Universidad Autónoma de Madrid

excelencia UAM CSIC

Escuela Politécnica Superior

Tema 3: Paralelismo de datos a gran escala con GPU

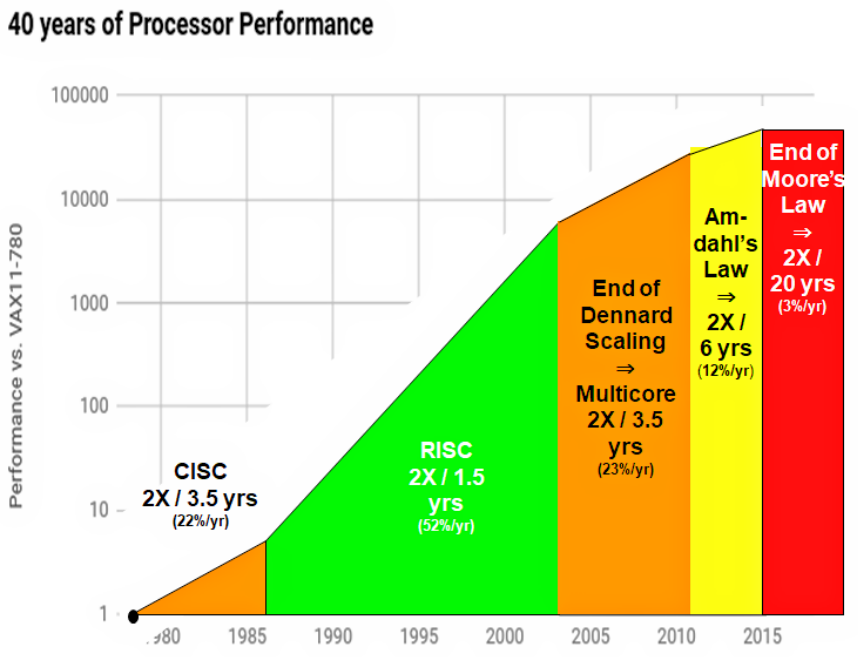
Francisco J. Gómez Arribas

Tema 3: Contenidos

- 3.1 Sistemas con coprocesadores GPU.
  - 3.1.1 SPMD en una arquitectura manycore. Modelo SIMT.
  - 3.1.2 Programación de propósito general con GPU (GPGPU).
  - GPU vs FPGA
- 3.2 Arquitectura y Modelo de Ejecución.
  - 3.2.1 Entorno de programación CUDA.
  - 3.2.2 Modelo de Programación.
  - 3.2.3 API de CUDA y Jerarquía de memoria.

Necesidad de Coprocesadores : Evolución del rendimiento

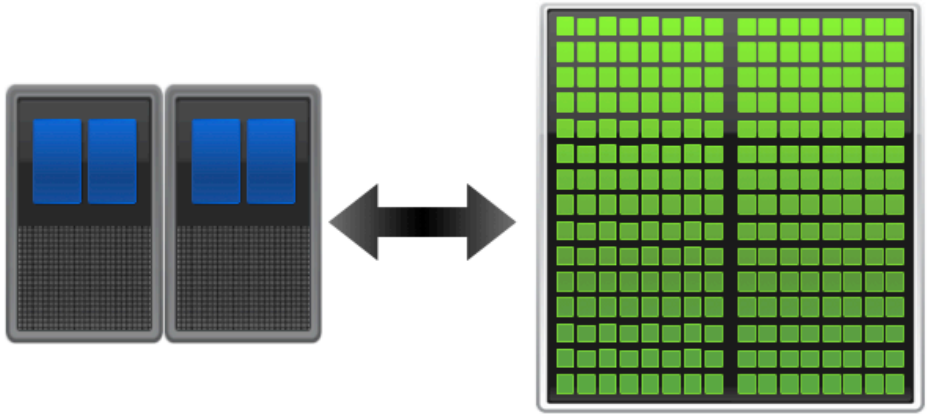
- El rendimiento de los procesadores de propósito general se está estancando:



Se necesita nuevas tendencias para dar soluciones:

- Coprocesadores: Aceleradores con arquitecturas específicas para cada dominio.
- Tecnologías más disruptivas: procesadores cuánticos, neurocomputación.

Heterogeneous Parallel Computing



Multicore CPU

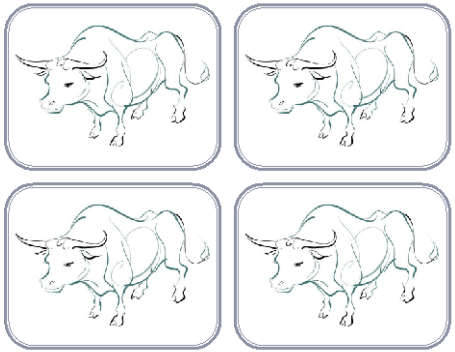
Fast Serial Processing

Manycore GPU

Scalable Parallel Processing

(GPGPU – general purpose graphics processing unit)

Multicore versus Manycore

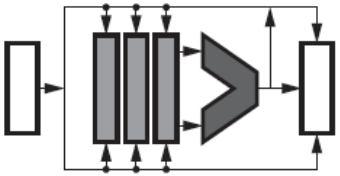
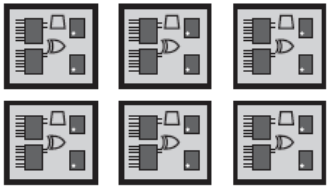
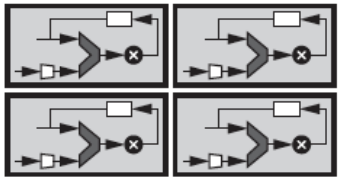


Multicore

Manycore

- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: Flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

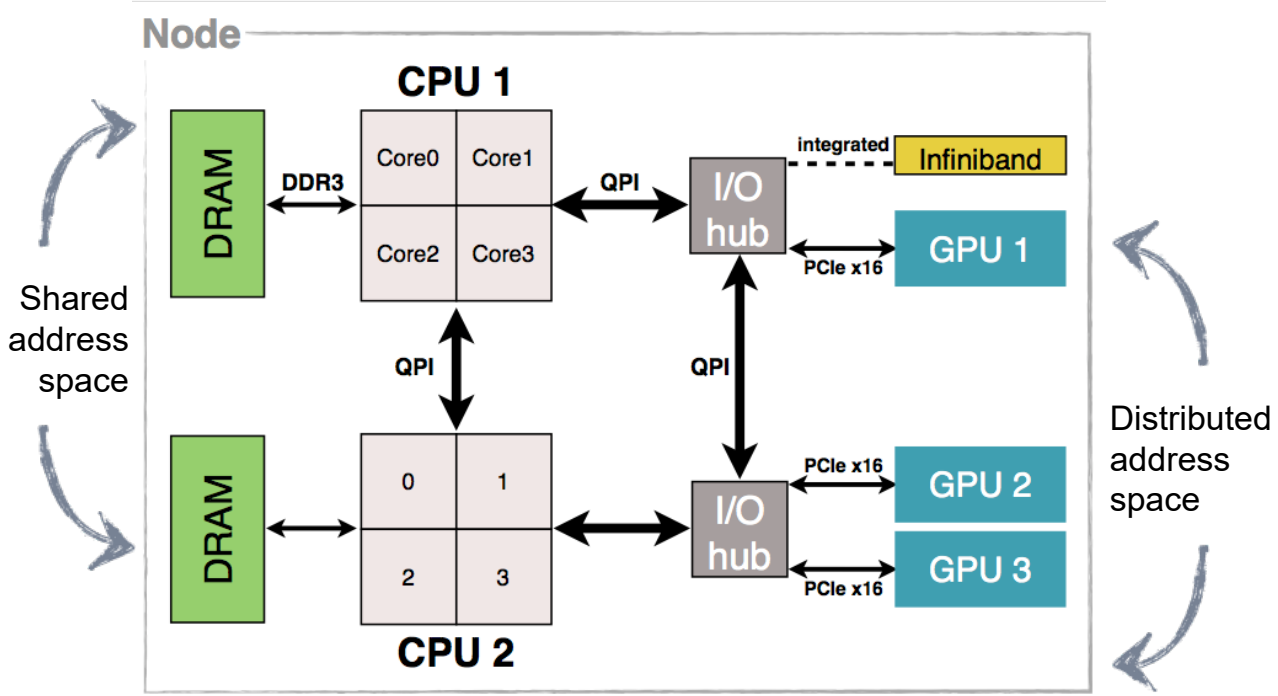
Coprocesadores Aceleradores

CPU	FPGA	GPU
Scalar Processing	**Adaptable Hardware**	**Vector Processing** *(e.g., GPU, DSP)*
		
Complex Algorithms and Decision Making	Processing of Irregular Data Structures *Genomic Sequencing*	Domain-specific Parallelism
	Latency Critical Workloads *Real-Time Control*	Signal Processing *Complex Math, Convolutions*
	Sensor Fusion *Pre-processing, Programmable I/O*	Video and Image Processing

WP505_02_092918

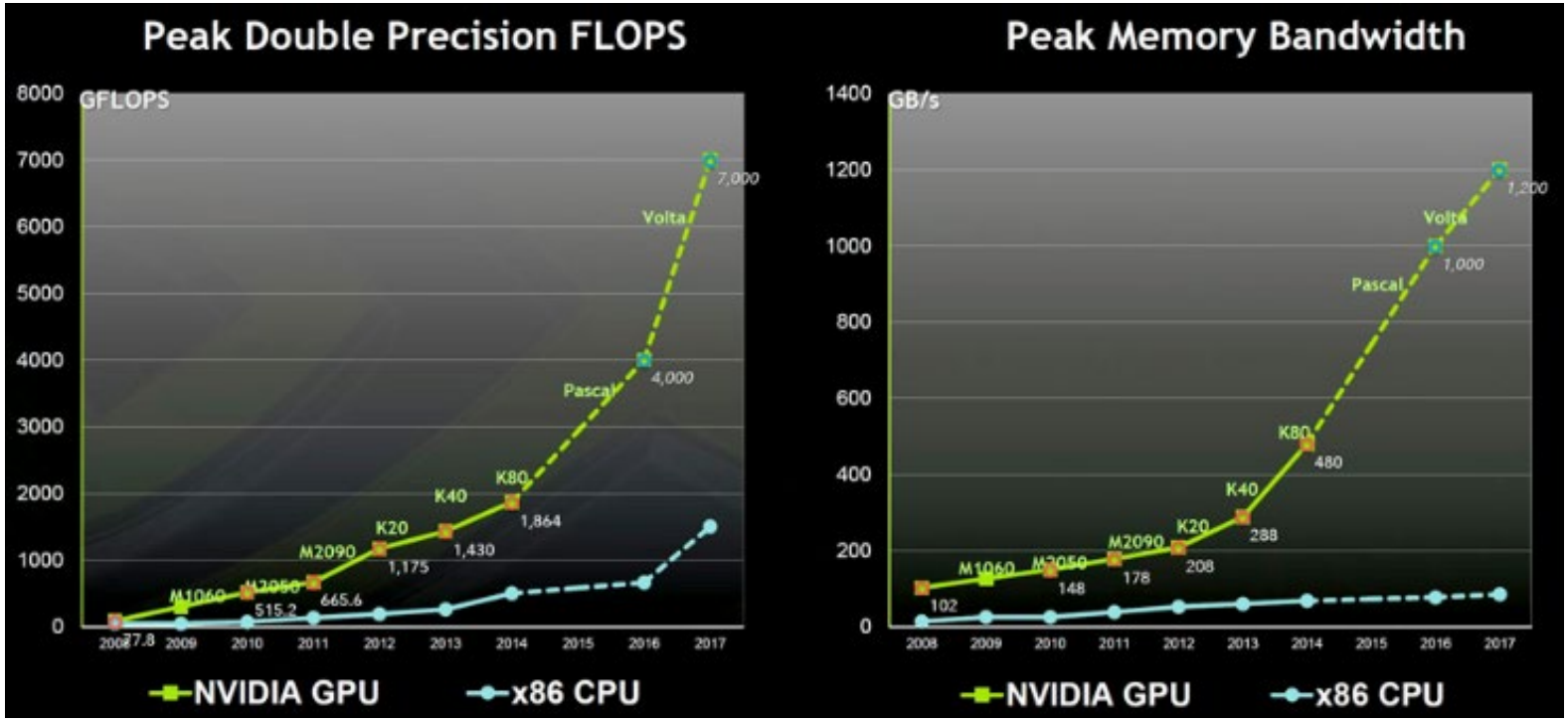
Multicore CPUs Connected to Manycore GPUs

- Typically, GPU devices are accessed over PCI



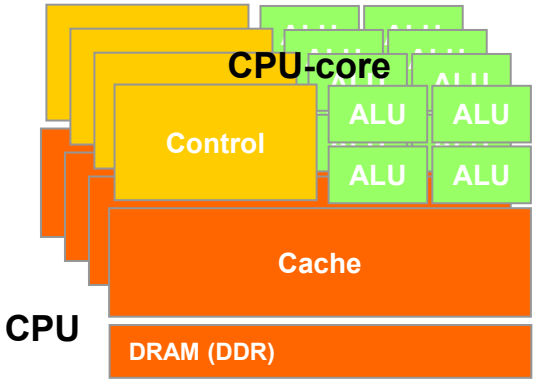
CPU vs GPUs

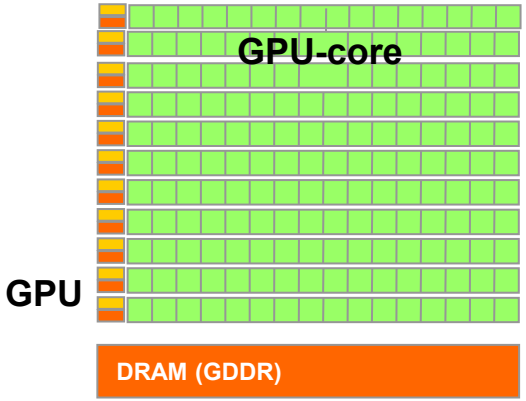
Performance Trends



CPU vs GPUs

- The GPU is specialized for compute-intensive, highly data parallel computation (what graphics rendering is about)
- So, more transistors can be devoted to data processing rather than data caching and flow control

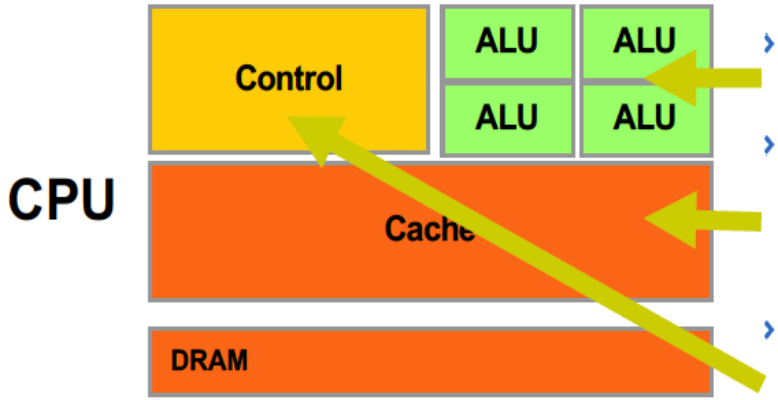




Different design philosophies

CPU: A few out-of-order cores vs GPU: Many in-order cores

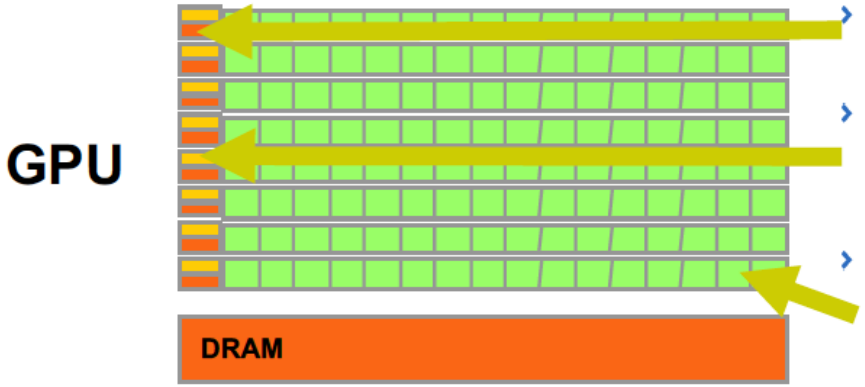
CPU: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- 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

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

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
  - Threading logic
  - Thread state

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

Aceleradores/Coprocesadores GPU

CPU **vs** **GPU**

Optimized for **low latency**

- \+ Large main memory
- \+ Fast clock rate
- \+ Large caches
- \+ Branch prediction
- \+ Powerful ALU
- – Relatively low memory bandwidth
- – Cache misses costly
- – Low performance per watt

Optimized for **high throughput**

- \+ High bandwidth main memory
- \+ Latency tolerant (parallelism)
- \+ More compute resources
- \+ High performance per watt
- – Limited memory capacity
- – Low per-thread performance
- – Extension card

Why Heterogeneity?

- Different goals produce different designs
  - Manycores assumes workload is highly parallel
  - Multicore must be good at everything, parallel or not
- Multicore: minimize latency experienced by 1 thread
  - Lots of big on-chip caches
  - Extremely sophisticated control
- Manycore: maximize throughput of all threads
  - Lots of big ALUs
  - Multithreading can hide latency… so skip the big caches
  - Simpler control, cost amortized over ALUs via SIMD

GPU vs other coprocessors

A GPU has **thousands of cores** designed for efficient execution of mathematical functions. The Tesla V100, contains:

- 5,120 CUDA cores for single-cycle multiply-accumulate operations and
- 640 tensor cores for single-cycle matrix multiplication.
- It has been flaunting massive processing power for target applications such as video processing, image analysis, signal processing and more.

Advantages

- GPU has a wider and mature ecosystem
- Offers an efficient platform for this new era

Disadvantages

- With the evolving data needs, the GPU architecture needs to evolve to stay relevant

Programación Coprocesadores GPU

Al diseñar el software y programar las GPU, los aspectos de mayor relevancia que ha de satisfacer nuestro código son:

- Disponer de abundantes paralelismo de grano fino.
- Maximizar el trabajo de la GPU.
- Minimizar la divergencia en la ejecución.
- Minimizar la divergencia en la memoria.
- Contar con una colección de tareas en paralelo coordinadas.

https://blog.ikoula.com/es/5-puntos-que-explican-porque-el-GPU-es-mejor-para-big-data-y-IA

Review of GPU Architecture

Streaming Multiprocessors (SM)

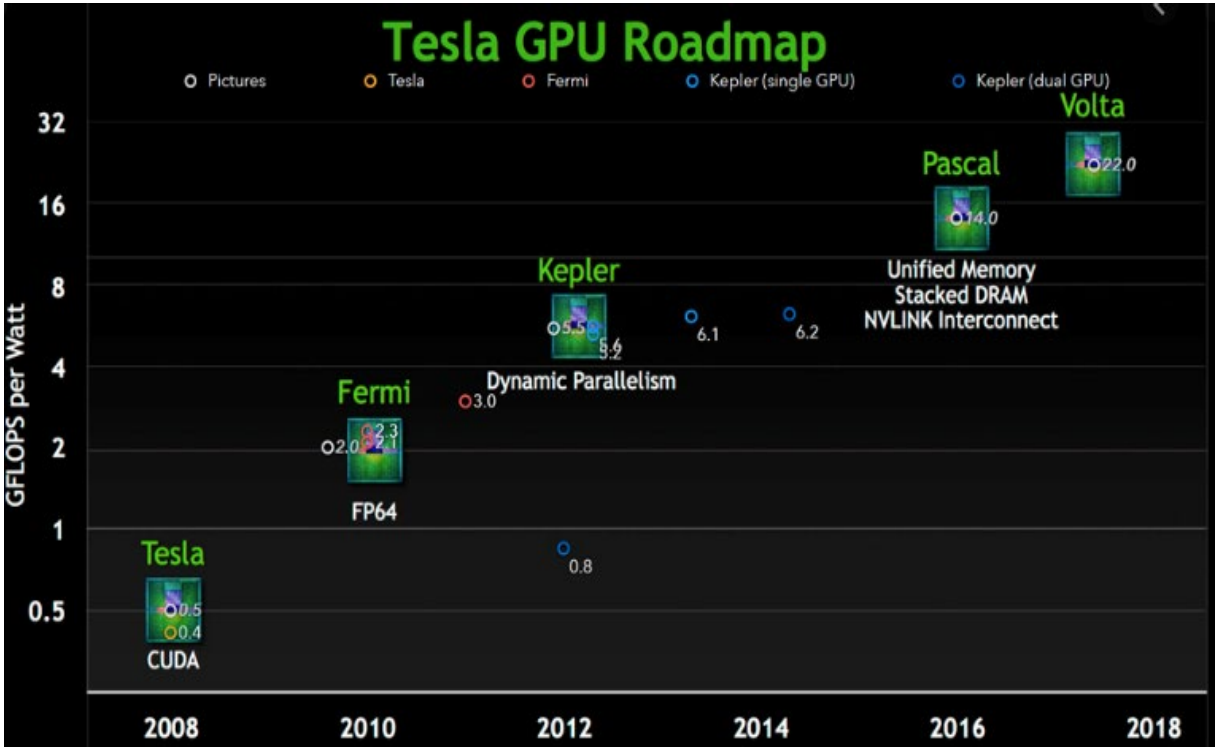
- Compute Units (CU)

Streaming Processors (SP) or CUDA cores

- Vector lanes

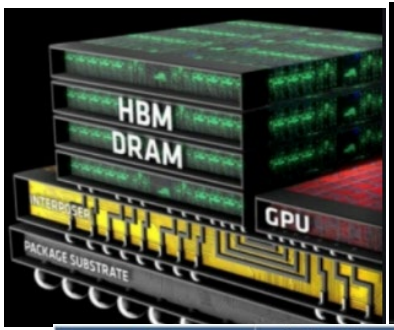
Number of SMs x SPs

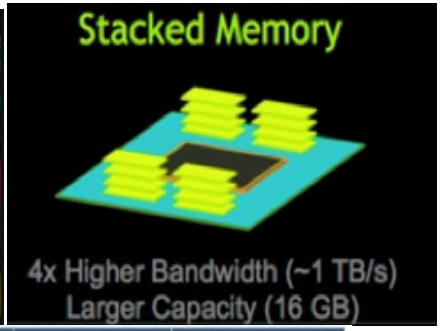
- Tesla (2007): 30 x 8
- Fermi (2010): 16 x 32
- Kepler (2012): 15 x 192
- Maxwell (2014): 24 x 128
- Pascal (2016): 56 x 64
- Volta (2018): 80 x 64



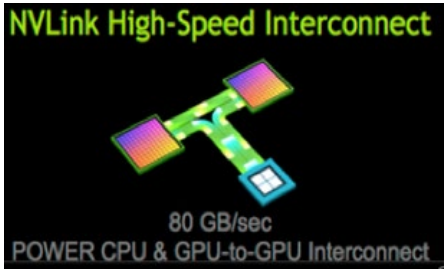
GPU: Improving Features

HBM **High Bandwidth Memory**

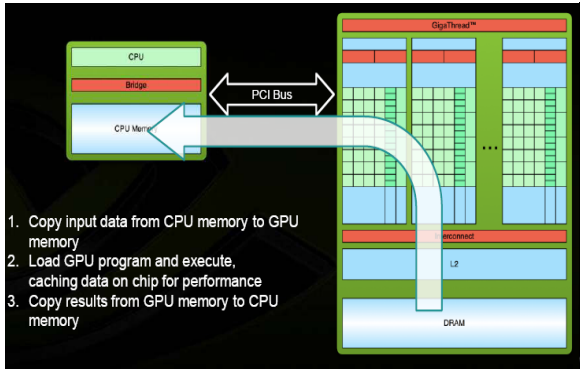


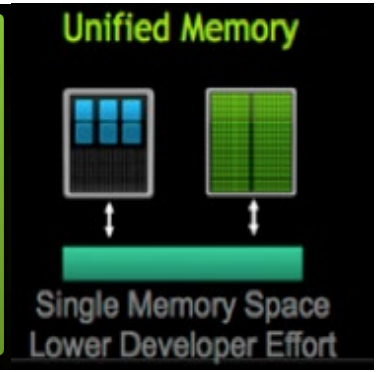


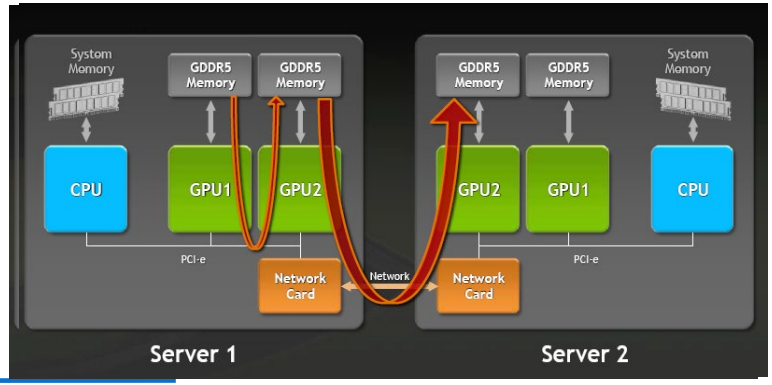
Item	DDR3 (x8)	GDDR5 (x32)	4-Hi HBM (x1024)
I/O	8	32	1024
Prefetch (Per IO)	8	8	2
Access Granularity (=I/O x Prefetch)	8Byte	32Byte	256Byte
Max. Bandwidth	2GB/s	28GB/s	128~256GB/s



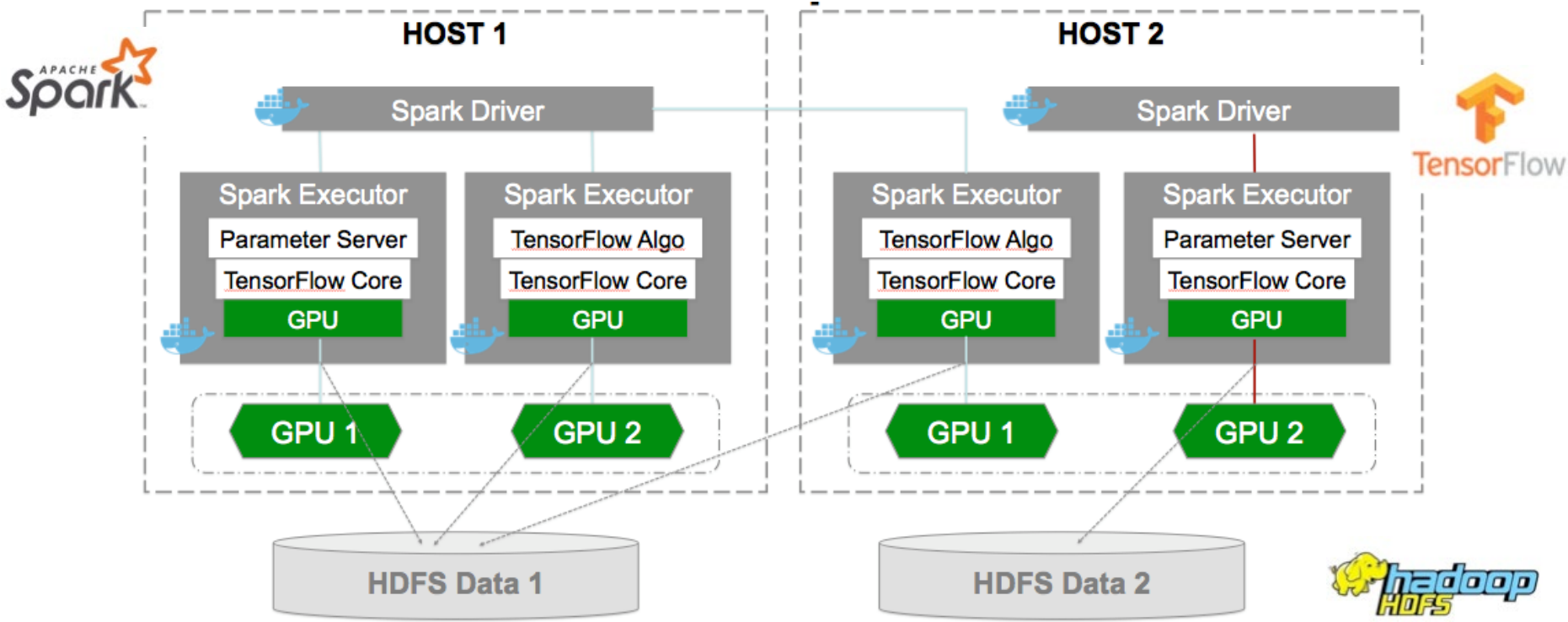
Full GPUDirect



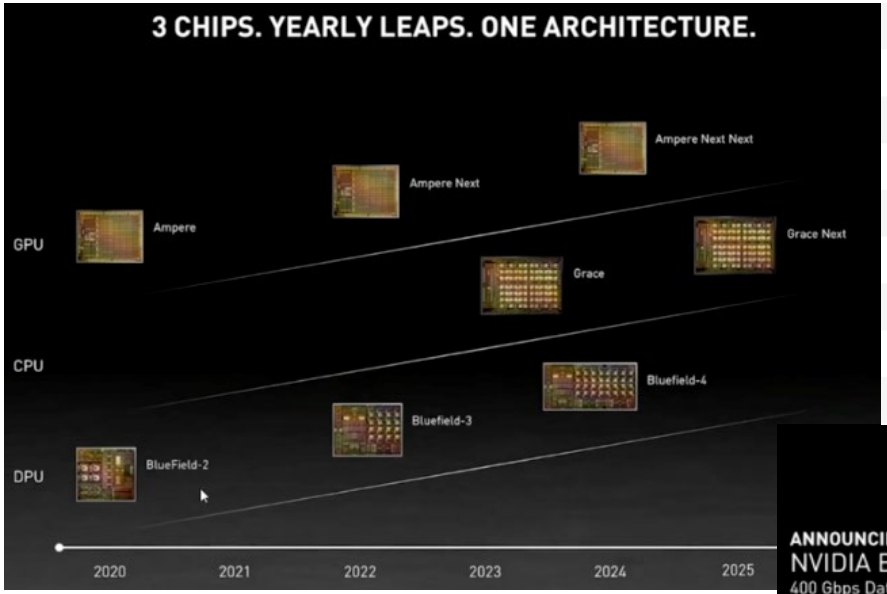




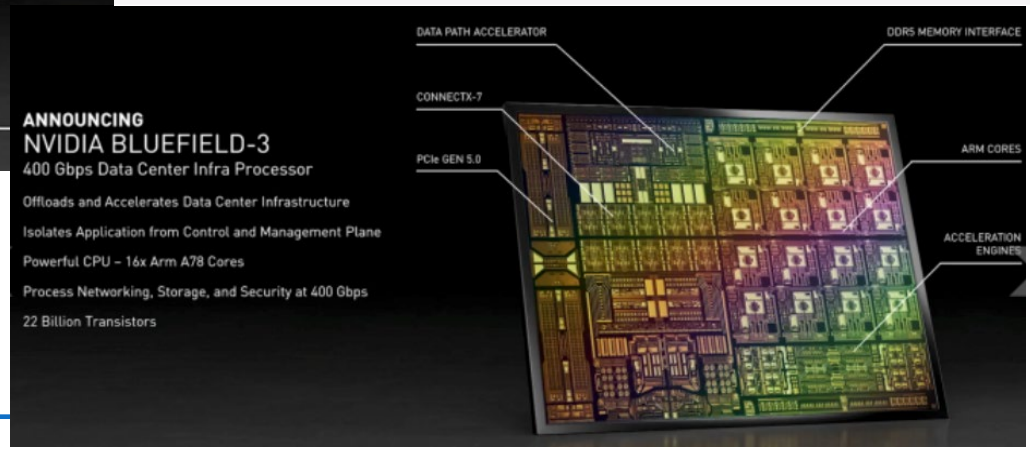
Deep Learning with TensorFlow and Spark



Roadmap of GPU Architectures



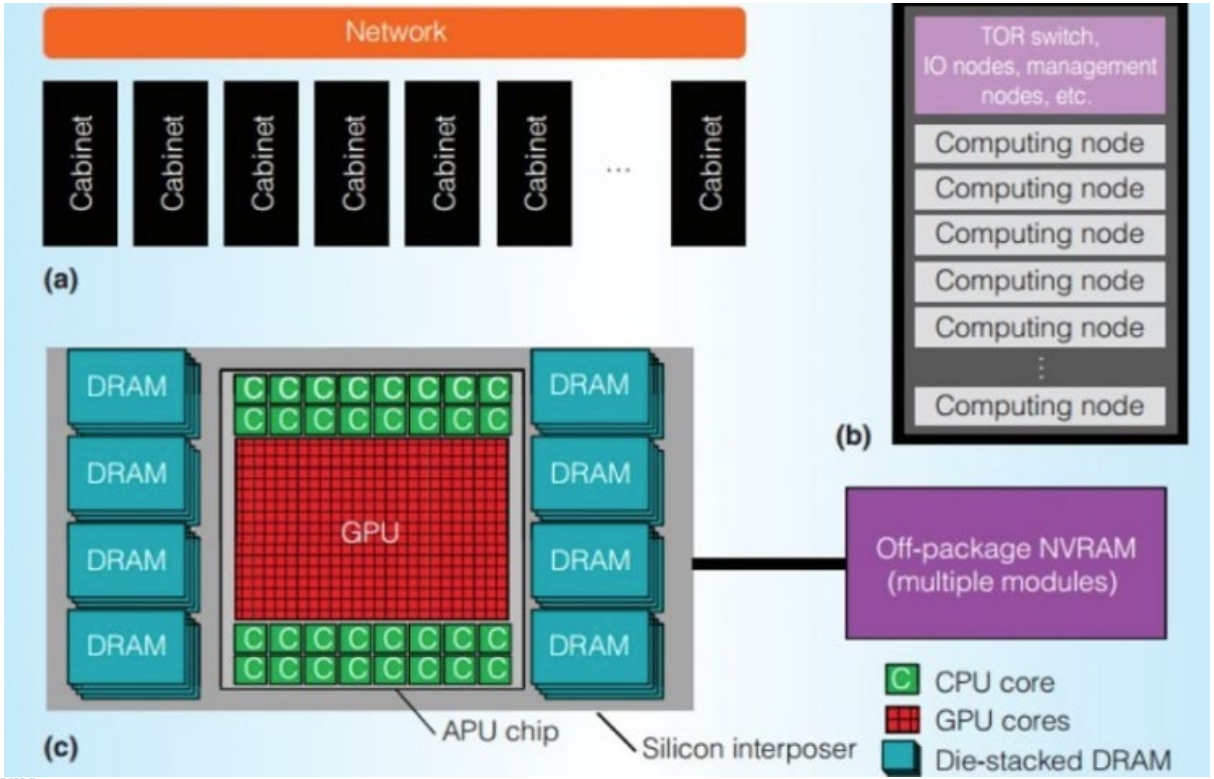
GPUs Nvidia Tesla con Ampere vs Volta vs Maxwell vs Kepler			
Modelo	**CUs / SMs**	**CUDA Cores**	**Memoria**
Modelo Ampere Filtrado #1	124	7936	32GB HBM2E
Modelo Ampere Filtrado #2	118	7552	24GB HBM2E
Modelo Ampere Filtrado #3	108	6912	48GB HBM2E
Tesla V100 (Volta)	80	5120	32GB HBM2
Tesla P100 (Pascal)	56	3584	16GB HBM2
Tesla M40 (Maxwell)	24	3072	12GB GDDR5
Tesla K40 (Kepler)	15	2880	12GB GDDR5



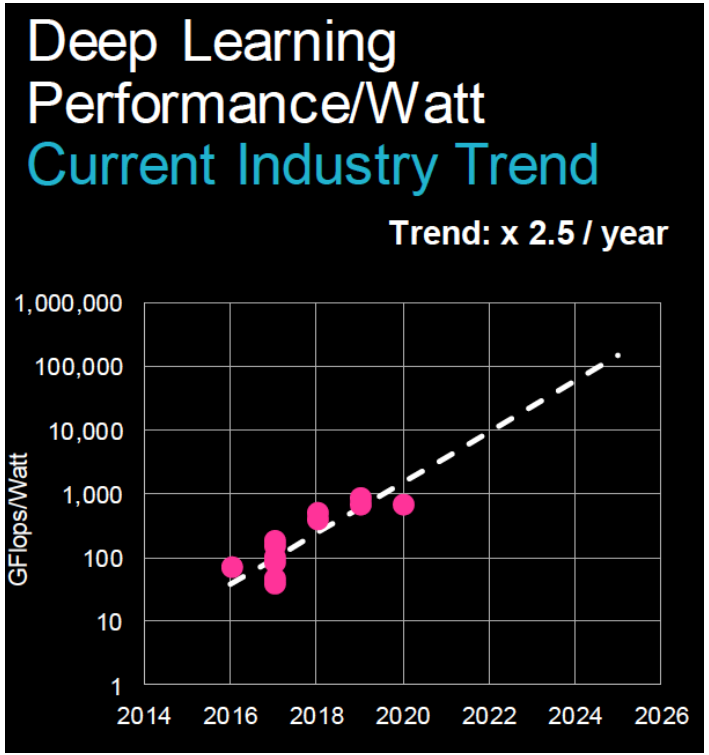
Unidad de procesamiento de datos (DPU) NVIDIA® BlueField® para CPD libera los núcleos de CPU del host.

CPU AMD: Adaptar el procesador a la computación

AMD Exascale Heterogeneous Processor: CPU, GPU y memoria HBM2 en una CPU



GPU and FPGA for Deep Learning



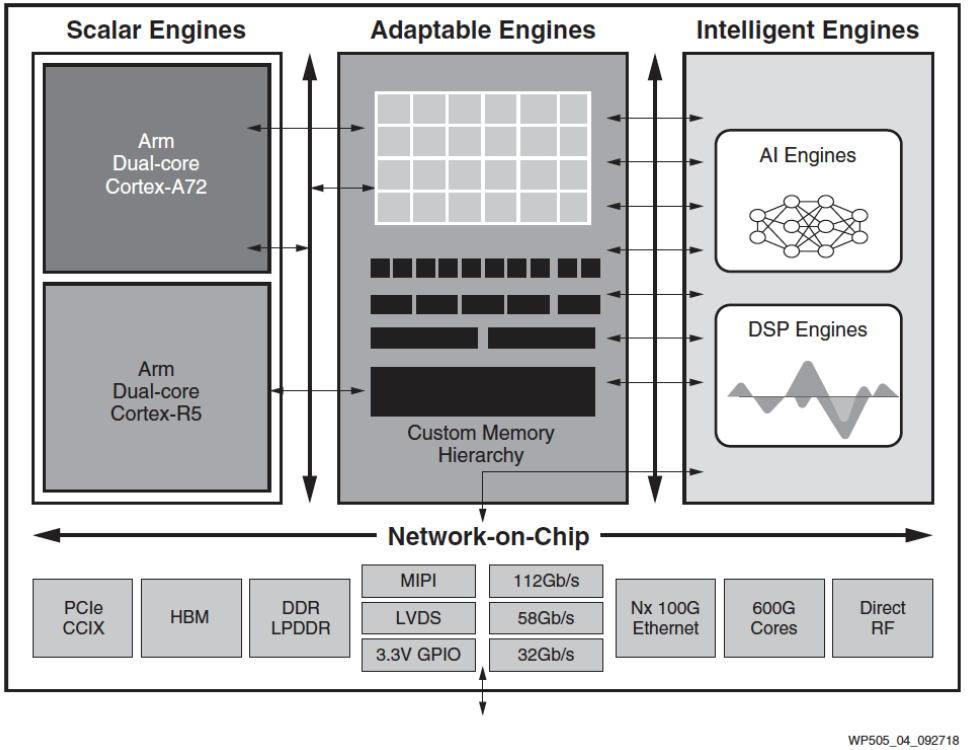
Los aceleradores actuales y los que se van utilizar a medio plaza para deep Learning están basados en tecnología CMOS

- Evolución de GPUs.
- Nuevas posibilidades con FPGAs
- Diseño de ASICs específicos.

Otros coprocesadores: FPGA

Hardware Reconfigurable
Capaz de integrar todas las necesidades de computación:

Adaptive
Compute
Acceleration
Platform (ACAP)

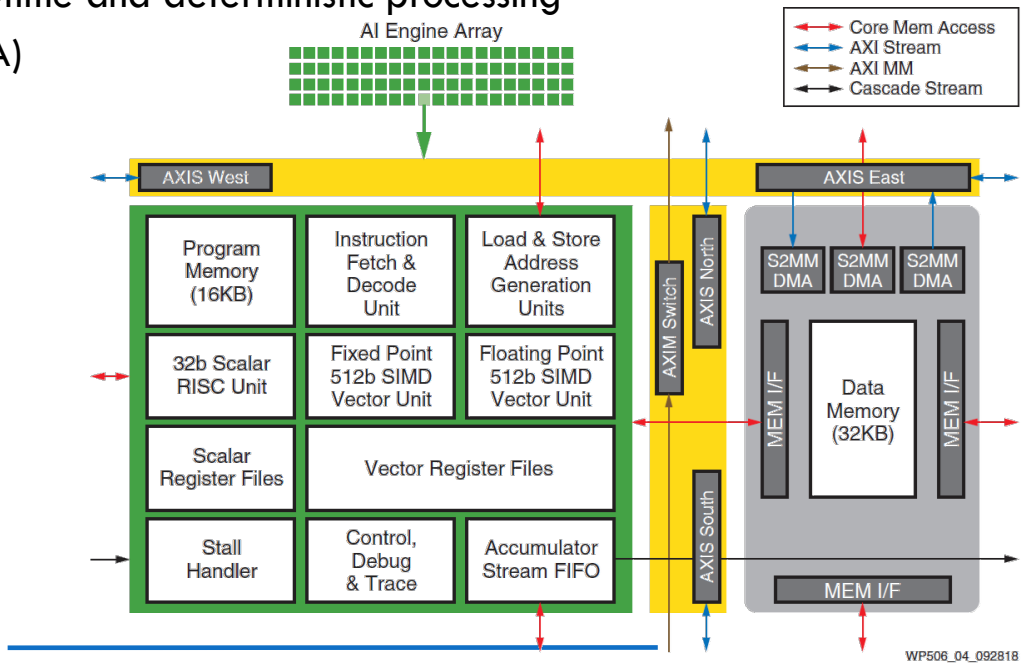


FPGA Xilinx versal: Processing System

- Dual-core ARM A72 with 2x single-threaded performance of previous generation A53's
- Dual-core ARM R5 for real-time and deterministic processing
- Peripherals:
  - 2 gigabit Ethernet controllers
  - 2 SPIs controllers
  - 2 I2Cs controllers
  - 2 CAN/CAN-FD controllers
  - 2 UARTs
  - GPIO
  - 1 USB 2.0 (device, host, and OTG) controller
- The DSP Engine
  - enables new modes of operation in addition to the conventional fixed-point operation

FPGA Xilinx versal: Processing System

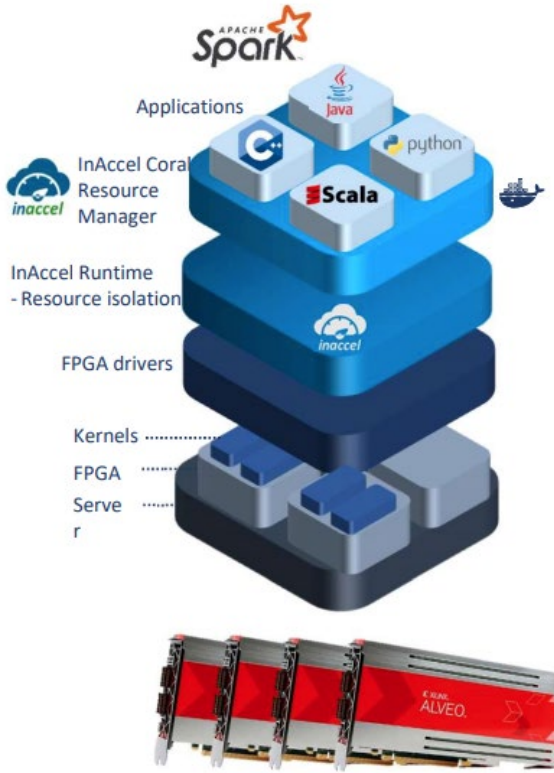
- Dual-core ARM A72 with 2x single-threaded performance of previous generation A53's
- Dual-core ARM R5 for real-time and deterministic processing
- Adaptable resources (FPGA)
- AI Engine



FPGA Xilinx versal: AI Engine Array

- AI Engine array is a two dimensional array of AI Engine tiles containing:
  - AI Engine,
  - High-performance VLIW vector (SIMD) processor
  - Integrated memory
  - Interconnects for streaming, configuration, and debug
- AI Engine tile contains a 32KB data memory divided into eight single-port banks, each 256-bit wide and 128b deep.
  - This structure allows up to eight parallel memory access transactions every clock cycle, with five cycle access latency

Coprocesadores FPGA



InAccel Coral:

- Program against your FPGAs like it's a single pool of accelerators
- "automated deployment, scaling, and management of FPGAs"

> **Up to 15x speedup for LR ML (7.5x overall)**

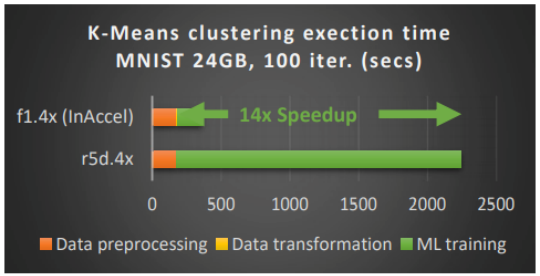
> **Up to 14x speedup for Kmeans ML (6.2x overall)**

> **Spark- GPU* (3.8x – 5.7x)**

> **F1.4x**
>> 16 cores + 2 FPGAs (InAccel)

> **R5d.4x**
>> 16 cores

*[Spark-GPU: An Accelerated In-Memory Data Processing Engine on Clusters]



Coprocesadores FPGA

The main differentiating factor is that **can be reconfigured** as opposed to the other chips:

- It allows for specifying hardware description language (HDL) that can be in turn configured in a way that matches the requirements of specific tasks or applications.
- It is known to consume less power and offer better performance.
- It can also offer a cost-effective option for prototype. It is much more flexible and is, therefore, a good choice for applications that involve customer-centric applications

Advantages

- It is highly flexible and is suited for rapidly growing and changing AI applications. For instance, with neural networks improving, it provides an architecture to undergo changes
- It shows better performance and consumption ratio
- Offers high accuracy
- FPGA shows efficiency in parallel processing. Overall it has significantly higher computer capability
- FPGAs offer lower latency than GPUs

Disadvantages

- Difficult to program and Development time is more
- Performance may not be up to the mark sometimes
- Not good for floating-point operation.

Universidad Autónoma de Madrid

excelencia UAM CSIC

Escuela Politécnica Superior

Programación GP-GPU

Paralelismo de Datos a Gran Escala

Francisco J. Gómez Arribas

Agenda

- **GPU Architecture**
- **CUDA Programing Model**
  - **Some Numbers: Fitting Blocks &Threads (G80 Arch)**
- **CUDA Memory architecture**
- **CUDA Warp execution**
  - **Latency Hiding**
  - **Threads divergence**

Review of GPU Architecture

- Streaming Multiprocessors (SM)
  - Compute Units (CU)
- Streaming Processors (SP) or CUDA cores
  - Vector lanes
- Number of SMs x SPs
  - Tesla (2007): 30 x 8
  - Fermi (2010): 16 x 32
  - Kepler (2012): 15 x 192
  - Maxwell (2014): 24 x 128
  - Pascal (2016): 56 x 64
  - Volta (2017): 80 x 64

	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
Marco temporal	2006-07	2008-09	2010-11	2012-13	2014-15	2016-17	2018-?
N (multiprocesadores)	16	30	14-16	13-15	4-24	56	80
M (cores/multiproc.)	8	8	32	192	128	64	64
# cores	128	240	448-512	2496-2880	512-3072	3584	5120

SM = Streaming Multiprocessor

SM is the workhorse in NVIDIA's GPUs

- Sometimes called SMX, SMM, etc. in marketing

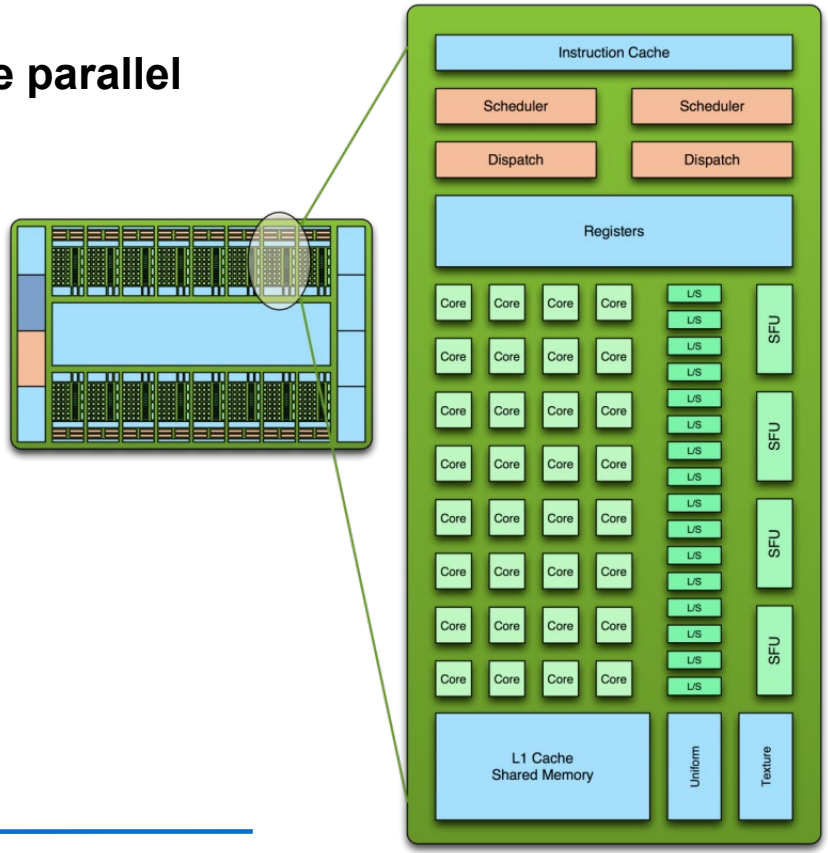
- Executes arbitrary code
  - Including shaderswhen doing graphics

- Capabilities and SM count depend on architecture and board
  - Tesla K2000 (Kepler): 2SMs (Maari-A)
  - GTX Titan X (Maxwell): 24 SMs
  - Tesla P100 (Pascal): 56 SMs –but half size

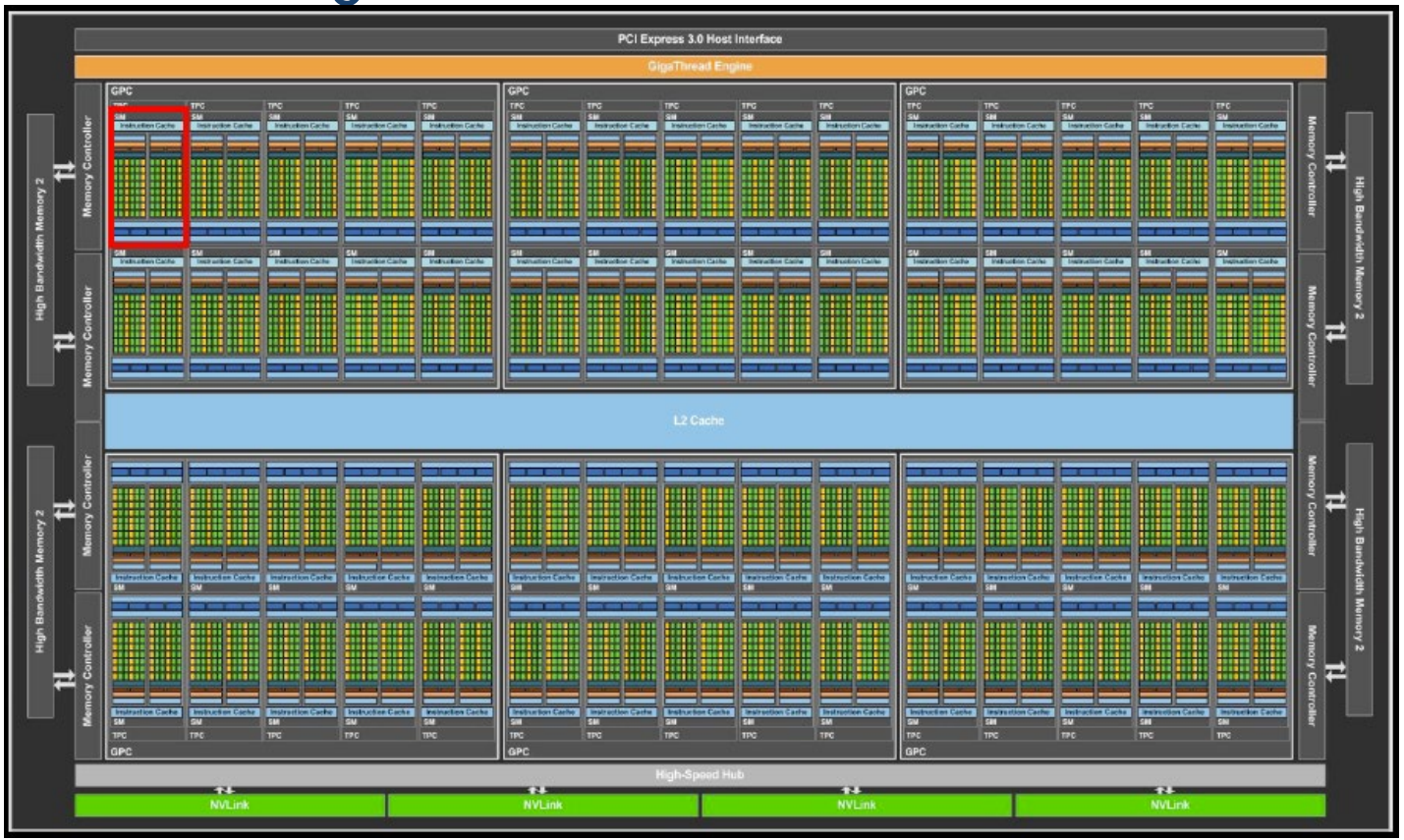
GPU : 20-Series Architecture

- 512 Scalar Processor (SP) cores execute parallel thread instructions

- 16 Streaming Multiprocessors (SMs) each contains
  - 32 scalar processors
    - 32 fp32 / int32 ops / clock,
    - 16 fp64 ops / clock
  - 4 Special Function Units (SFUs)
  - Shared register file (128KB)
  - 48 KB / 16 KB Shared memory
  - 16KB / 48 KB L1 data cache

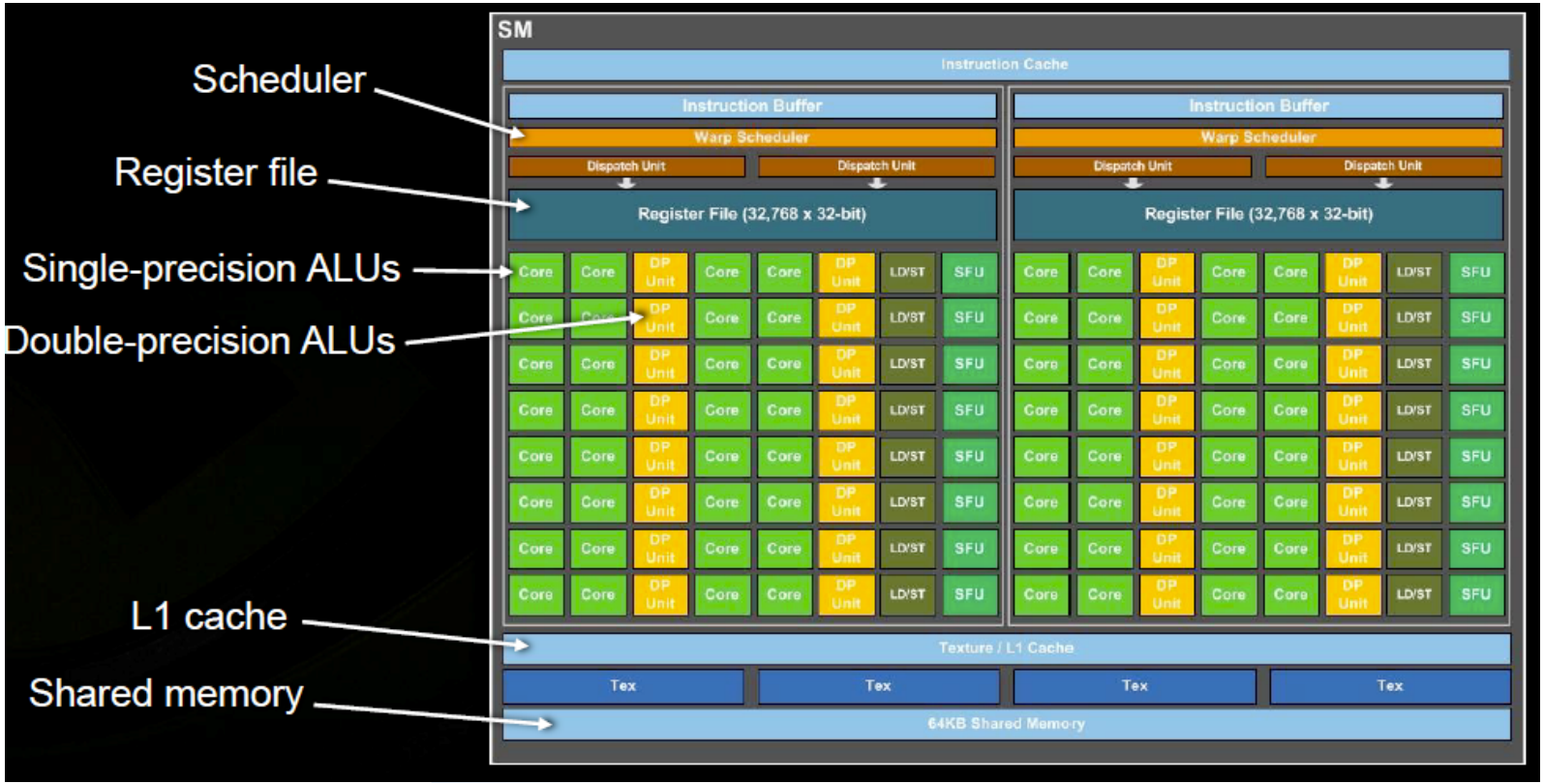


P100 Block diagram

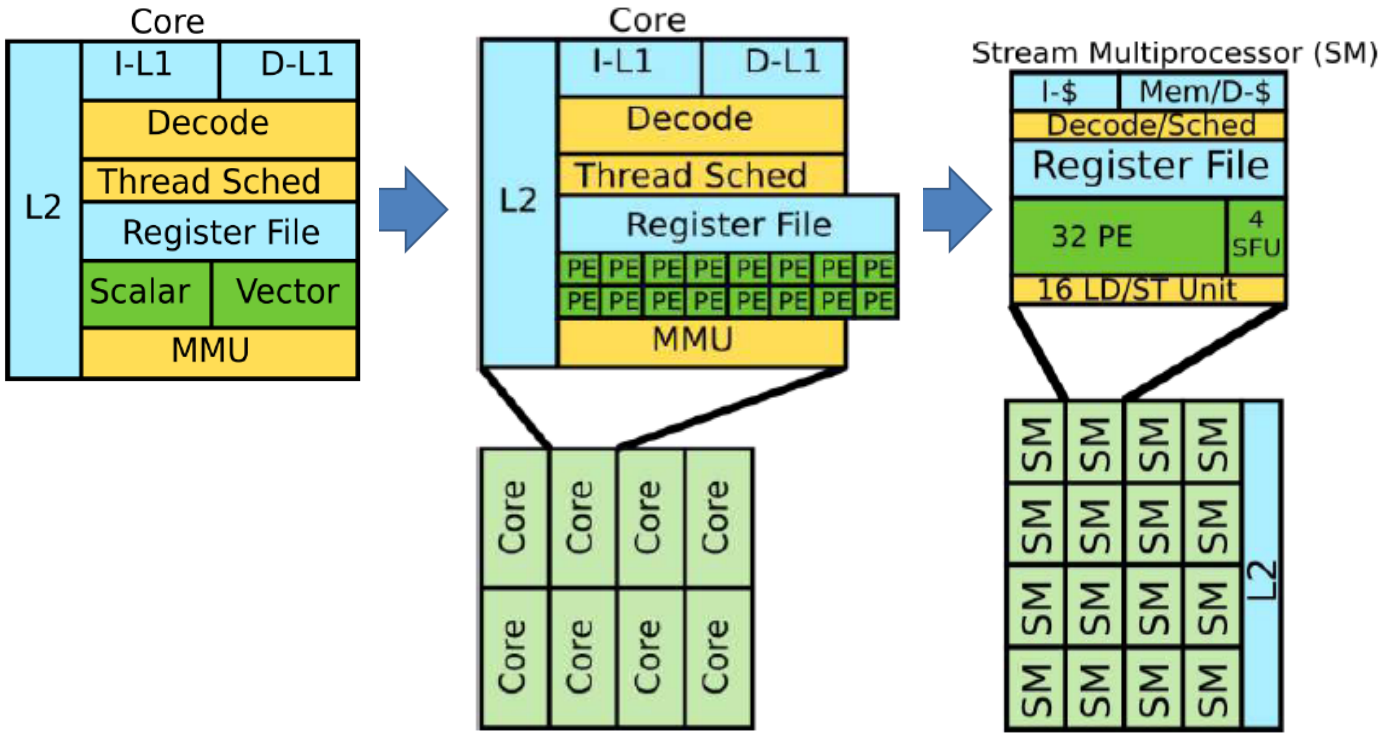


P100 Block diagram

Pascal SM

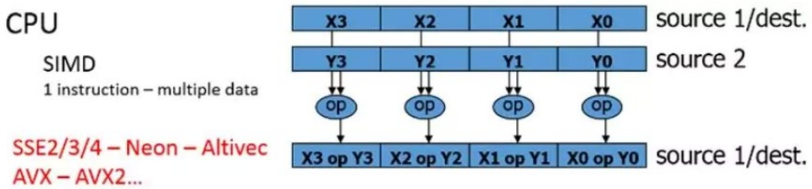

Scheduler
Register file
Single-precision ALUs
Double-precision ALUs
L1 cache
Shared memory
SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory

Multicore vs Stream multiprocesor

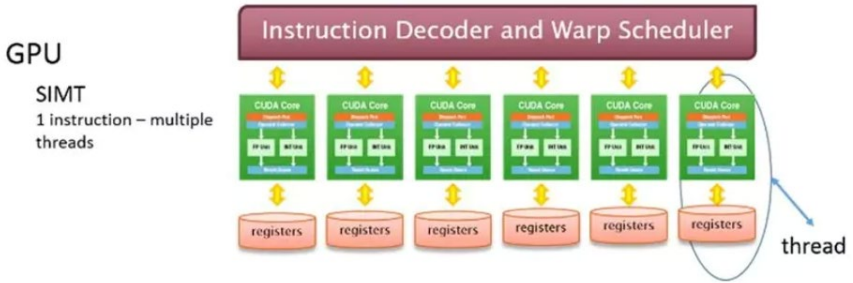


GPUs are SIMD Engines Underneath

- GPU: A massively parallel architecture
  - The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
  - However, the programming is done using threads, NOT SIMD instruction



- Before, let's distinguish between
  - Programming Model (Software) vs.
  - Execution Model (Hardware)



SIMT execution model

- Program Counter (PC)
  - **All threads (in a warp) have the same PC**
  - **I.e., they execute the same instruction on a given cycle**

- **How is this possible?**
  - Sounds like SIMD, but how can threads be independent?
- **SIMT= Single Instruction, Multiple Threads**
  - Close to SIMD, but allows free per-thread control flow
  - Built into SM instructions and scheduler
  - Dedicated hardware is necessary for efficient implementation

SIMT vs SIMD

- **SIMD (Single Instruction Multiple Data)**
  - Used in CPUs, e.g. Intel's SSE/AVX extensions
  - Programmer sees a scalar thread with access to a wide ALU
  - For example, able to do 4 or 8 additions with a single instruction

- **SIMT (Single Instruction Multiple Thread)**
  - Programmer sees independent scalar threads with scalar ALUs
  - Hardware internally converts independent control flow into convergent control flow

Execution Model

Software

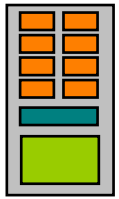
Thread

Thread Block

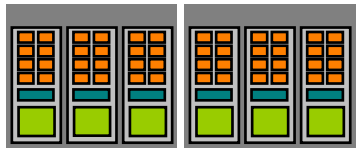
Grid

Hardware

Scalar Processor



Multiprocessor



Device

Threads are executed in Warps by scalar processors

Thread blocks are executed on multiprocessors

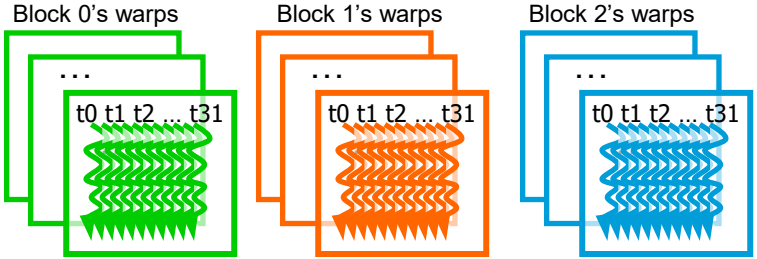
Thread blocks do not migrate

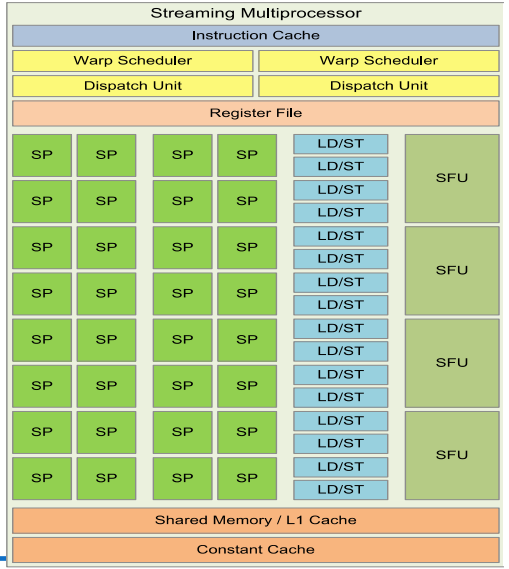
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

GPU Architecture: Thread scheduling

- Blocks are divided into warps
  - SIMD unit (32 threads)
  - Each Streaming Processor (SP) executes a thread
- Thread block occupies a number of warps in the same SM
  - E.g., block of 128 threads will be packed into 4warps
- Streaming Multiprocessor (SM) operates at warp granularity
  - Resource allocation
  - Execution





Undertanding Warps and Blocks

Threads in GPUs are organized in two ways:

- Warps (always 32 threads)
  - helps the hardware design:
    - lots of arithmetic power with a simpler control
    - you will have to live with this even if it is inconvenient for you
- Blocks (you can choose the block size (e.g. 64 or 256 threads)
  - helps the programmer:
    - threads of a block can easily and efficiently communicate with each other using “shared memory”
    - blocks are a useful feature that the programmer can use

Warps helps the Hardware

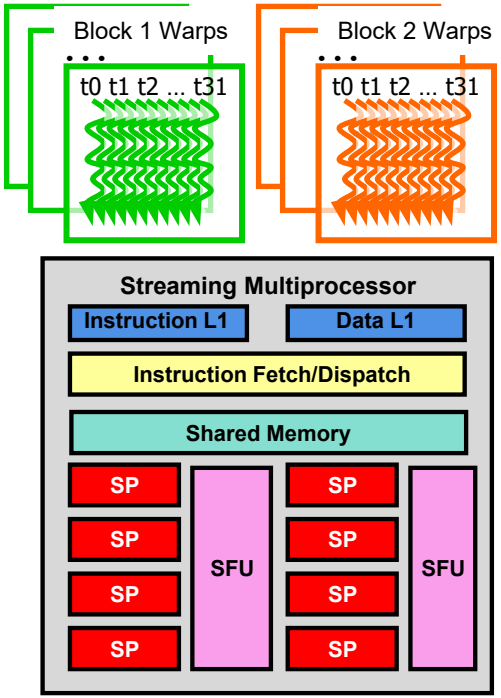
- What if there were no warps?
  - Our GPU has 640 arithmetic units for doing scalar operations
  - If we had only individual threads, you would need 640 schedulers that process instructions from individual threads and move operands to the right arithmetic units
  - By organizing threads in warps, we only need 20 schedulers that process instructions from complete warps and move warp-wide operands to warp-wide arithmetic units
  - Less space & energy used by control logic, more space & energy left for useful work

Blocks helps the Programmer

- Blocks are there to help you!
  - You can allocate a small amount of very fast "shared memory" for each block
    - "small amount" ≈ kilobytes per block
    - "very fast" ≈ L1 cache
  - All threads of a block see the same shared memory
  - Threads of a block can use shared memory to communicate with each other and coordinate their work

Thread Scheduling/Execution

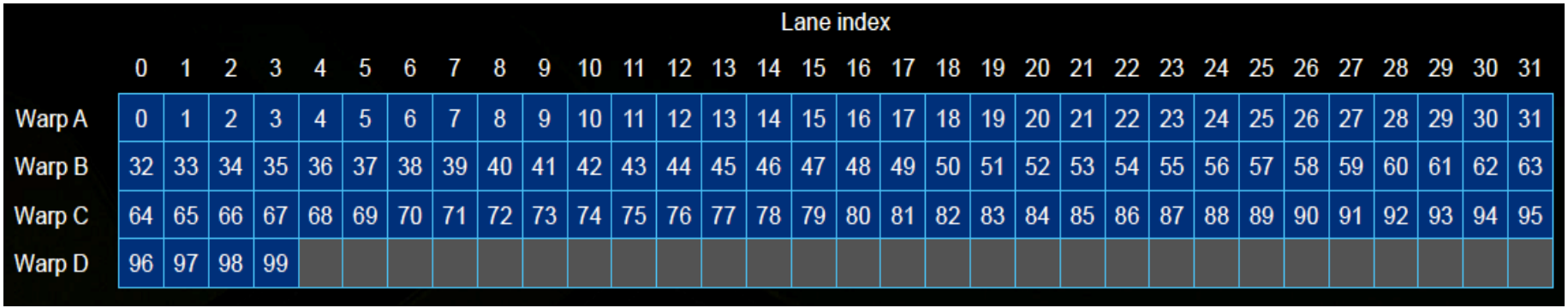
- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



Warps example

Block of 100 threads

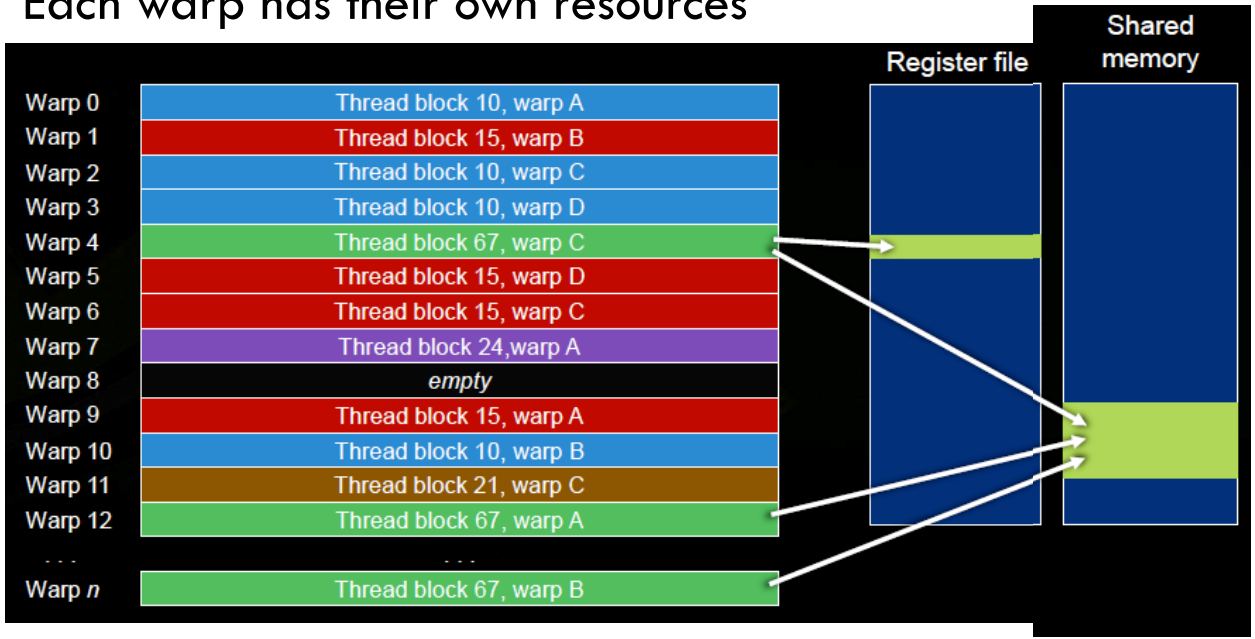
```
blockDim.x= 100
```



Note: If blockDim.x== 32, then threadIdx.x will give the lane index

SM view of warps

- SM has a pool of resident warps
  - Each warp has their own resources



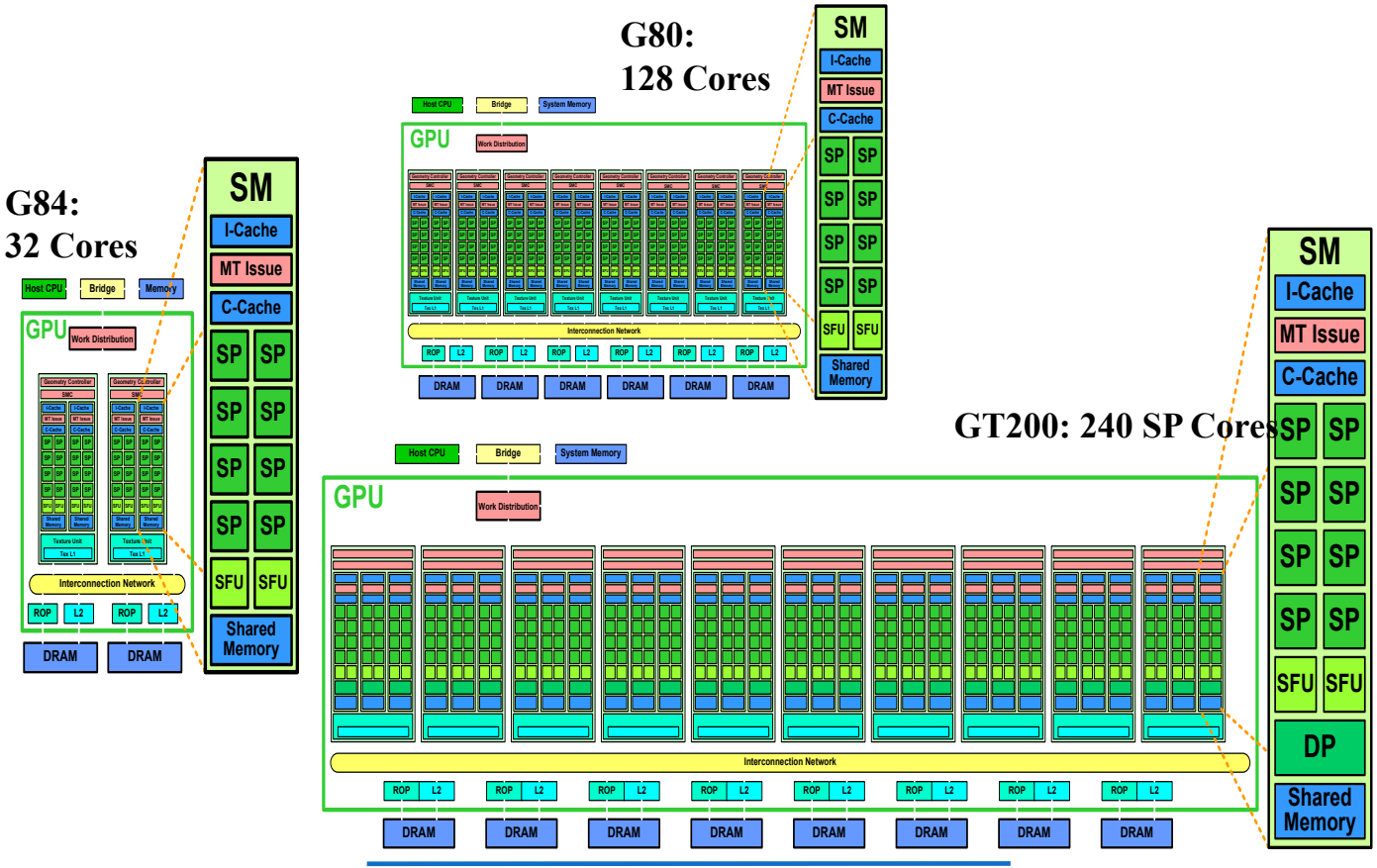
Occupancy and latency hiding

- **Having a large number of warps is how GPU hides latencies**
- **Anything that limits occupancy is bad for latency hiding**
  - Too high register usage
  - Too high shared memory usage
  - Too small thread blocks => cannot utilize all warps
- **Kernel's occupancy can be queried using CUDA API**
- **Or calculated using the "CUDA Occupancy Calculator"spreadsheet**

Agenda

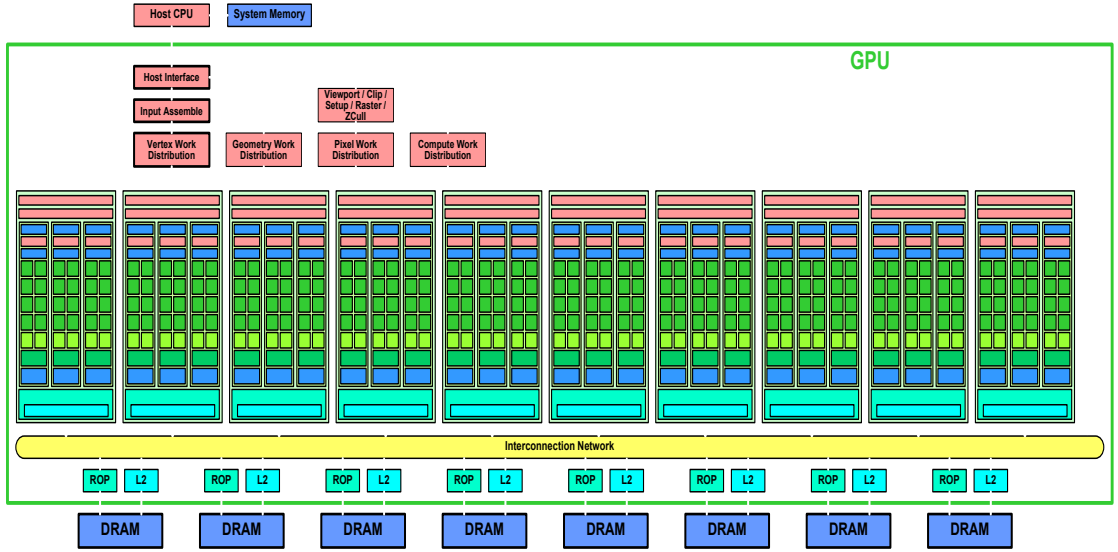
- **GPU Architecture**
  - **Some Numbers: Fitting Blocks &Threads (G80 Arch)**
- **CUDA Programing Model**
- **CUDA Memory architecture**
- **CUDA Warp execution**
  - **Latency Hiding**
  - **Threads divergence**

Reason for blocks: GPU scalability



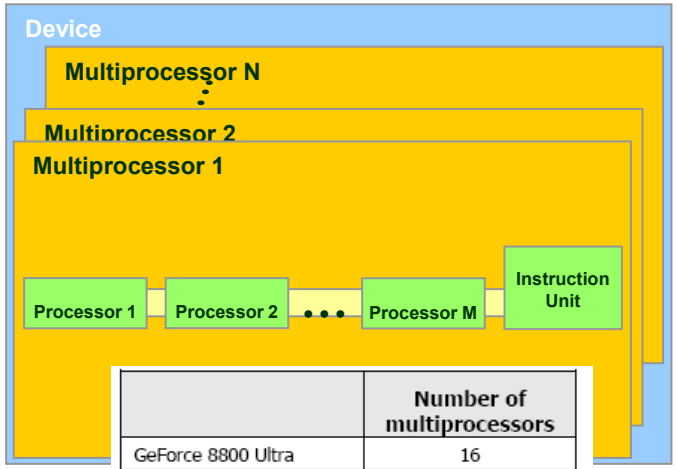
Manycore GPU

- GT200 chip (Tesla C1060 / one GPU of Tesla S1070)
- 240 Units execute kernel threads, grouped into 10 multiprocessors
- Up to 30,720 parallel threads active in the multiprocessors
- Threads are grouped in blocks, providing shared memory: scalability



Hardware Implementation: SIMD Multiprocessors

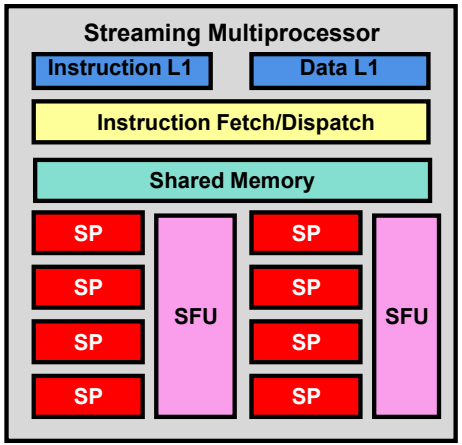
- The device is a set of 16 Streaming multiprocessors (SM) in G80 Arch.
- Each multiprocessor is a set of 32-bit streaming processors (SP) with a Single Instruction Multiple Data architecture – shared instruction unit
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
  - The number of threads in a warp is the warp size



	Number of multiprocessors
GeForce 8800 Ultra	16
GeForce 8800 GTX	16
GeForce 8800 GTS	12
GeForce 8600 GTS	4
GeForce 8600 GT	2
GeForce 8500 GT	2
Quadro FX 5600	16
Quadro FX 4600	12
Tesla C870	16
Tesla D870	2x16
Tesla S870	4x16

G80 Architecture

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and memory access

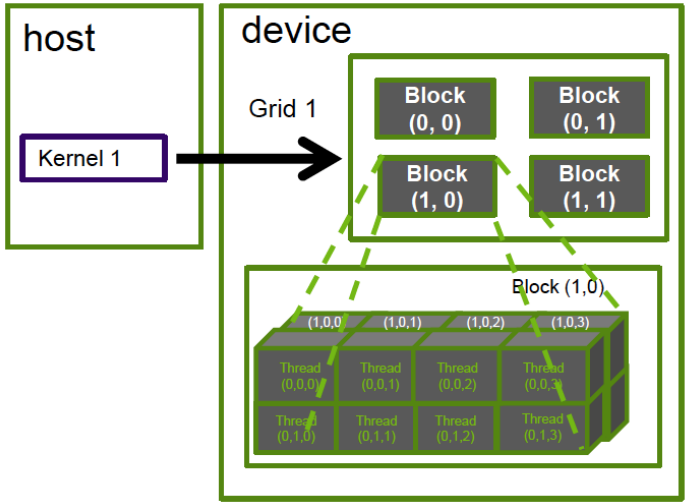


Threads, Warps, Blocks

- There are (up to) 32 threads in a Warp
  - Only <32 when there are fewer than 32 **total** threads
- There are (up to) **16 Warps in a Block**
- Each Block (and thus, each Warp) executes on a single SM
- G80 has 16 SMs
  - At least 16 Blocks required to "fill" the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

Technical Specification (G80)

Grid/Block/Thread limitation



- The maximum number of threads per block is 512;
- The maximum sizes of the x-, y-, and z-dimension of a thread blocks are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;

Technical Specification (G80)

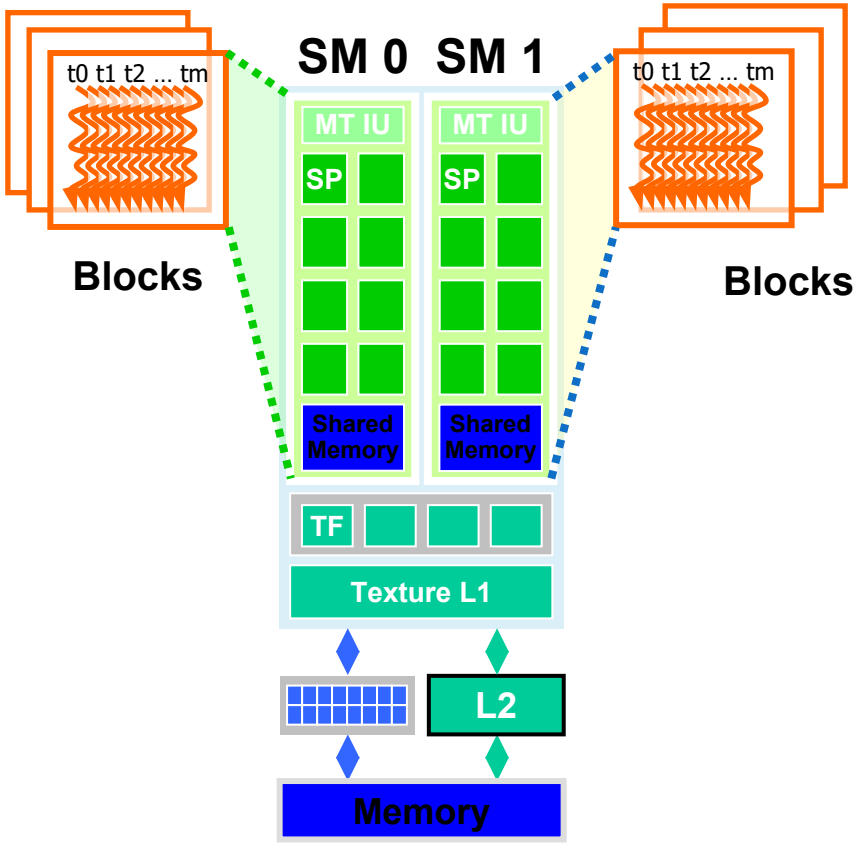
- The number of registers per multiprocessor is 8192;
- The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks
- The amount of constant memory available is 64 KB with a cache working set of 8 KB per multiprocessor;
- The cache working set for one-dimensional textures is 8 KB per multiprocessor;
- The maximum number of blocks that can run concurrently on a multiprocessor is 8;
- The maximum number of warps that can run concurrently on a multiprocessor is 24;
- The maximum number of threads that can run concurrently on a multiprocessor is 768;

Technical Specification (G80)

- The limit on kernel size is 2 million of native instructions;
- Each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles.
  - ***The scalar streaming processors have a pipeline of length 4***
- The maximum number of blocks that can run concurrently on a multiprocessor is 8;
  - ***The MP can be scheduled with 8 blocks. It can only run a single warp concurrently but it can pick a warp to execute from any of these 8 blocks.***

SM Executes Blocks

- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 384 (threads/block) * 2 blocks
    - Or 192 (threads/block) * 4 blocks
    - Or 128 (threads/block) * 6 blocks
    - Or 96 (threads/block) * 8 blocks
    - But not 64 (threads/block) * 12 blocks,



Granularity Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it can take up to 12 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 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM (in G80 architecture)

Agenda

- **GPU Architecture**
  - **Some Numbers: Fitting Blocks &Threads (G80 Arch)**
- **CUDA Programing Model**
- **CUDA Memory architecture**
- **CUDA Warp execution**
  - **Latency Hiding**
  - **Threads divergence**