Master Informatics Eng.

2016/17

A.J.Proença

Data Parallelism 3 (*GPU & CUDA*)

(most slides are borrowed)

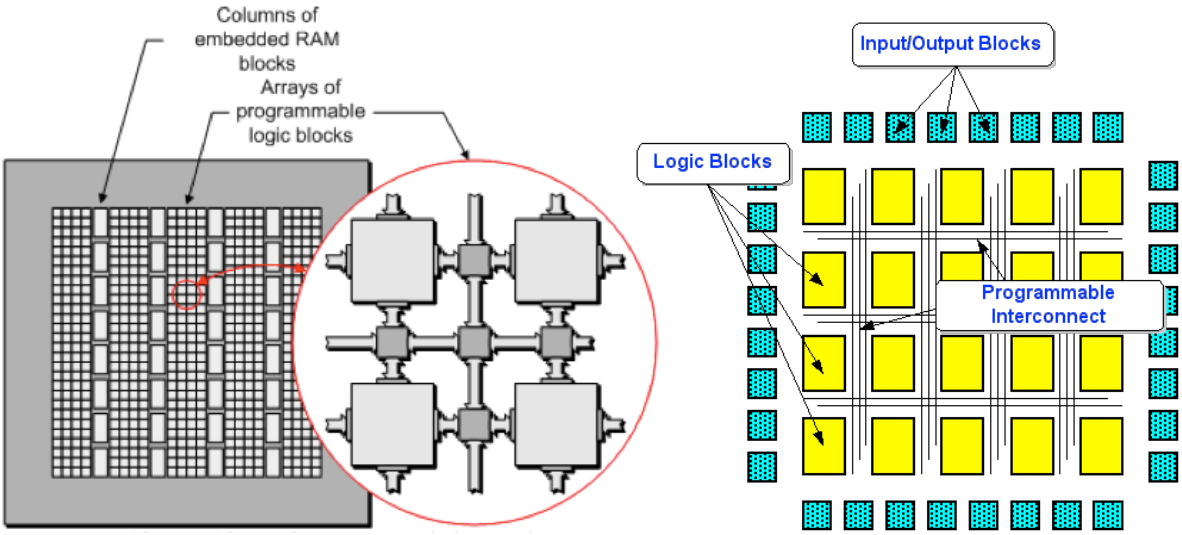
Beyond Vector/SIMD architectures

- Vector/SIMD-extended architectures are hybrid approaches
  - mix (super)**scalar + vector** op capabilities on a single device
  - **highly pipelined** approach to reduce memory access penalty
  - **tightly-closed access to shared memory**: lower latency
- Evolution of Vector/SIMD-extended architectures
  - **CPU cores with wider vectors and/or SIMD cores**:
    - DSP VLIW cores with vector capabilities: **Texas Instruments** (...?)
    - PPC cores coupled with SIMD cores: **Cell** (past...) , **IBM Power BQC...**
    - ARM64 cores coupled with SIMD cores: from Tegra to Parker (**NVidia**) (...?)
    - x86 many-core: **Intel** MIC / Xeon KNL**, PEZY-SC, AMD** FirePro**...**
    - other many-core: **ShenWay** 260, **Adapteva** Epiphany-V**...**
  - **coprocessors (require a host scalar processor): accelerator devices**
    - on disjoint physical memories (e.g., Xeon KNC with **PCI-E**xpr, **PEZY-SC**)
    - focus on SIMT/SIMD to hide memory latency: **GPU**-type approach
    - ISA-free architectures, code compiled to silica: **FPGA**

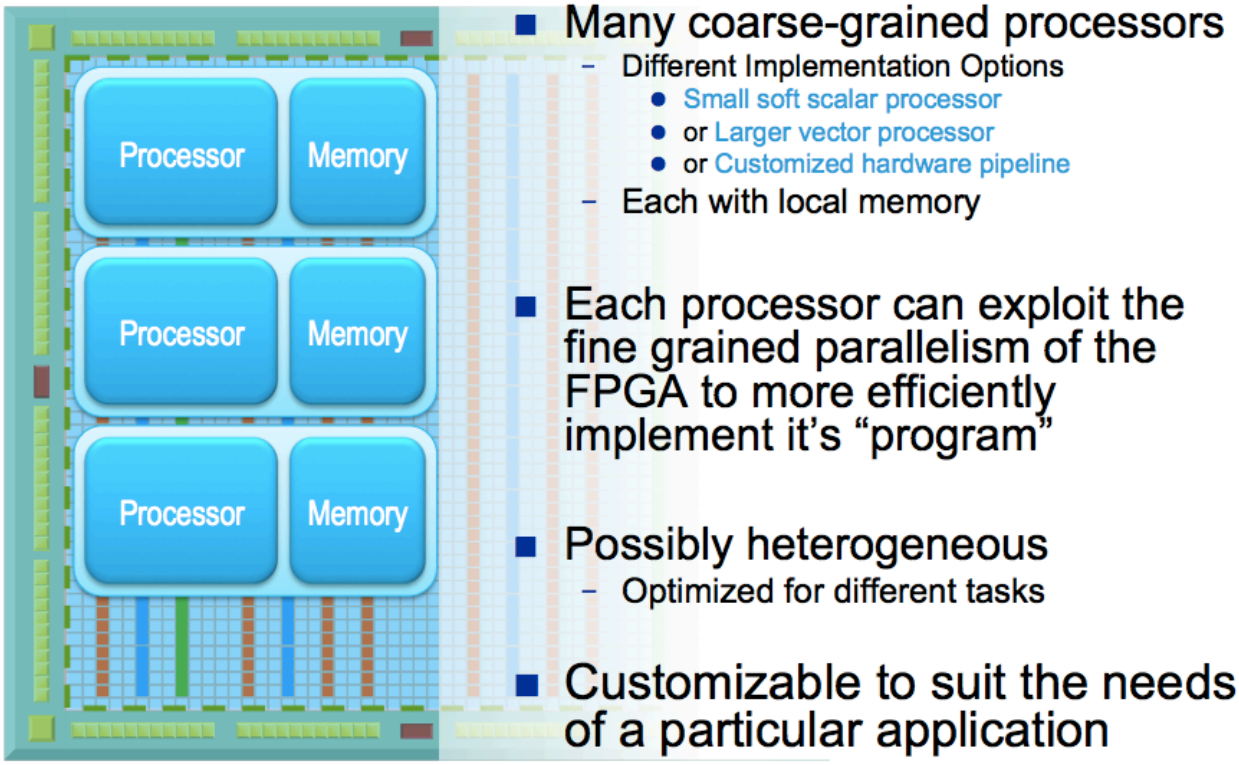
What is an FPGA

Field-Programmable Gate Arrays (FPGA)

A fabric with 1000s of simple configurable logic cells with LUTs, on-chip SRAM, configurable routing and I/O cells



FPGA as a multiple configurable ISA



- Many coarse-grained processors
  - Different Implementation Options
    - Small soft scalar processor
    - or Larger vector processor
    - or Customized hardware pipeline
  - Each with local memory
- Each processor can exploit the fine grained parallelism of the FPGA to more efficiently implement it's "program"
- Possibly heterogeneous
  - Optimized for different tasks
- Customizable to suit the needs of a particular application

Beyond Vector/SIMD architectures

- Vector/SIMD-extended architectures are hybrid approaches
  - mix (super)**scalar + vector** op capabilities on a single device
  - **highly pipelined** approach to reduce memory access penalty
  - **tightly-closed access to shared memory**: lower latency
- Evolution of Vector/SIMD-extended architectures
  - **CPU cores with wider vectors and/or SIMD cores**:
    - DSP VLIW cores with vector capabilities: **Texas Instruments** (...?)
    - PPC cores coupled with SIMD cores: **Cell** (past...) , **IBM Power BQC...**
    - ARM64 cores coupled with SIMD cores: from Tegra to Parker (**NVidia**) (...?)
    - x86 many-core: **Intel** MIC / Xeon KNL**, PEZY-SC, AMD** FirePro**...**
    - other many-core: **ShenWay** 260, **Adapteva** Epiphany-V**...**
  - **coprocessors (require a host scalar processor): accelerator devices**
    - on disjoint physical memories (e.g., Xeon KNC with **PCI-E**xpr, **PEZY-SC**)
    - focus on SIMT/SIMD to hide memory latency: **GPU**-type approach
    - ISA-free architectures, code compiled to silica: **FPGA**

Graphical Processing Units

- Question to GPU architects:
  - *Given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?*
- Key ideas:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA_threads*
  - Programming model follows SIMT: "*Single Instruction Multiple Thread* "

cores/processing elements in several devices

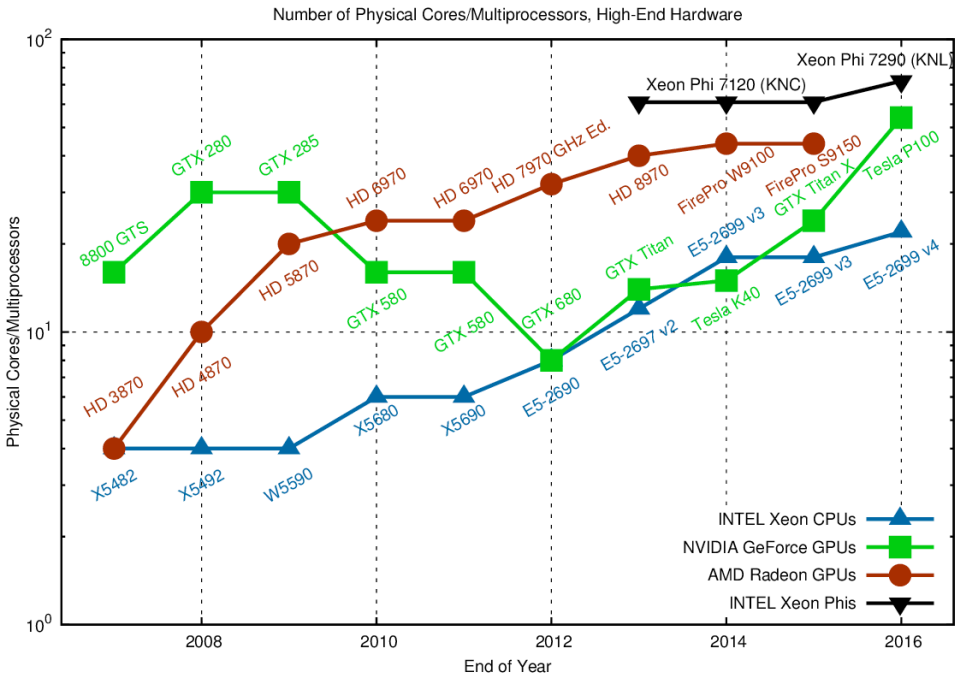
Key question: what is a **core**?

a) IU+FPU?
GPU-type...

b) A SIMD processor?
CPU-type..

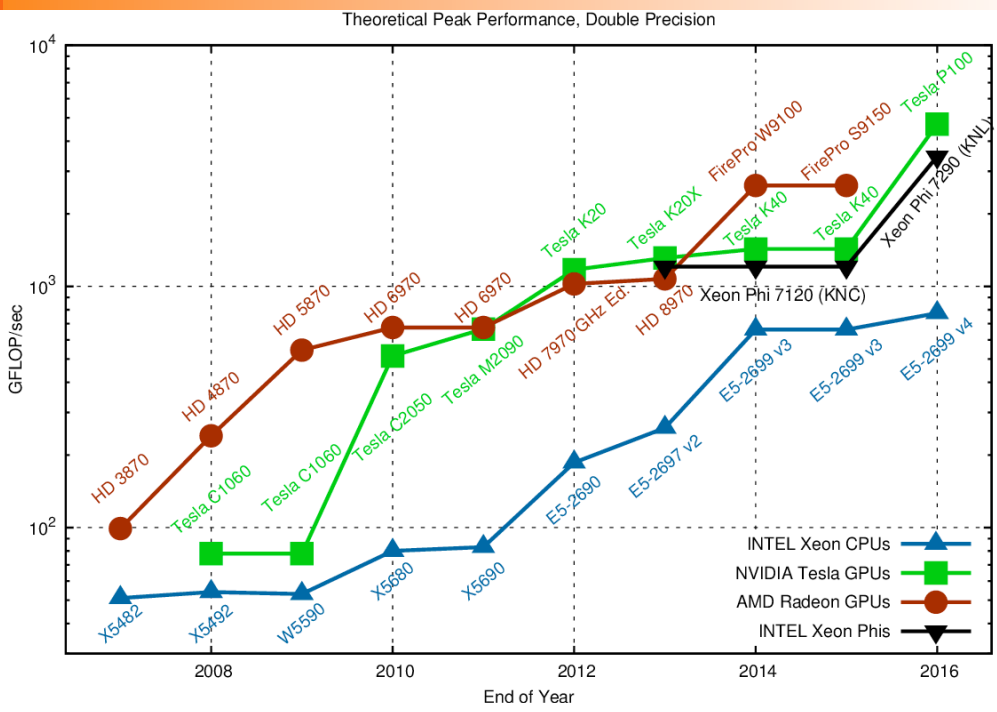
This updated slide and in this course:
- **b)**

Note: the web link with these plots was updated in Aug'16



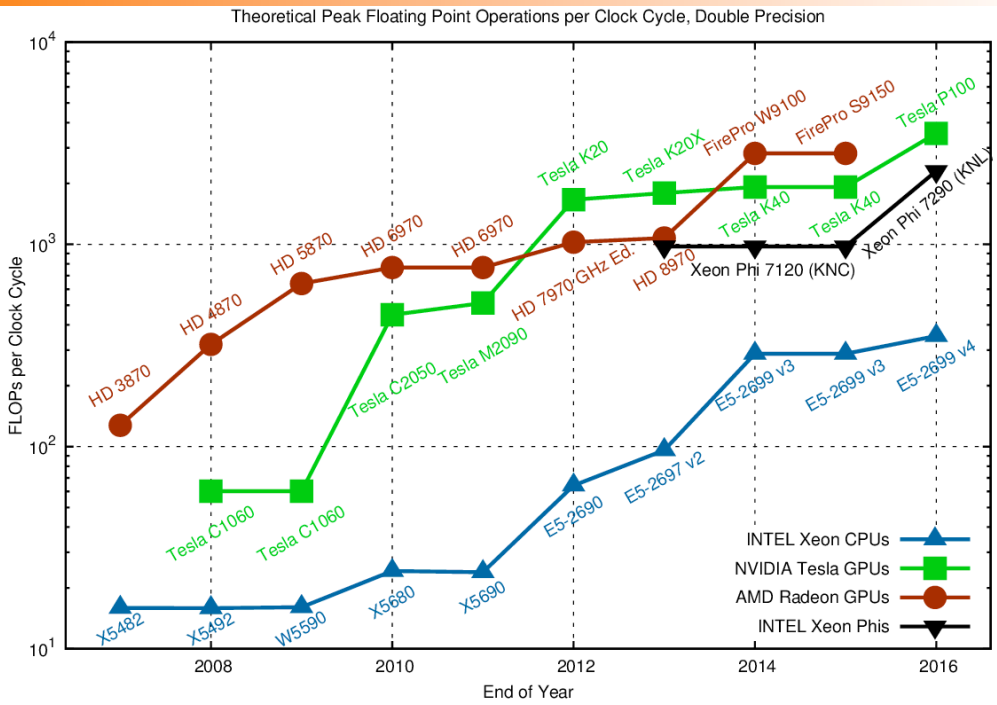
http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Theoretical peak performance in several computing devices (DP)



http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Theoretical peak FP Op's per clock cycle in several computing devices (DP)



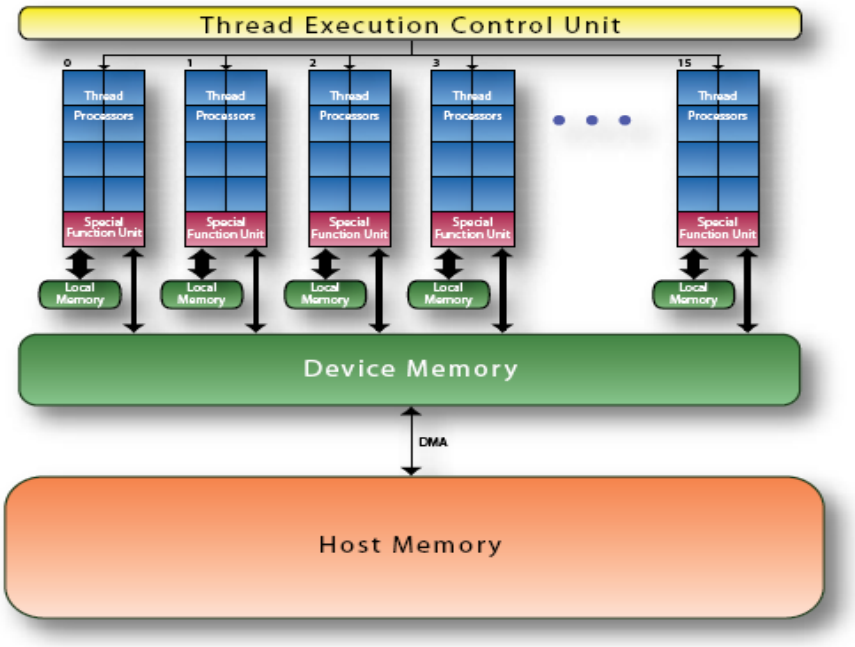
http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Graphical Processing Units

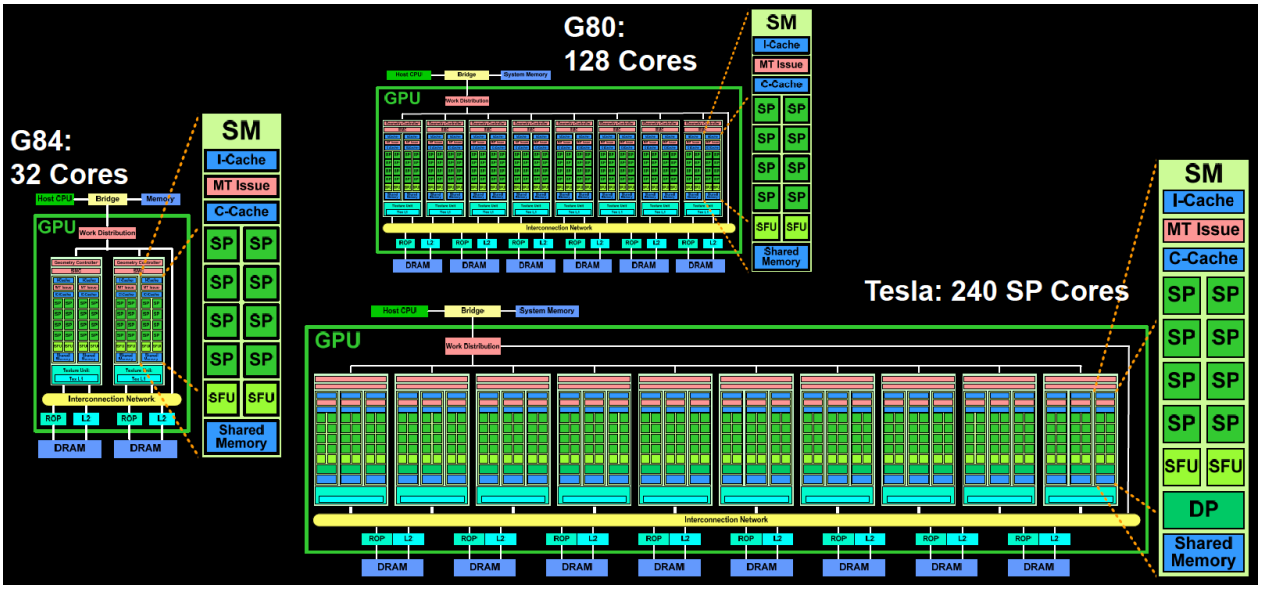
NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

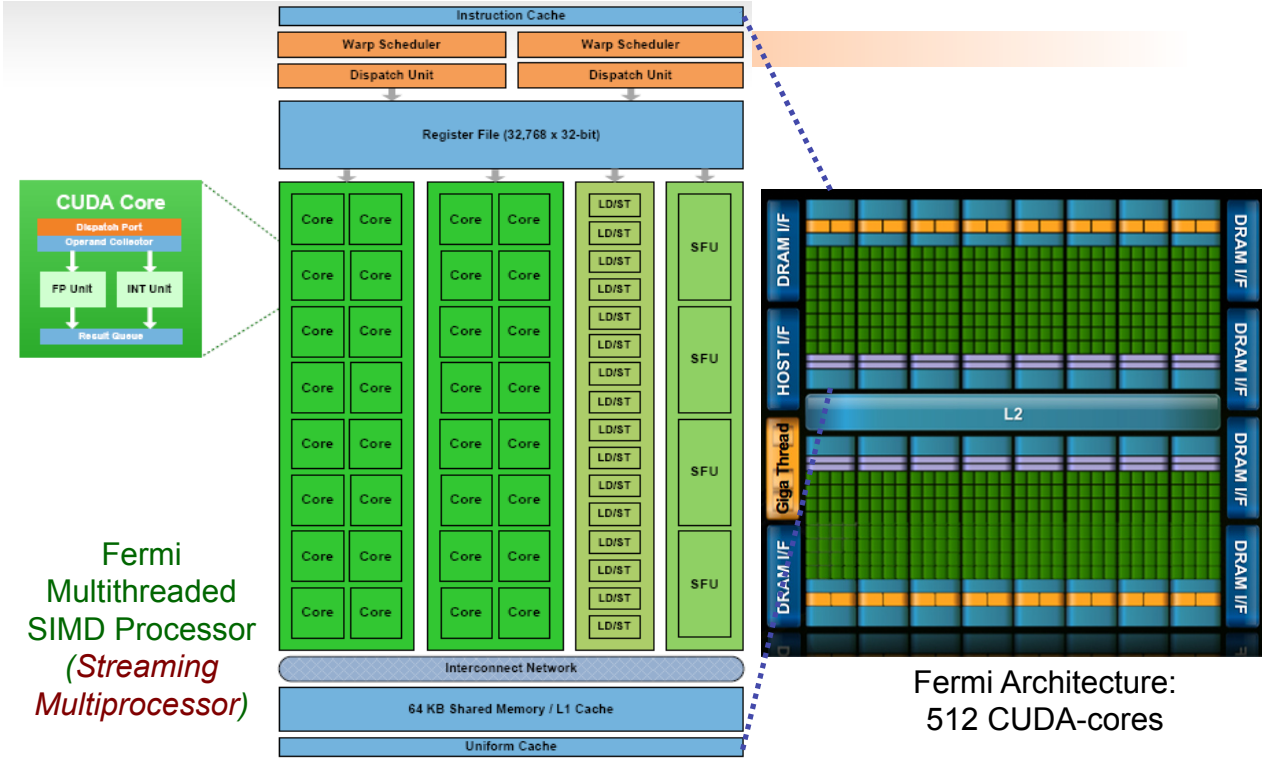
The GPU as a compute device: the G80



NVidia GPU structure & scalability



The NVidia Fermi architecture



Fermi Multithreaded SIMD Processor *(Streaming Multiprocessor)*

Fermi Architecture: 512 CUDA-cores

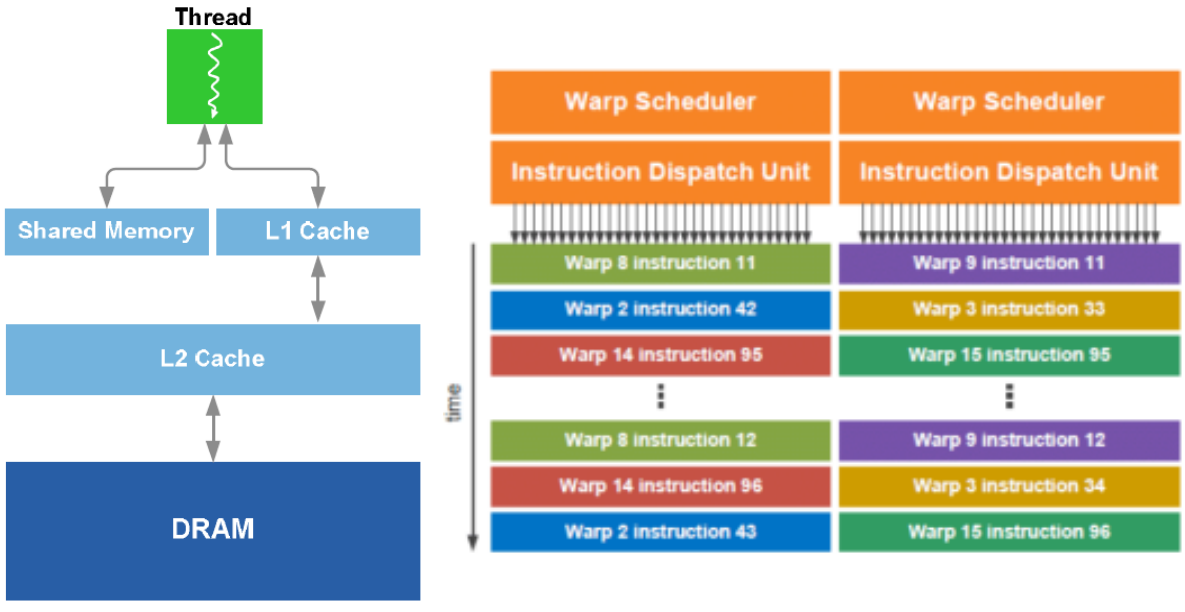
Fermi Architecture Innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

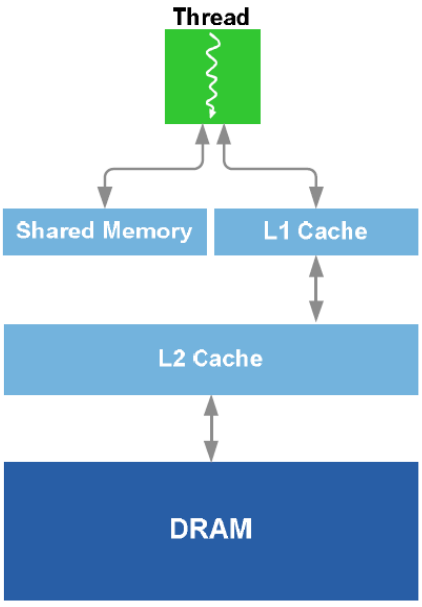
NVIDIA GPU Memory Structures

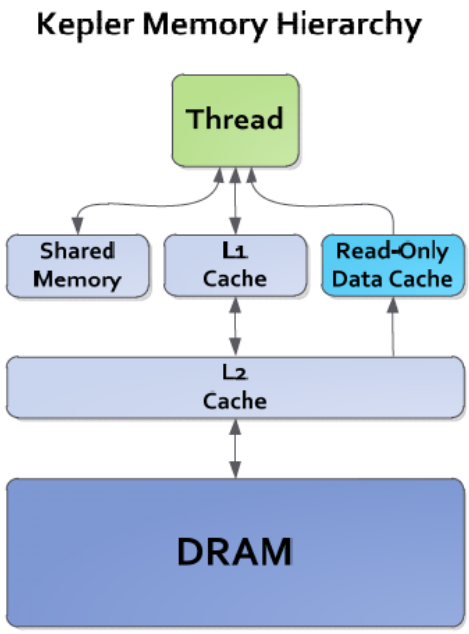
- Each SIMD Lane has private section of **off-chip DRAM**
  - "Private memory" *(Local Memory)*
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory *(Shared Memory)*
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory *(Global Memory)*
  - Host can read and write GPU memory

Fermi: Multithreading and Memory Hierarchy

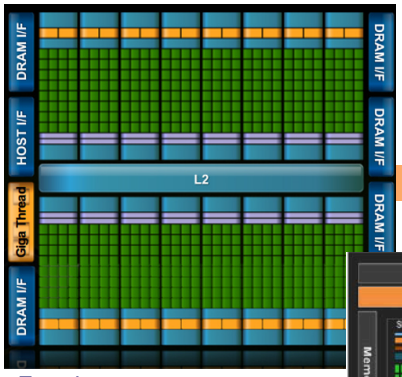


From Fermi into Kepler: The Memory Hierarchy





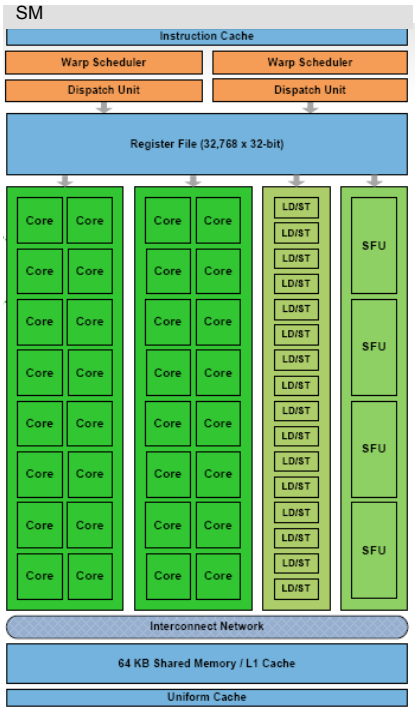
From the GF110 to the GK110 Kepler Architecture

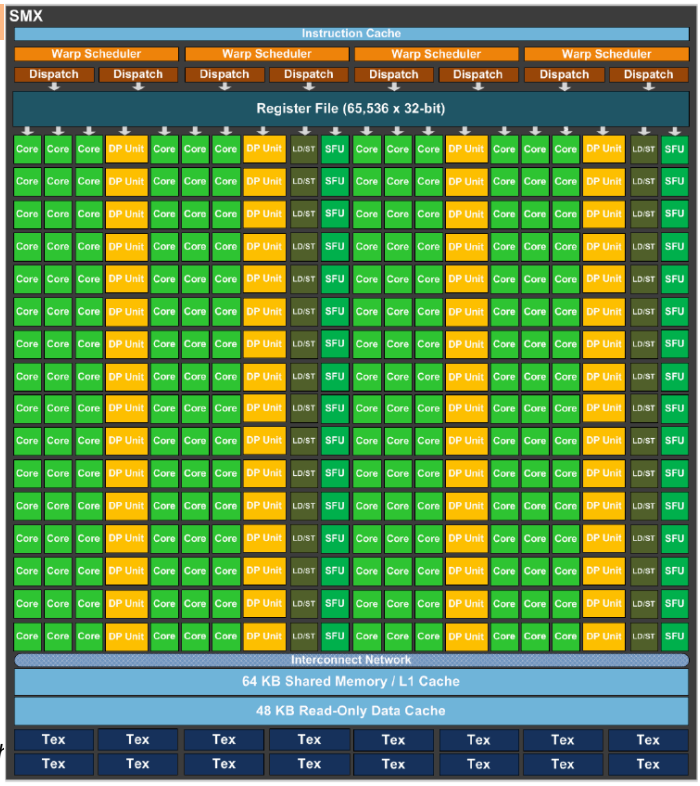


Fermi:
512 CUDA-cores

Kepler:
2880 CUDA-cores

From Fermi to Kepler core: SM and the SMX Architecture





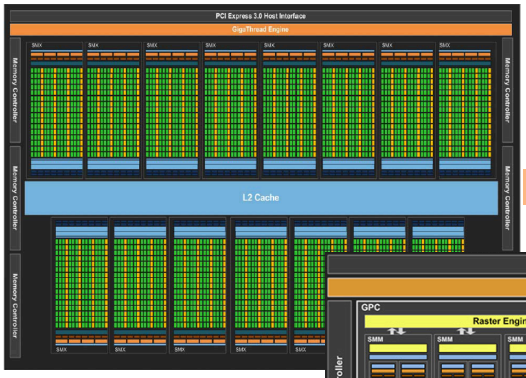
SMX:
192 CUDA-cores

Ratio **DP**unit : **SP**unit —> **1 : 3**

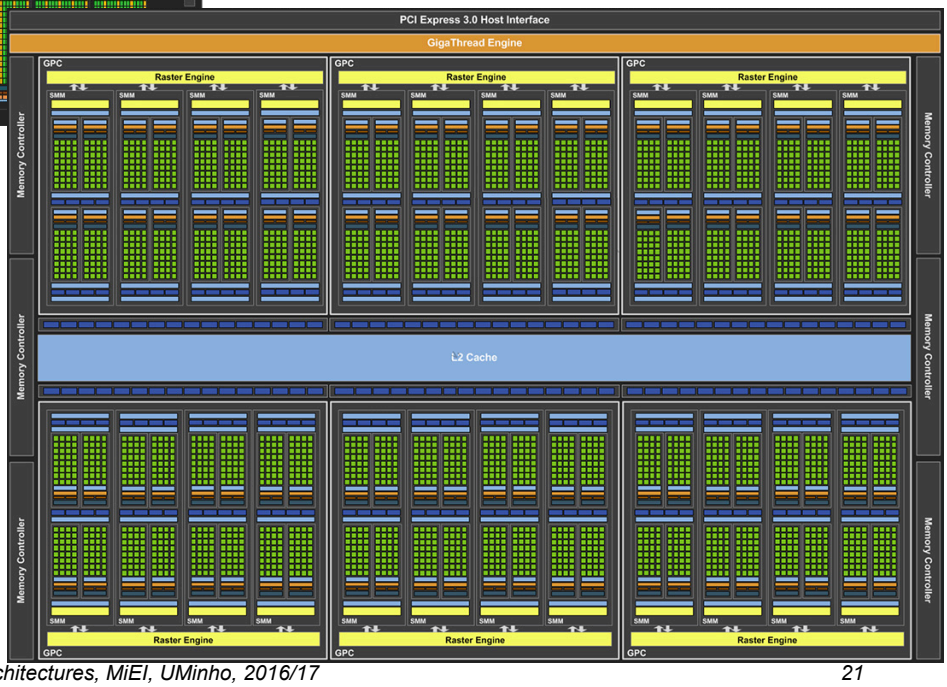
From Fermi into Kepler: Compute capabilities

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

From the GK110 to the GM200 Maxwell Architecture

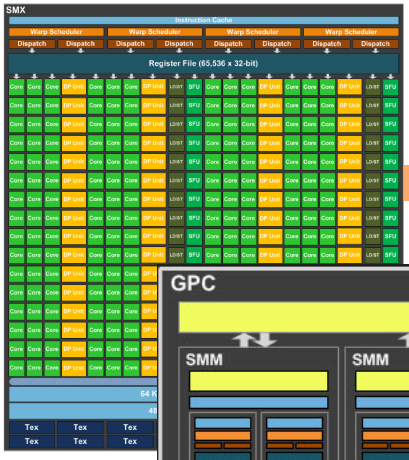


Kepler:
2880 CUDA-cores



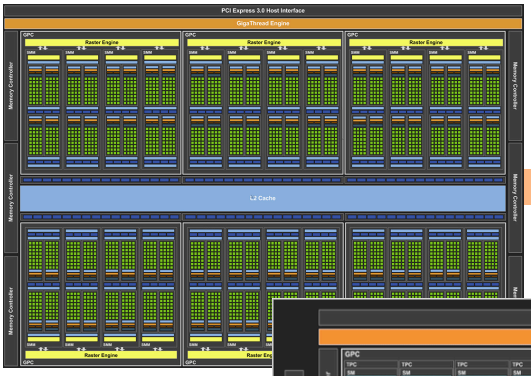
Maxwell:
3072 CUDA-cores

The move from Kepler to Maxwell : from 15 SMXs to 48 SMMs in 6 GPCs

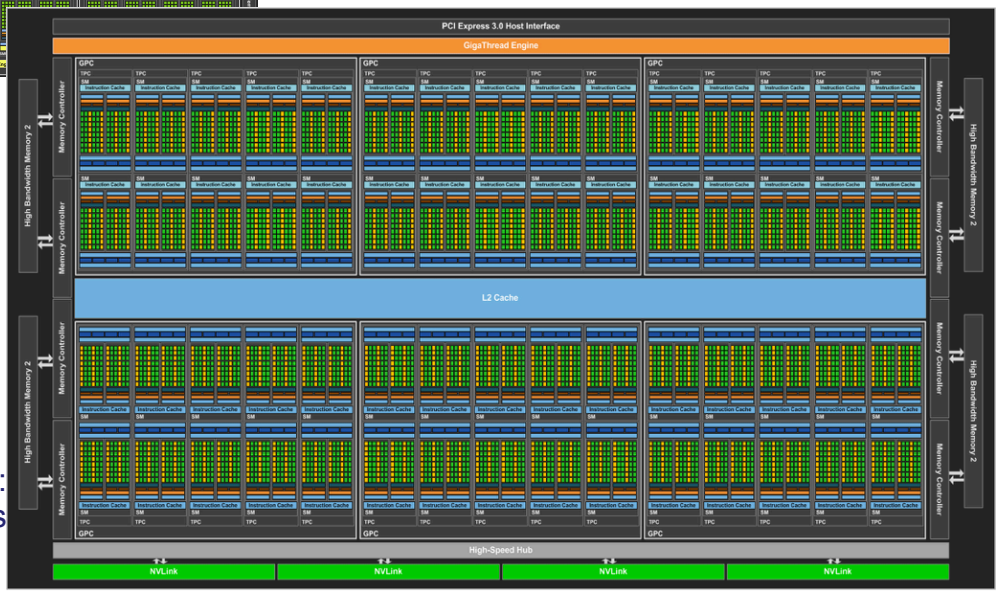


SMM: 128 CUDA-cores
Ratio **DP**unit : **SP**unit –> **1 : 32**

From the M200 to the GP100 Pascal Architecture

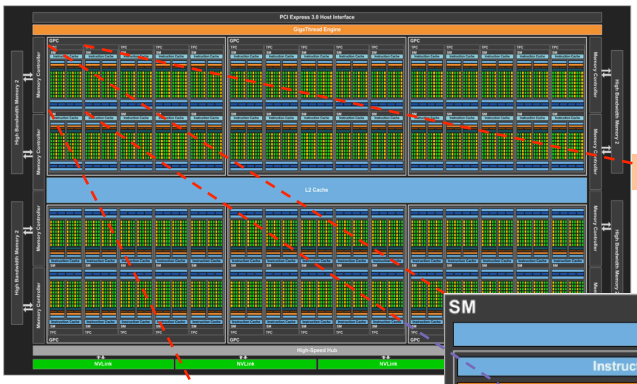


Maxwell:
3072 CUDA-cores



Pascal:
3584 CUDA-cores

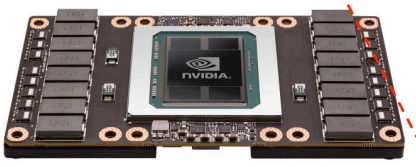
Pascal Architecture: 6x GPCs, 60 SMs



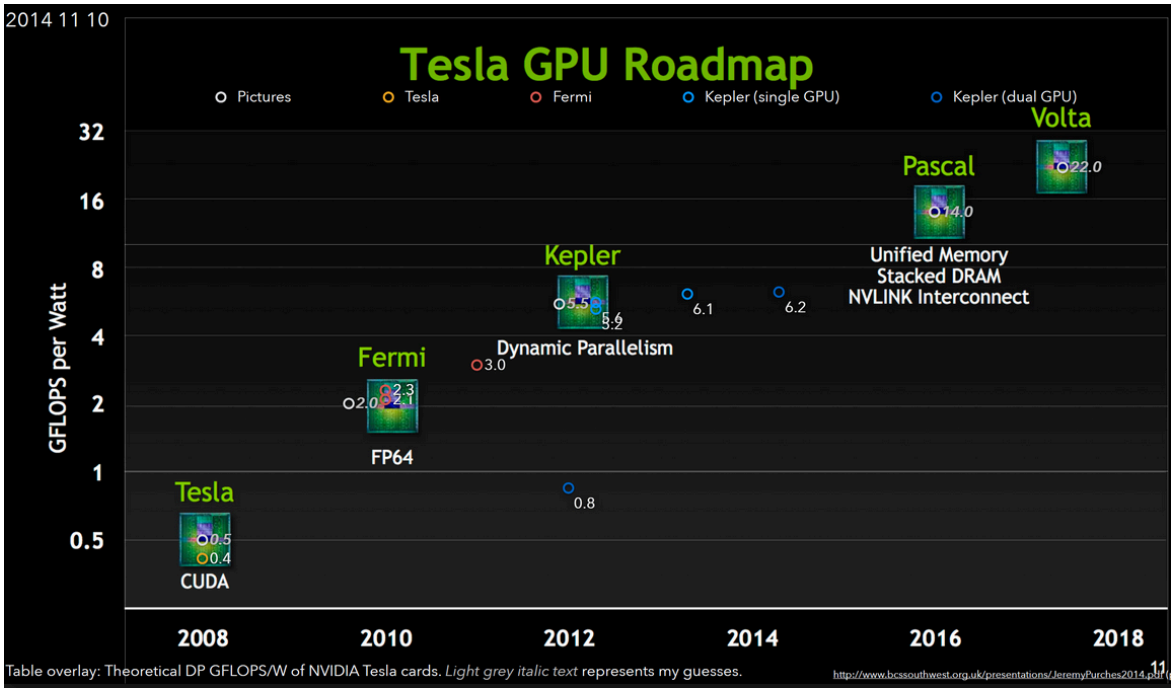
Pascal SM:
32 CUDA-cores

Ratio **DP**unit : **SP**unit —> **1 : 2**

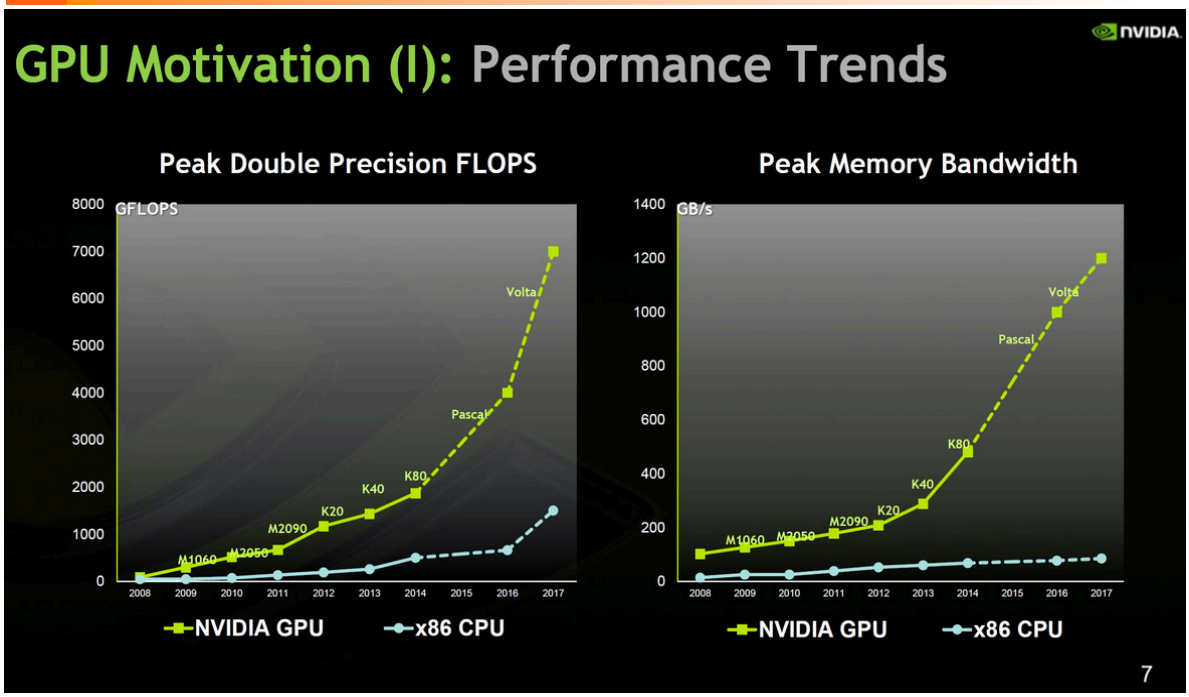
Pascal P100 w/ 16GB HBM2



Families in NVidia Tesla GPUs



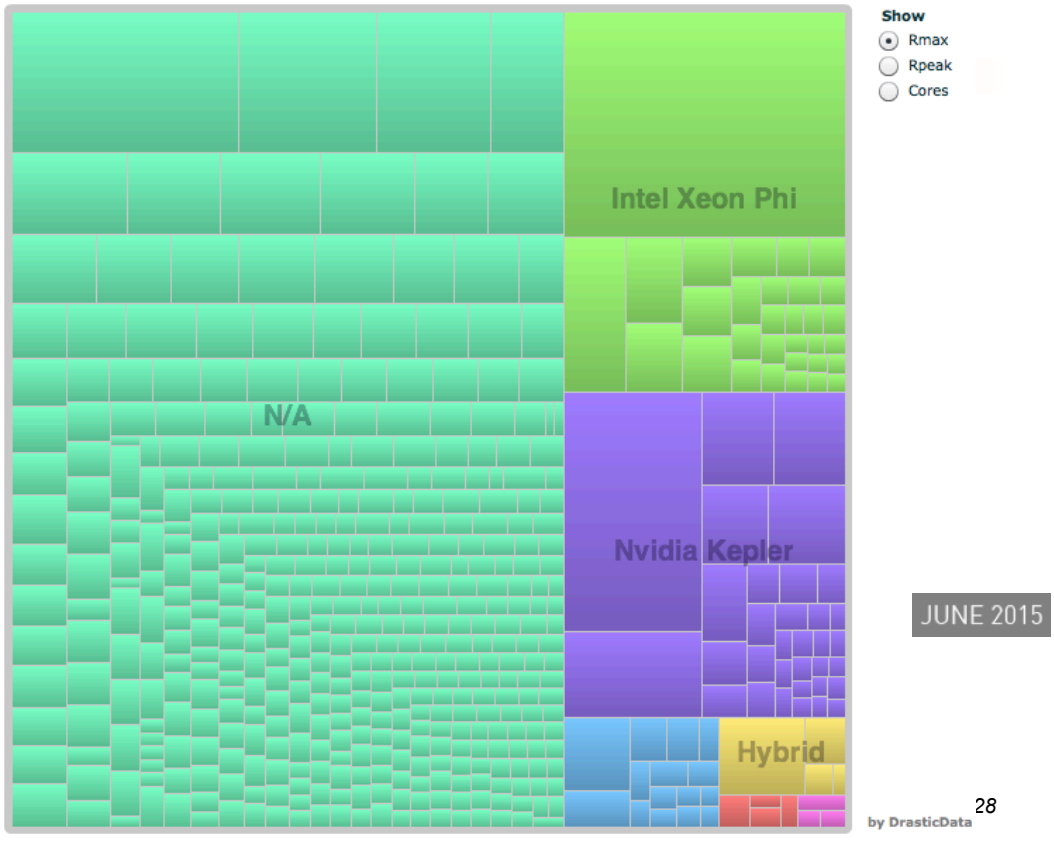
Families in NVidia Tesla GPUs

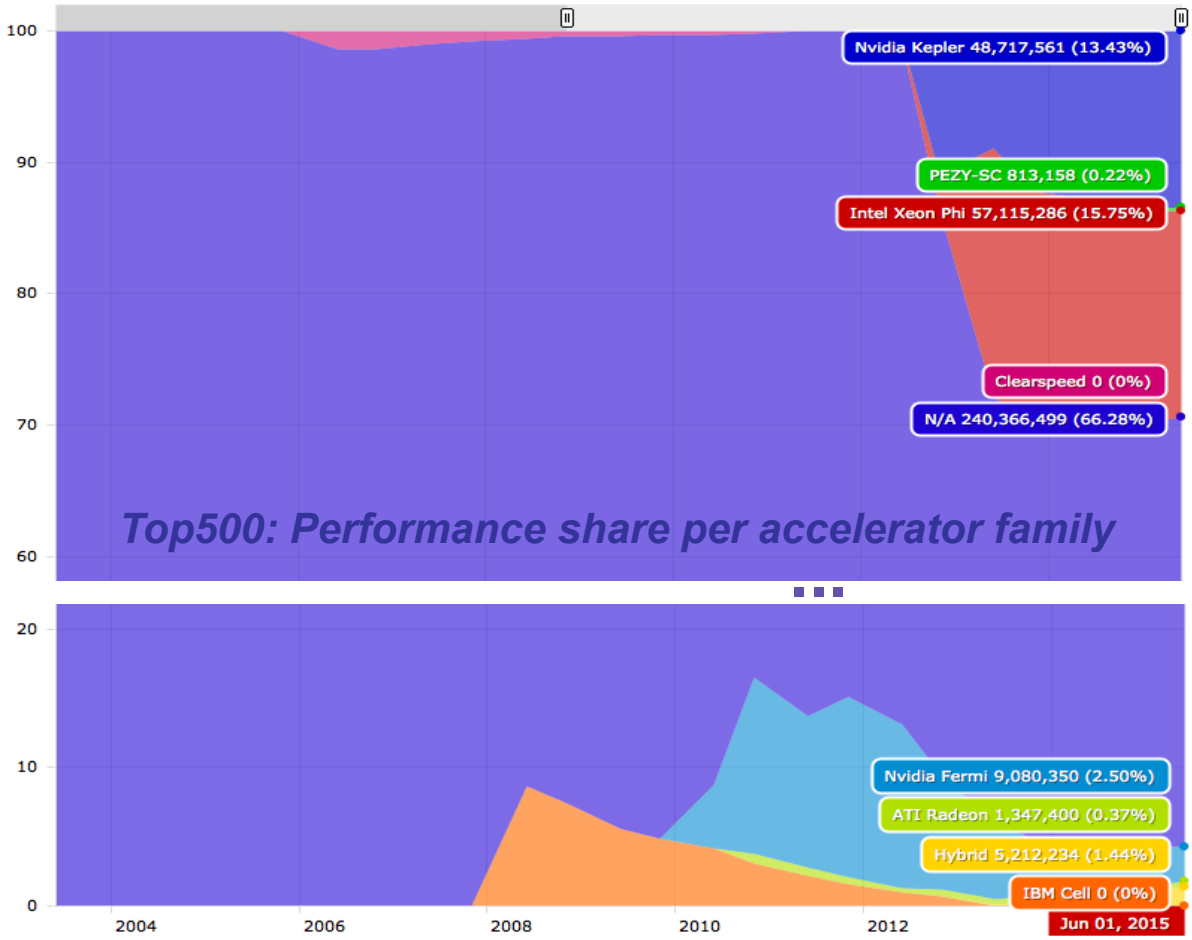


Families in NVidia GPU

Nvidia Tesla Workstation GPU Specifications Comparison					
	Full P2xx	P100	M40	K80	K40
Family	Pascal (2nd-gen)	Pascal	Maxwell (2nd-gen)	Kepler (2nd-gen)	Kepler
Architecture	N/A	GP100	GM200	GK210	GK110
Cores	3840	3584	3072	2 x 2496	2880
ROPs	N/A	N/A	96	96	48
Texture Units	240	224	192	416	240
Core Clock	N/A	1328MHz	948MHz	562MHz	745MHz
Memory Clock	N/A	1400MHz	1500MHz	2500MHz	3004MHz
Memory Bandwidth	N/A	N/A	288GB/s	480GB/s	288GB/s
Memory Bus Width	4096-bit	4096-bit	384-bit	2x 384-bit	384-bit
Memory Type	HBM2	HBM2	GDDR5	GDDR5	GDDR5
Memory Size	16GB or higher	16GB	24GB	2 x 12GB	12GB
Die Size	N/A	610mm^2	601mm^2	561mm^2	551mm^2
Transistors	N/A	15.3 billion	8 billion	2 x 7.08 billion	7.08 billion
Register File Size / SM	256KB	256KB	256KB	512KB	256KB
L2 Cache	N/A	4MB	3MB	1.5MB	1.5MB
TDP	N/A	300W	250W	300W	235W
Manufacturing Process	TSMC 16nm	TSMC 16nm	TSMC 28nm	TSMC 28nm	TSMC 28nm
Release Date	N/A	Jul-16	Nov-15	Nov-14	Nov-13

Top500: Accelerator distribution over all 500 systems





The CUDA programming model

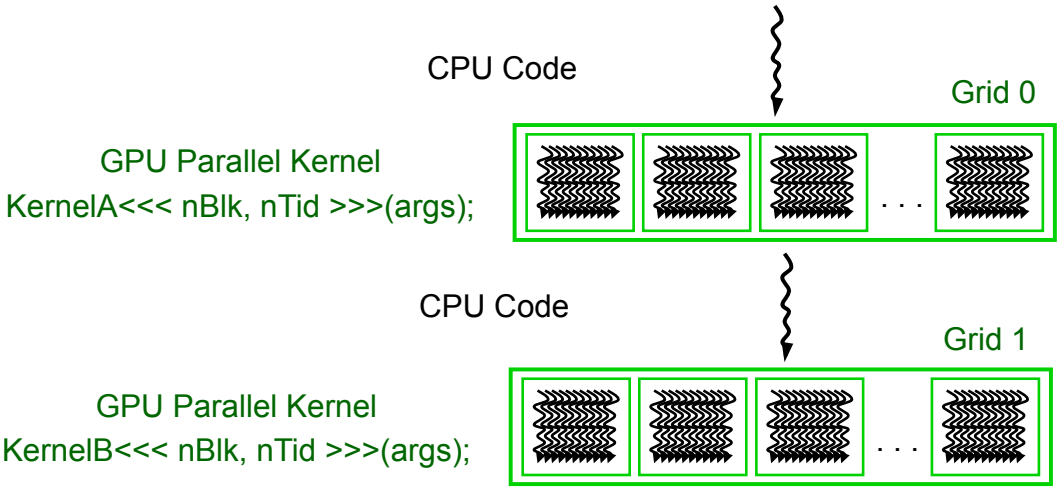
- ***C**ompute **U**nified **D**evice **A**rchitecture*
- CUDA is a recent programming model, designed for
  - a multicore CPU ***host*** coupled to a many-core ***device***, where
  - *devices* have wide SIMD/SIMT parallelism, and
  - the *host* and the *device* do not share memory
- CUDA provides:
  - a thread abstraction to deal with SIMD
  - synchr. & data sharing between small groups of threads
- CUDA programs are written in C with extensions
- OpenCL inspired by CUDA, but hw & sw vendor neutral
  - programming model essentially identical

CUDA Devices and Threads

- A compute device
  - is a coprocessor to the CPU or host
  - has its own DRAM (device memory)
  - runs many threads in parallel
  - is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads - **SIMT**
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - very little creation overhead, requires LARGE register bank
  - GPU needs 1000s of threads for full efficiency
    - multi-core CPU needs only a few

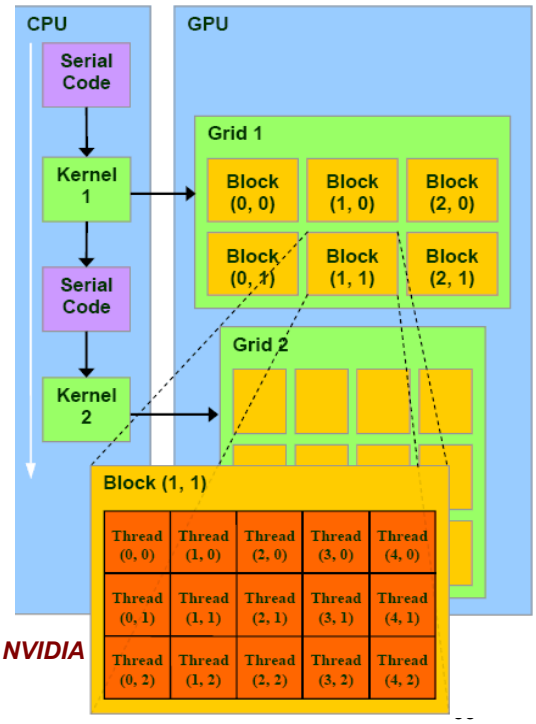
CUDA basic model: Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



Programming Model: SPMD + SIMT/SIMD

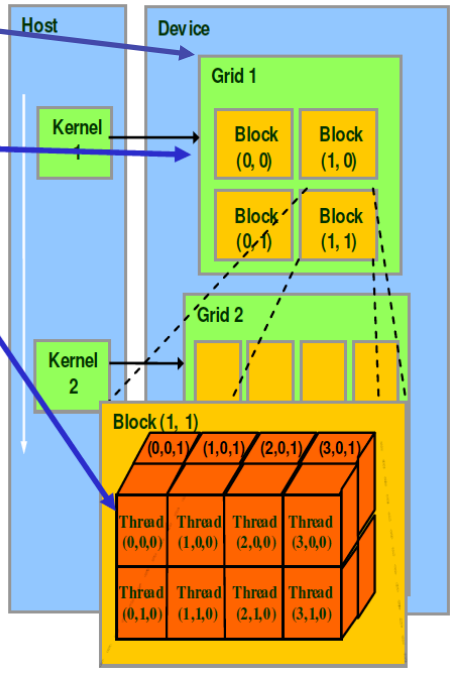
- Hierarchy
  - Device => Grids
  - Grid => Blocks
  - Block => Warps
  - Warp => Threads
- Single kernel runs on multiple blocks (SPMD)
- Threads within a warp are executed in a lock-step way called single-instruction multiple-thread (SIMT)
- Single instruction are executed on multiple threads (SIMD)
  - Warp size defines SIMD granularity (32 threads)
- Synchronization within a block uses shared memory



Courtesy NVIDIA

The Computational Grid: Block IDs and Thread IDs

- A kernel runs on a computational grid of thread blocks
  - Threads share global memory
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- A thread block is a batch of threads that can cooperate by:
  - Sync their execution w/ barrier
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate



Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-16 multithreaded SIMD processors

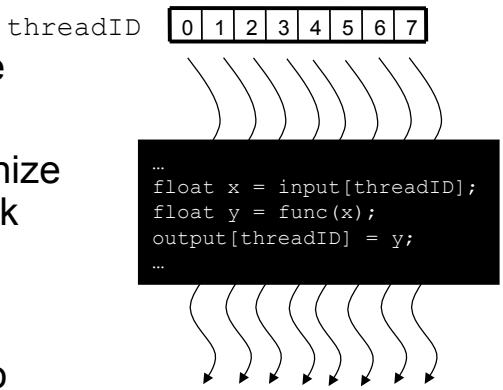
Terminology *(and in NVidia)*

- *Threads of SIMD instructions (**warps**)*
  - Each has its own IP (up to 48/64 per SIMD processor, Fermi/Kepler)
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Threads are organized into blocks & executed in groups of 32 threads *(**thread block**)*
    - Blocks are organized into a grid
- The thread block scheduler schedules blocks to SIMD processors *(**Streaming Multiprocessors**)*
- Within each SIMD processor:
  - 32 SIMD lanes *(**thread processors**)*
  - Wide and shallow compared to vector processors

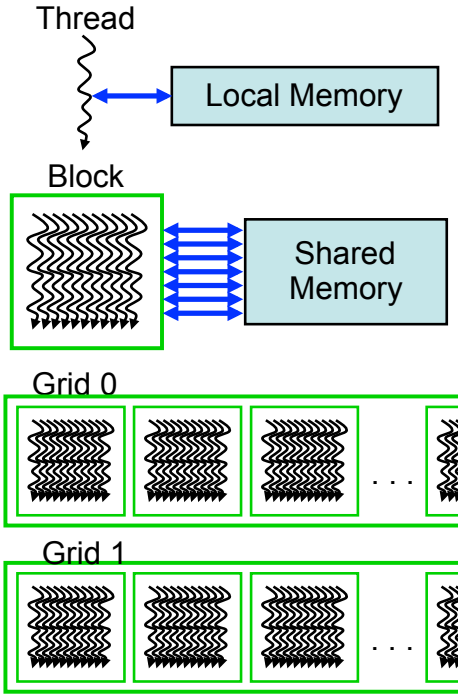
CUDA Thread Block

- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have thread id numbers within Block
- Thread program uses thread id to select work and address shared data

CUDA Thread Block



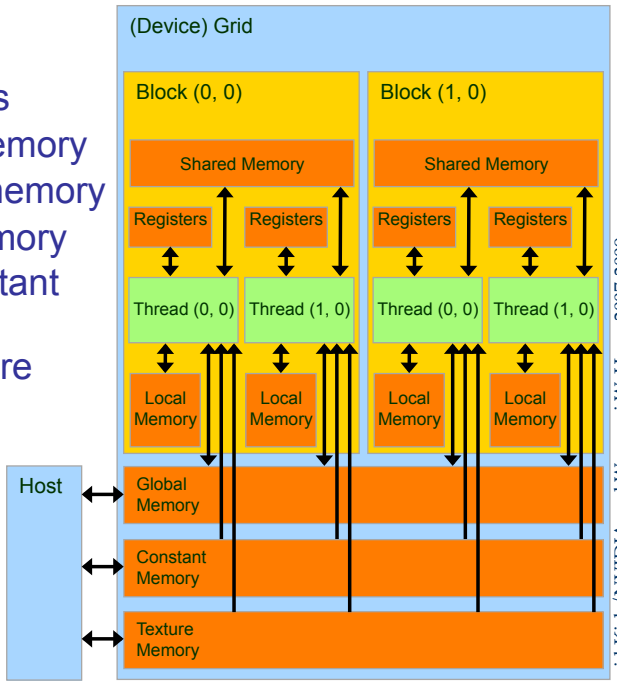
Parallel Memory Sharing



- Local Memory: **per-thread**
  - Private per thread
  - Auto variables, register spill
- Shared Memory: **per-block**
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: **per-application**
  - Shared by all threads
  - Inter-Grid communication

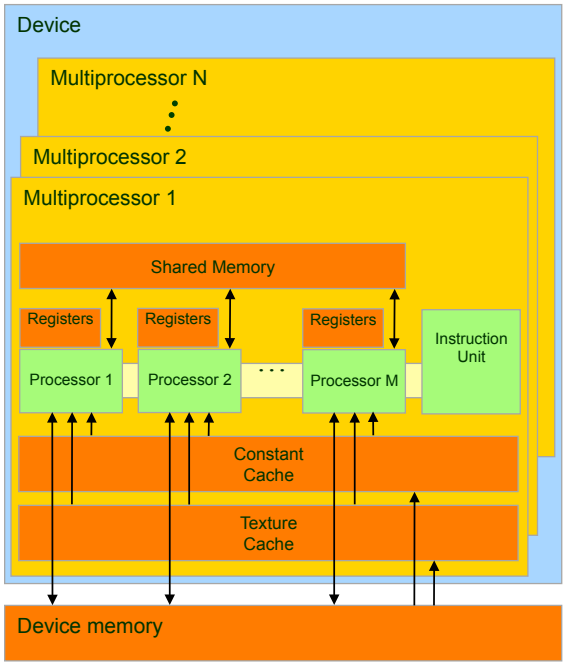
CUDA Memory Model Overview

- 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



Hardware Implementation: Memory Architecture

- Device memory (DRAM)
  - Slow (2~300 cycles)
  - Local, global, constant, and texture memory
- On-chip memory
  - Fast (1 cycle)
  - Registers, shared memory, constant/texture cache

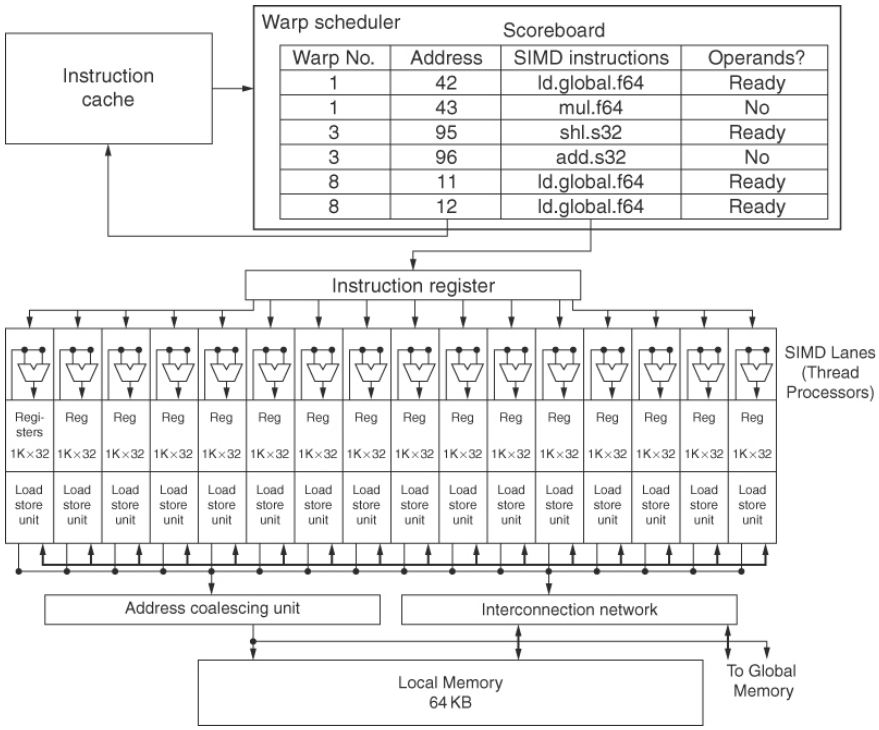


Courtesy NVIDIA

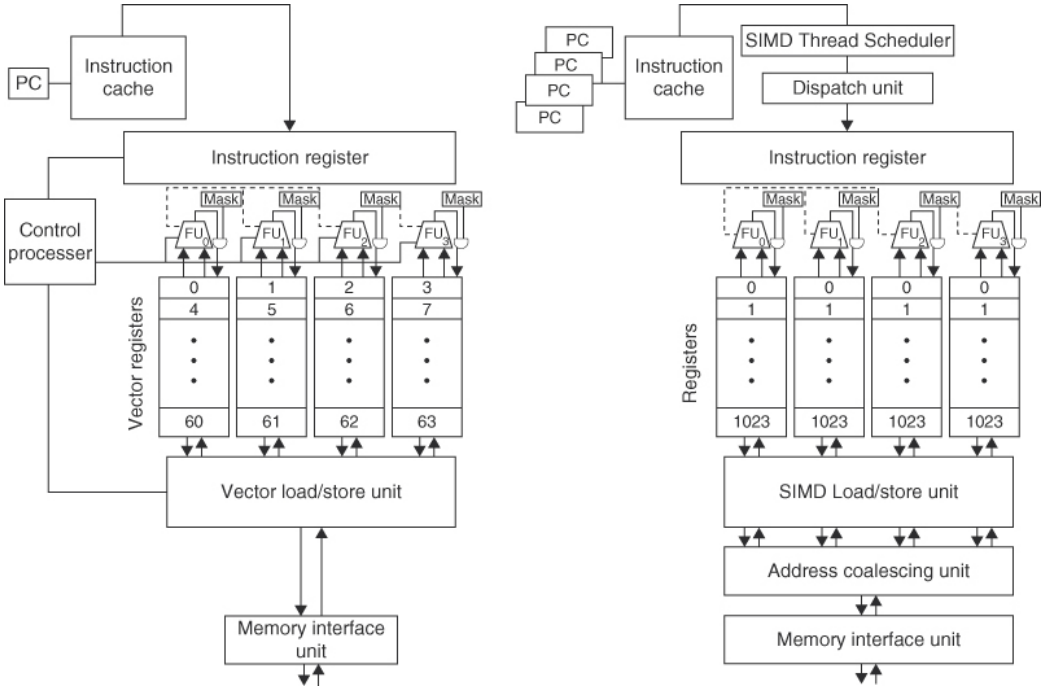
NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of **off-chip DRAM**
  - "Private memory" *(Local Memory)*
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory *(Shared Memory)*
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory *(Global Memory)*
  - Host can read and write GPU memory

Example



Vector Processor *versus* CUDA core



Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer