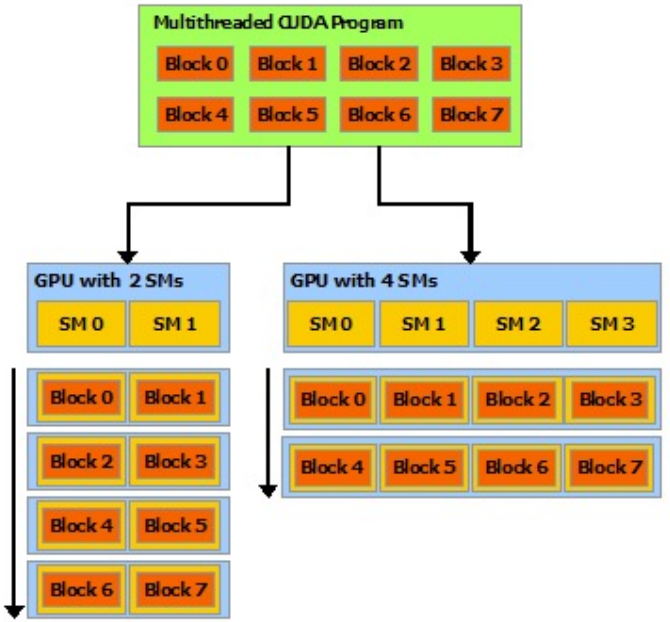
Terminology Recap

- device = GPU = Set of multiprocessors
- Multiprocessor = Set of processors & shared memory
- Kernel = Program running on the GPU
- Grid = Array of thread blocks that execute a kernel
- Thread block = Group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No (Has L2 cache)	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No (Has L2 cache)	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read/write	All threads + host

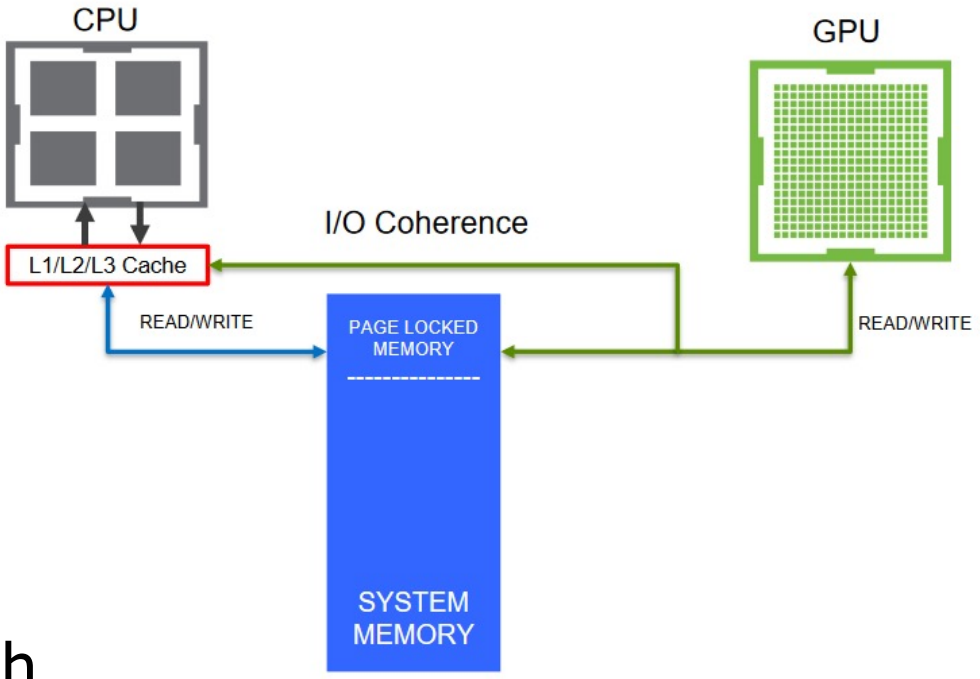
Scalability

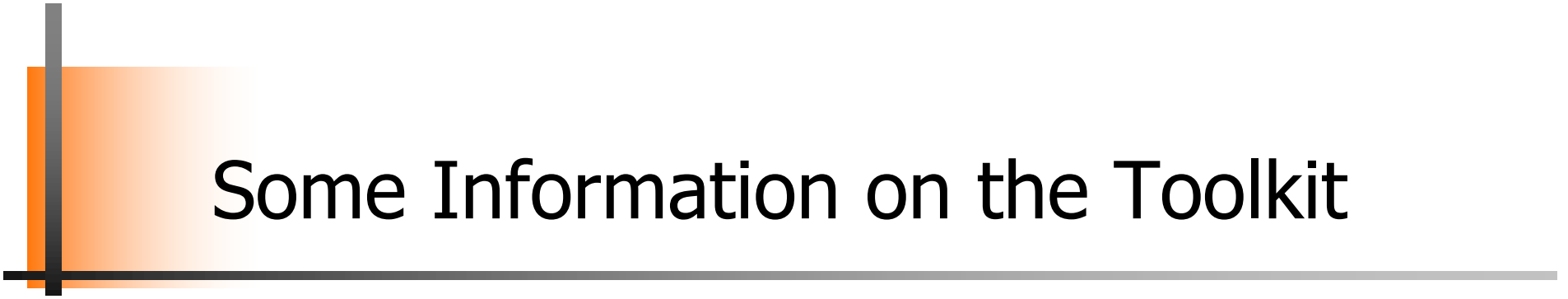
- GPU is built around an array of Streaming Multiprocessors (SMs)
- CUDA has three key abstractions:
  - Hierarchy of thread groups
  - Shared memories
  - Barrier synchronization
- The CUDA Runtime will scale the program to the available resources
  - JIT Compile



Data movement between CPU and GPU

- You are developing for a system were CPU and GPU share memory!
- Xavier supports I/O Coherency:
  - Hardware cache coherency between CPU and GPU
  - GPU can snoop CPU cache hierarchy
- Enables memory models such as Managed Memory and Pinned Host Memory




Some Information on the Toolkit

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, etc.
- nvcc can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

Linking & Profiling

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)
- Several tools are available to optimize your application
  - nVIDIA CUDA Visual Profiler
  - nVIDIA Occupancy Calculator
- NVIDIA Parallel Nsight for Visual Studio and Eclipse

Before you start…

- Four lines must be added to your group users *.bashrc* file

```
PATH=$PATH:/usr/local/cuda-10.2/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-10.2/lib64:/lib

export PATH
export LD_LIBRARY_PATH
```

- Code samples is installed with CUDA

  /usr/local/cuda/samples
- Copy and build in your user's home directory

Some usefull resources

NVIDIA CUDA Programming Guide 10.2

https://docs.nvidia.com/cuda/archive/10.2/

NVIDIA CUDA C Best Practices Guide 10.2

https://docs.nvidia.com/cuda/archive/10.2/cuda-c-best-practices-guide/

Tuning CUDA Applications for Volta

https://docs.nvidia.com/cuda/archive/10.2/volta-compatibility-guide/

CUDA for Tegra

https://docs.nvidia.com/cuda/archive/10.2/cuda-for-tegra-appnote/

Parallel Thread Execution ISA – Version 6.3

https://docs.nvidia.com/cuda/archive/10.2/parallel-thread-execution/