GRAPHICS PROCESSING UNITS

Slides by: Pedro Tomás & Leonel Sousa

Additional reading: *Computer Architecture: A Quantitative Approach"*, 6th edition, Chapter 4, John L. Hennessy and David A. Patterson, Morgan Kaufmann, 2017

Outline

- Types of parallelism
  - Flynn's taxonomy
- GPU Architectures

Types of parallelism

- Instruction-Level Parallelism
  - Available when multiple instructions can be executed simultaneously
  - Exploitable by pipelining execution and through superscalar and VLIW architectures
- Data-Level Parallelism
  - Available by applying the same operation over different data or, equivalently, by dividing the data among multiple processing elements
  - Exploitable by:
    - Using SIMD instructions (vector processing) – fine-grained parallelism
    - Relying on multiple cores within a single processor – coarse-grained parallelism
    - Using multiple machines (cluster-level) – coarser-grained parallelism
- Task-Level Parallelism
  - Available by dividing the workload into multiple operations (functions), which are applied on the same or on different data
  - Exploited by using multiple processing elements (e.g., different cores or machines)

Graphics Processing Pipeline

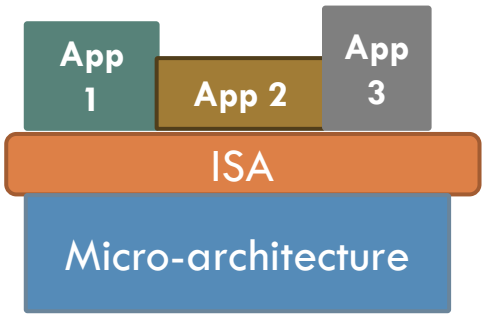
What are the computational requirements?

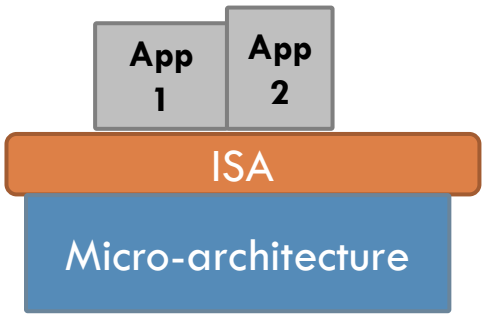
Which parallelism levels can be exploited?

How to design the architecture?

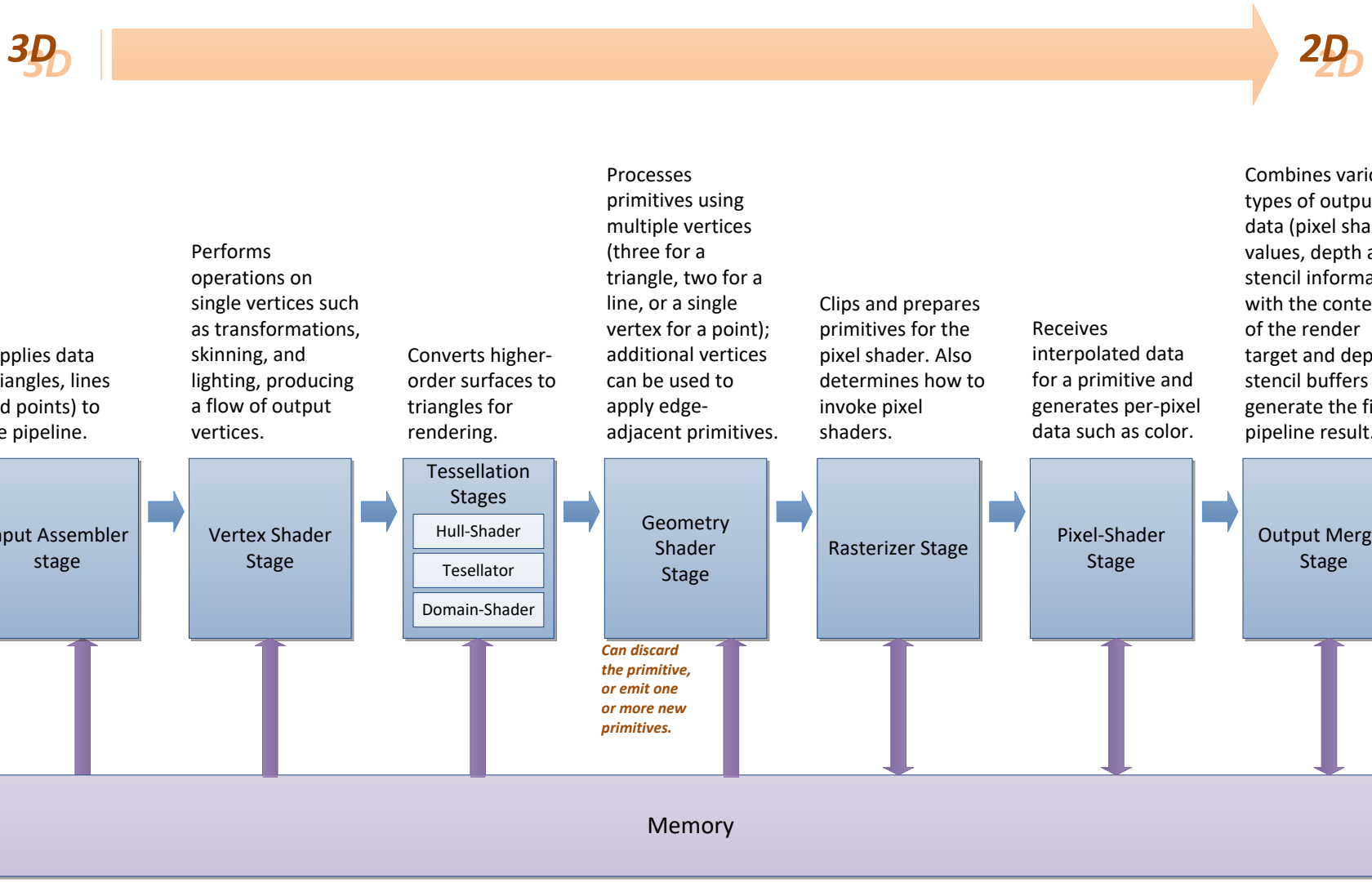
Micro-architecture design goals

	General Purpose Processors (GPPs)	Special Purpose Processors (SPPs)
Design Goals	Support for a wide range of applications → Requires a flexible ISA	Efficient support for a reduced set of applications → Can use a specialized ISA
Design Constraints	Speed (Latency/Throughtput) Cost Performance Power and Energy consumption	





DirectX11 Graphics Pipeline



Exploitable parallelism

- Millions of parallel computations to perform
  - In general the computation of a given pixel is independent from the adjacent pixels

Massive data-level parallelism

  - The modern graphics pipeline is complex, having to support different operations, over different display areas

Requires support for task-level parallelism

Exploitable parallelism

- Millions of parallel computations to perform
  - In general the computation of a given pixel is independent from the adjacent pixels

    ➔ **Massive data-level parallelism**

  - The modern graphics pipeline is complex, having to support different operations, over different display areas

    ➔ **Requires support for task-level parallelism**

  - Modern GPUs provide a uniform way of exploiting both parallelism levels by means of **thread-level parallelism**

TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- What should the be main characteristics?
  - Out-of-order execution?
  - Branch Prediction?
  - Pipeline Length?
  - Forwarding mechanisms?
  - Register file size?
  - SIMD instructions?
  - VLIW instructions?

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

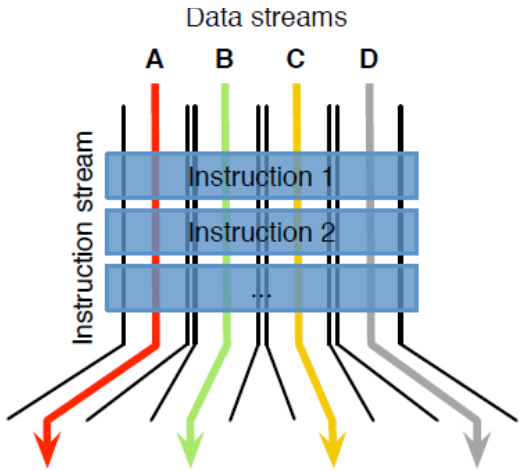
- What should the be main characteristics?
  - Out-of-order execution? **No!**
  - Branch Prediction? **No!**
  - Pipeline Length? **Long!**
  - Forwarding mechanisms? **No!**

  } (resolve hazards by interleaving execution with other threads)

  - Register file size? **Very Large!** (simultaneous support for multiple running threads)
  - SIMD instructions? **Yes!** (Each thread corresponds to a different vector element)
  - VLIW instructions? **Yes!** (Can be used by allow multiple threads per VLIW)

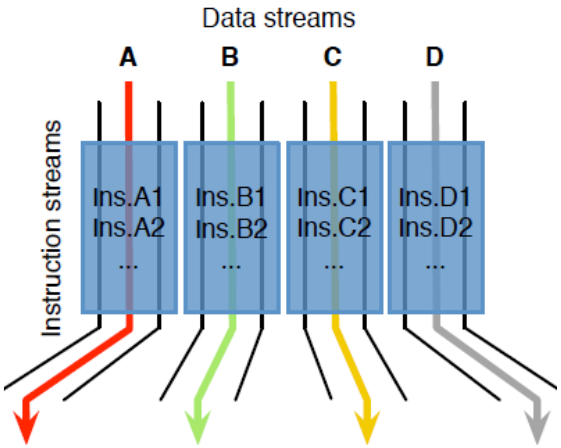
Micro-architecture single-core design

- SIMD-Like: At each cycle, each core issues the same instruction for a group of threads
  - A single instruction is performed on multiple data
    - Reduces pressure on the IF stages
  - Simpler control
  - If there is at least one execution unit per vector element, there are no structural (execution) hazards
  - If one thread stalls (e.g., memory accesses) all threads (in the same vector) stall
  - Low functional unit utilization
    - e.g., when using FP operations, the integer execution units are stalled



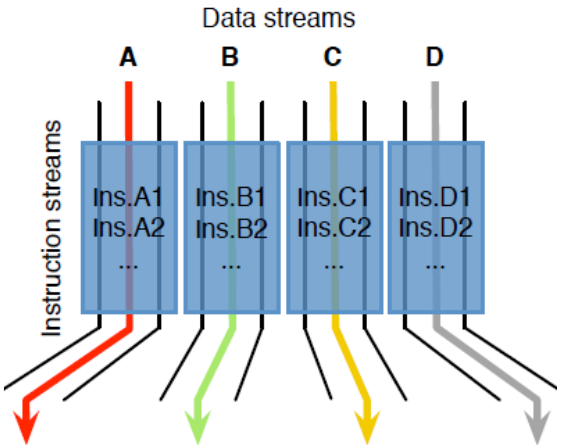
Micro-architecture single-core design

- VLIW-Like: At each cycle, the core issues a bundle of independent instructions from one or more threads
  - Stalls on one thread have limited impact on the execution of other threads
  - Reduced number of execution units
    - Reduces static power consumption
  - Increases hardware resource usage
  - Requires a high instruction memory (cache) throughput
  - Requires inter-thread parallelism to be extracted dynamically
    - Instruction scheduling is more complex
    - Leads to a higher power consumption



Micro-architecture single-core design

- MIMD-Like: At each cycle, the core issues multiple instructions from multiple vectors (each vector is composed of a group of n threads)
  - Allows increasing the functional unit utilization, regarding a pure SIMD-like approach
  - If one thread stalls, issue a different set of threads
  - Threads are statically bound to a given group to ease thread control
    - NVIDIA uses the terminology 'warp' to designate a group of 32 threads
    - AMD uses the naming 'wavefront' to represent a group of 64 threads
  - Requires the co-existence of multiple hardware *schedulers*



TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- How can we scale the performance?
  - Use simple cores
  - Allow the co-existence of multiple cores, each supporting a massive number of threads
    - Modern GPPs are usually composed of a relatively small number of cores (8-16), each supporting a small number of running threads (up to 2)
    - Modern GPUs (e.g., NVIDIA GeForce GTX 2080Ti) are composed of more cores (60 SMs), each supporting the parallel execution of hundreds of threads (2048), typically executed in groups (1 warp = 32 threads)

TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- Instruction Set Architecture?
  - Modern GPPs use standard scalar ISAs, which are further extended to support a set of vector instructions
  - Modern GPUs use special purpose ISA, which are oriented for parallel thread (warp) execution
    - Generally supports most general purpose programming operations, although usually relying on predicated instructions to overcome control operations (branches) that lead to warp-level divergence

TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- Memory hierarchy?
  - Modern GPPs use multiple cache memories (L1, L2, L3) to minimize data access latency and avoid read-after-load hazards
  - Modern GPUs typically use only 2 cache levels (L1 and L2), together with a main (global) memory
    - Typical GPUs do not have access to the CPU global memory, hence requiring explicit data transfers between CPU and GPU
    - Each GPU core usually provides a low latency (reduced size) scratchpad memory, as well as a constant (texture) memory
    - The GPU L2 cache memory is shared among all cores
    - The GPU L1 cache is special: it can be used for register caching and optionally for data caching

TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- Syncronization?
  - Modern GPPs provide mechanisms for explicit synchronization and cache coherence among cores
  - Modern GPUs typically do not provide for such synchronization mechanisms:
    - Warps/Wavefronts are grouped in blocks and the execution of all threads within a block is assigned to a given core. Synchronization within a block of threads is possible, but not among different blocks (since blocks may be scheduled to different cores).

TÉCNICO LISBOA

Micro-architecture single-core design

- Design goals:
  - Exploit highly parallel and data intensive computations
  - Provide support for a wide number of threads
  - Requires high throughput for high definition displays

- Problems?
  - GPU micro-architectures are specially efficient for massively parallel applications… but… they struggle when
    - Application requires periodic synchronization with the host processor
    - The overheads related with memory transfers between the host and the GPU cannot be efficiently hidden behind kernel execution
    - Kernels have limited parallelism (remember that GPUs use parallelism to hide data hazards)
    - Kernels have too much control (there is no branch prediction)
    - Kernels are too small and have a very fast execution time (overhead of launching the kernel limits processing performance)

NVIDIA GPU micro-architectures

NVIDIA G80 micro-architecture

- In November 2006, the G80 micro-architecture introduced:
  - Support for C programming
  - Unification of the several processing pipelines, in order to allow support for vertex, geometry, pixel and computing programs.
  - Introduction of the single-instruction multiple-thread (SIMT) execution model
  - Shared memory and barrier synchronization mechanisms for inter-thread communication

- In June 2008, NVIDIA made a revision of the G80 architecture (codenamed GT200)
  - The number of stream processing cores increased from 128 to 240
  - Increased the size of the physical register file
  - Introduced hardware memory access coalescing
  - Introduced double precision floating point format

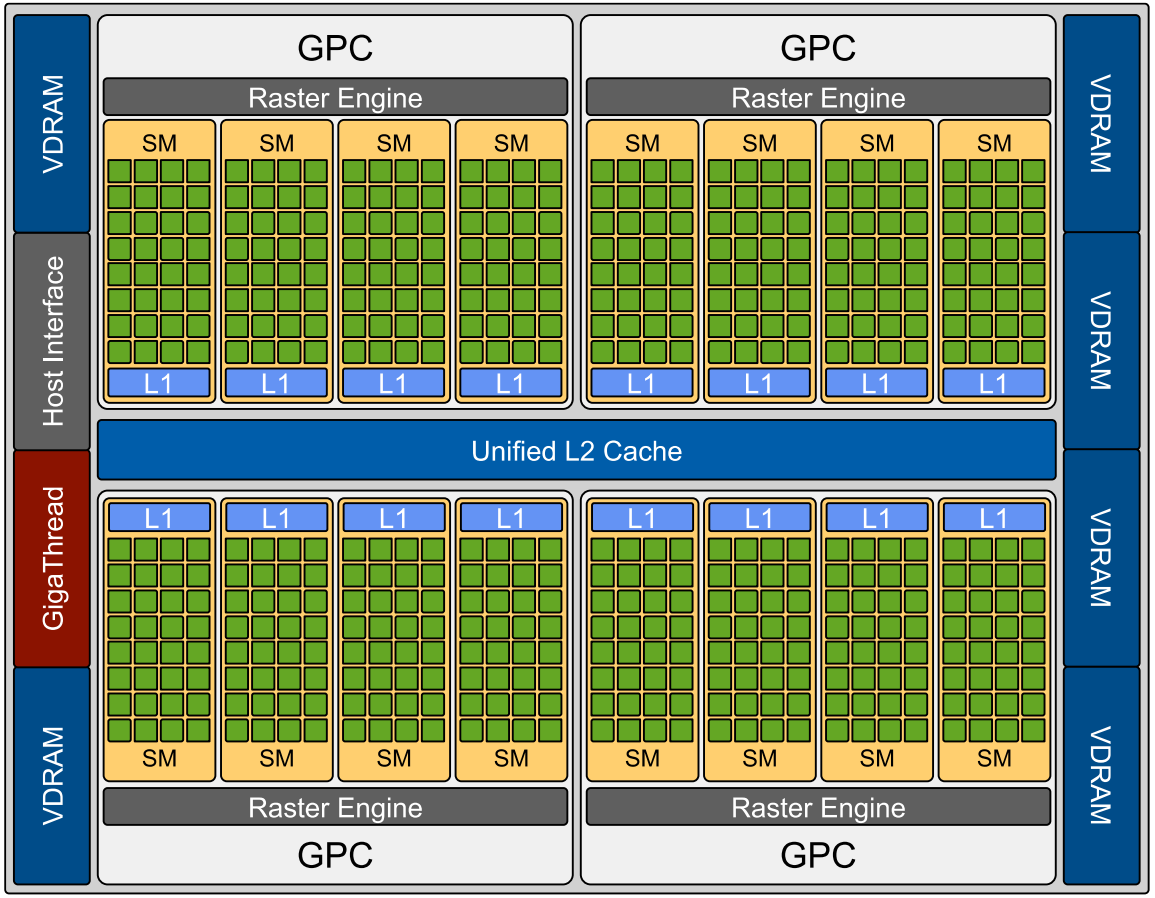
TÉCNICO LISBOA

Design goals (vs GT200)

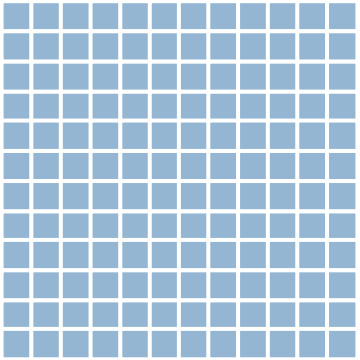
- Increase performance (especially double precision)
  - New Stream Multiprocessor (SM) architecture
  - Dual Warp Scheduler to allow simultaneously fetching and dispatching instructions from two independent warps
- Provide ECC support, to protect data against memory errors
- Introduce a true cache hierarchy
- Increase shared memory
- Faster context switching between programs
- Faster atomic operations

Architecture overview

- Each core is named a SM (Simultaneous Multiprocessor)
- A high-level GigaThread engine schedules the execution of the grid of blocks of threads to SMs

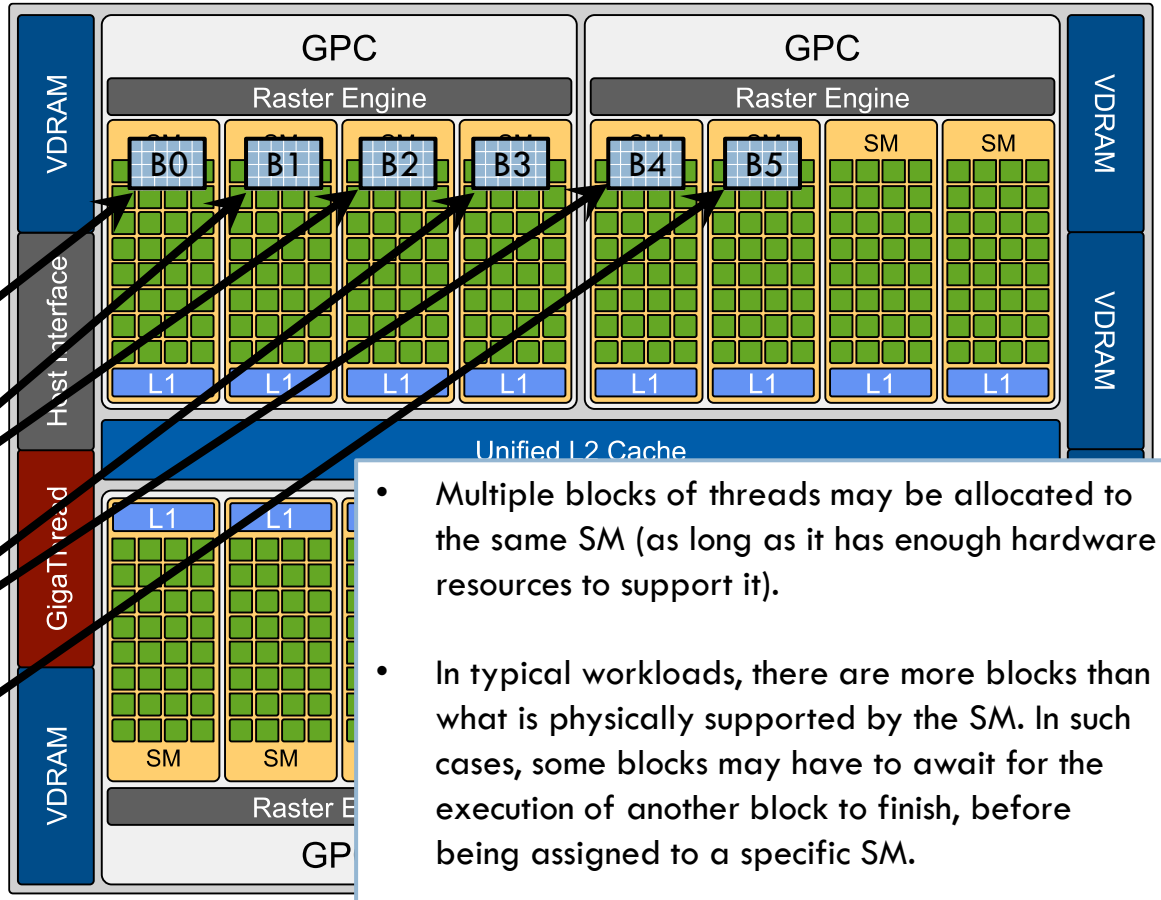


Workload (total number of threads)

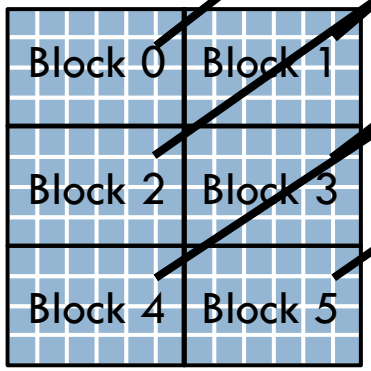


Block scheduling

- Each core is named a SM (Simultaneous Multiprocessor)
- A high-level GigaThread engine schedules the execution of the grid of blocks of threads to SMs



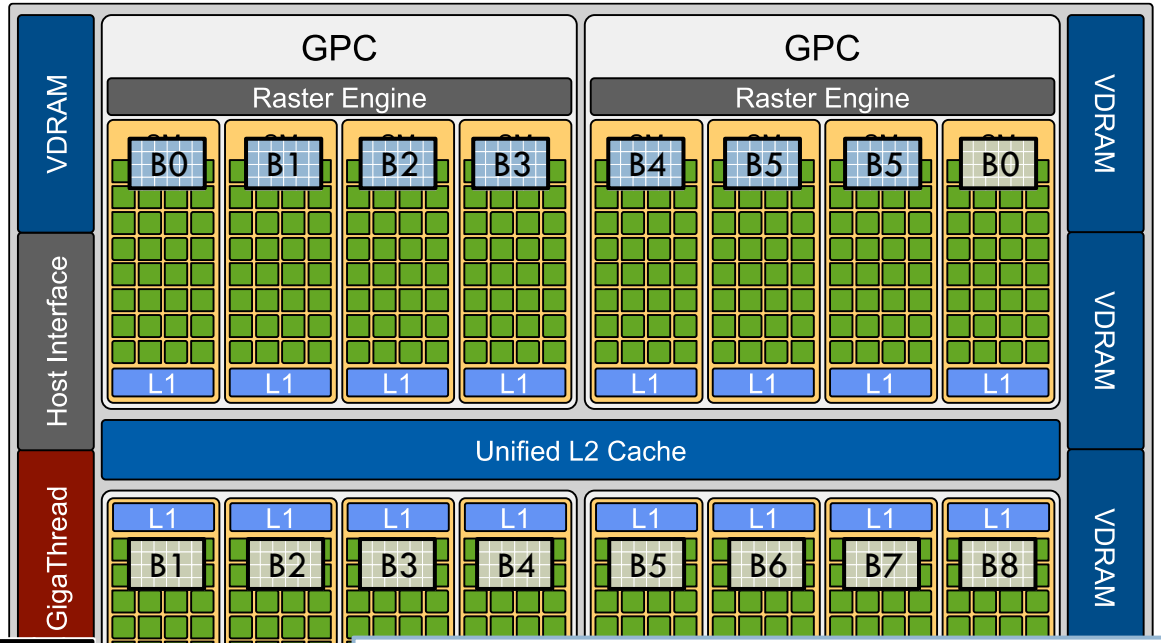
Workload (total number of threads)



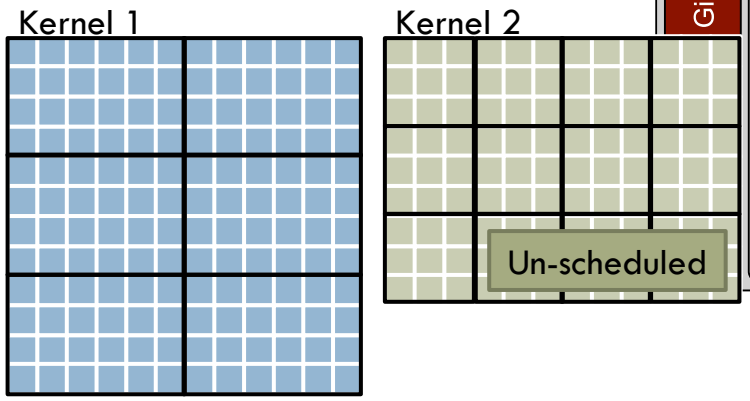
- Multiple blocks of threads may be allocated to the same SM (as long as it has enough hardware resources to support it).
- In typical workloads, there are more blocks than what is physically supported by the SM. In such cases, some blocks may have to await for the execution of another block to finish, before being assigned to a specific SM.
- Once a block is assigned to a given SM, it remains at that SM until completion.

Block scheduling

- Each core is named a SM (Simultaneous Multiprocessor)
- A high-level GigaThread engine schedules the execution of the grid of blocks of threads to SMs



Workload (total number of threads)

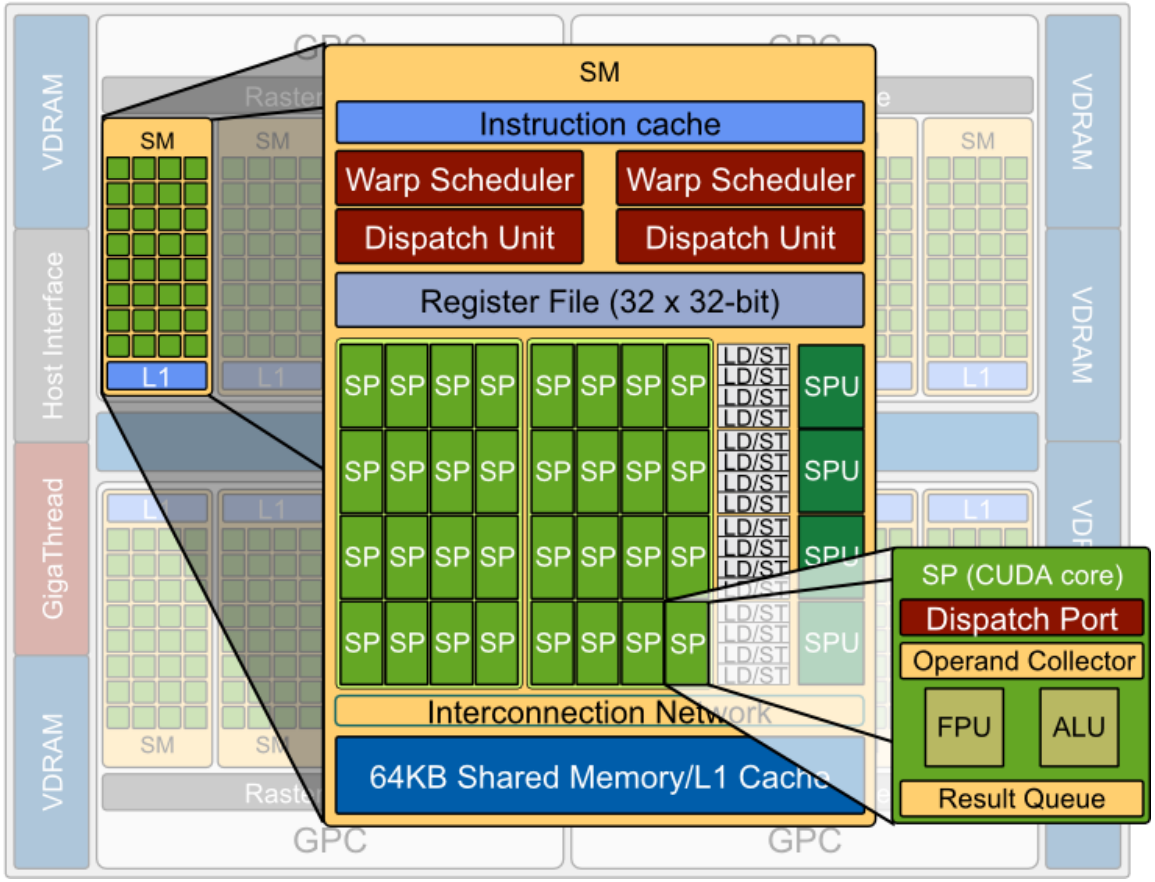


- Multiple kernels (up to 16) may execute concurrently has long as they are from the same context
- However, blocks are dispatched in-order. Thus, all blocks from one kernel have to be dispatched (i.e., assigned to a given SM), before starting dispatching blocks from the next kernel.

Simultaneous Multiprocessor (SM) Architecture

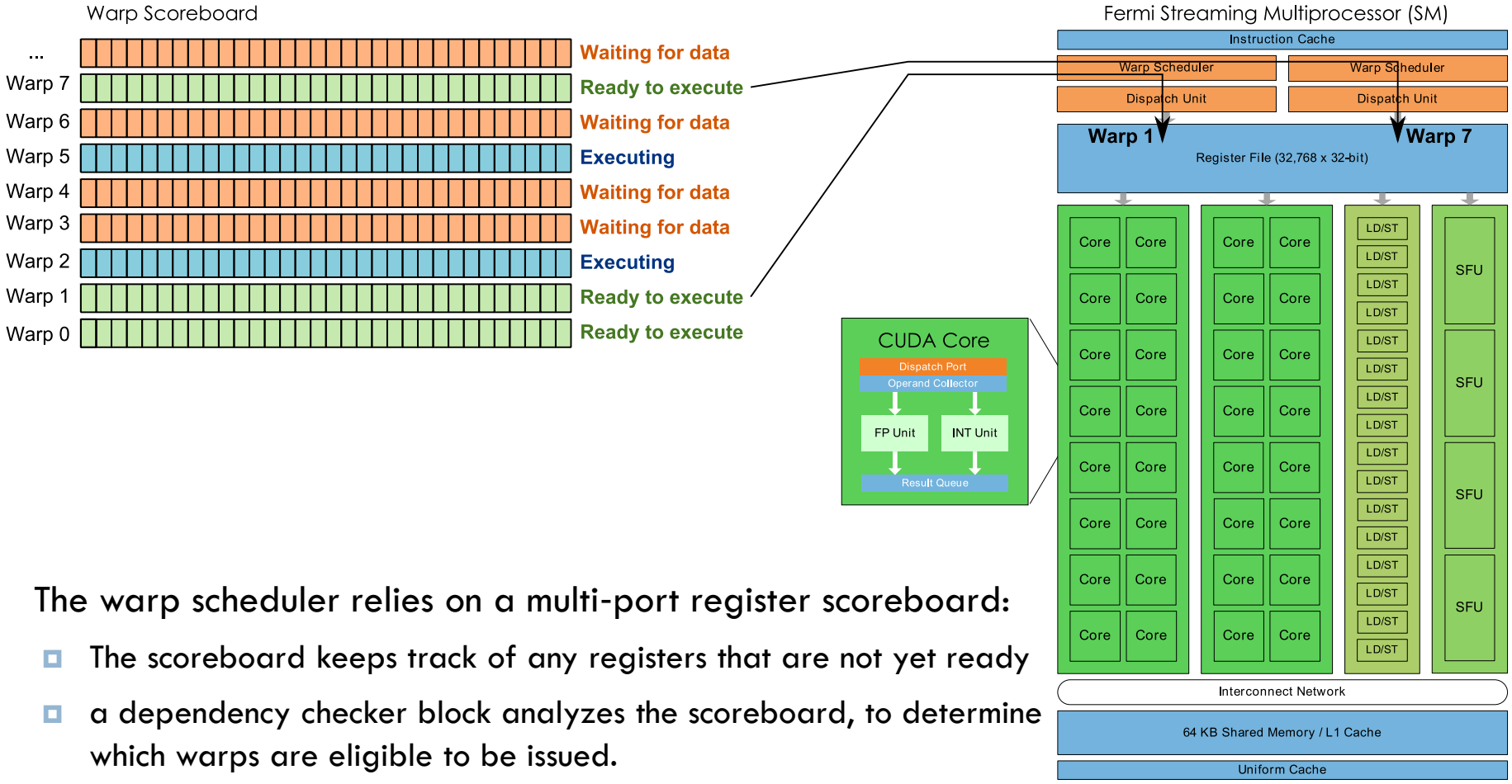
Each SM is composed of:

- Register file with 32K 32-bit registers
- 32 SPs, each composed of pipelined FP and integer ALU units
  - Supports up to 16 double precision fused-multiply and add (FMA) operations per clock
- 16 Load/Store units
  - A warp of 32 threads takes 2 cycles to execute
- 4 specialized processing units (SPUs)
  - Supports transcendental instructions such as sin, cosine, reciprocal, and square root
  - Executes one thread per clock, i.e., a warp of 32 threads takes 8 cycles to execute)
- 2 warp schedulers capable of issuing up to one instruction per clock cycle from a single warp of threads
  - Because threads are assumed to be independent no inter-warp dependency check is performed



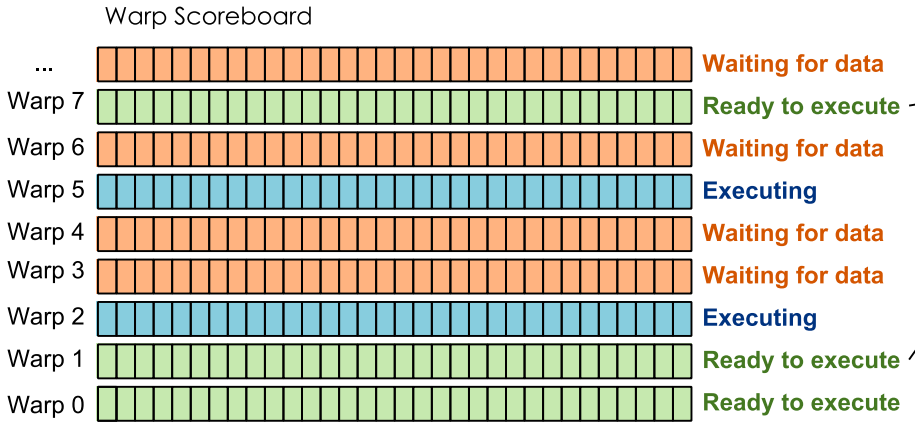
May lead to conflicts if two threads access the same memory position

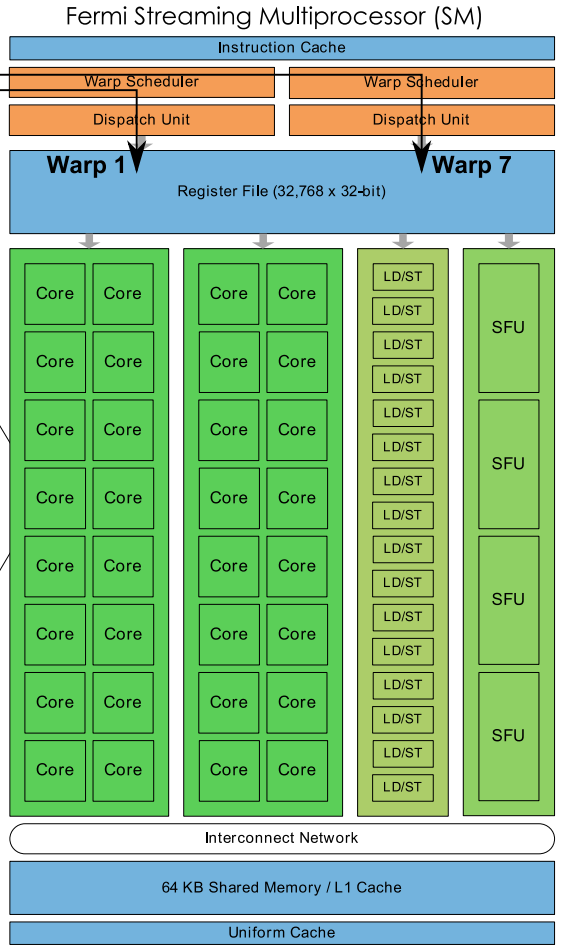
Warp scheduling

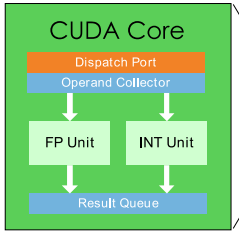


- The warp scheduler relies on a multi-port register scoreboard:
  - The scoreboard keeps track of any registers that are not yet ready
  - a dependency checker block analyzes the scoreboard, to determine which warps are eligible to be issued.

Warp scheduling



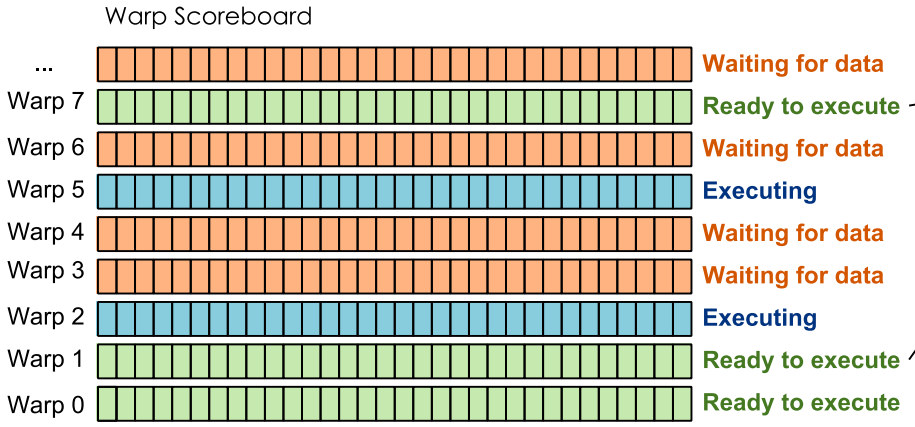


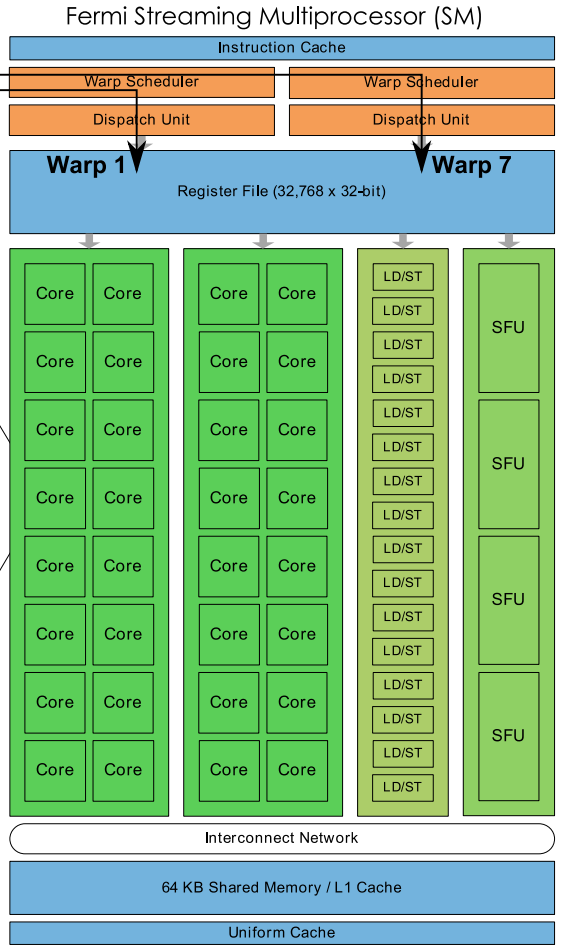


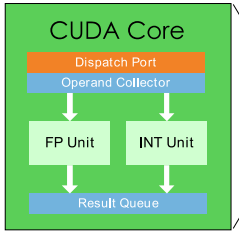
Each cycle:

- Each warp scheduler selects one warp to execute
  - If there is 1 (or less) warps ready to execute, one (or both) of the warp schedulers stall; this stalls may be overcome by increasing the number of threads in the same SM.
  - If there are 3 (or more) warps ready to execute, the warp stalls; the warp may also be stalled because the next assembly instruction has not yet been fetched, or because of a intra-block synchronization call (CUDA `_syncthreads()` function)
  - Whenever a double-precision operation is issued, the second warp scheduler must always stall. This is likely due to bandwidth limitations when fetching data from the register file.
- Structural hazards may also occur because the required HW resource is busy, e.g.:
  - The 4 SFUs are already operating over another warp
  - The warp requires loading data from memory, but too many outstanding requests have already been issued

Instruction Execution



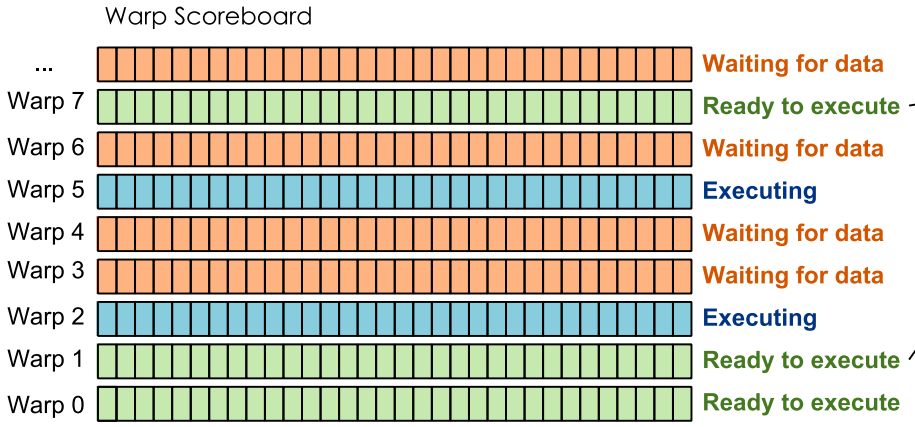


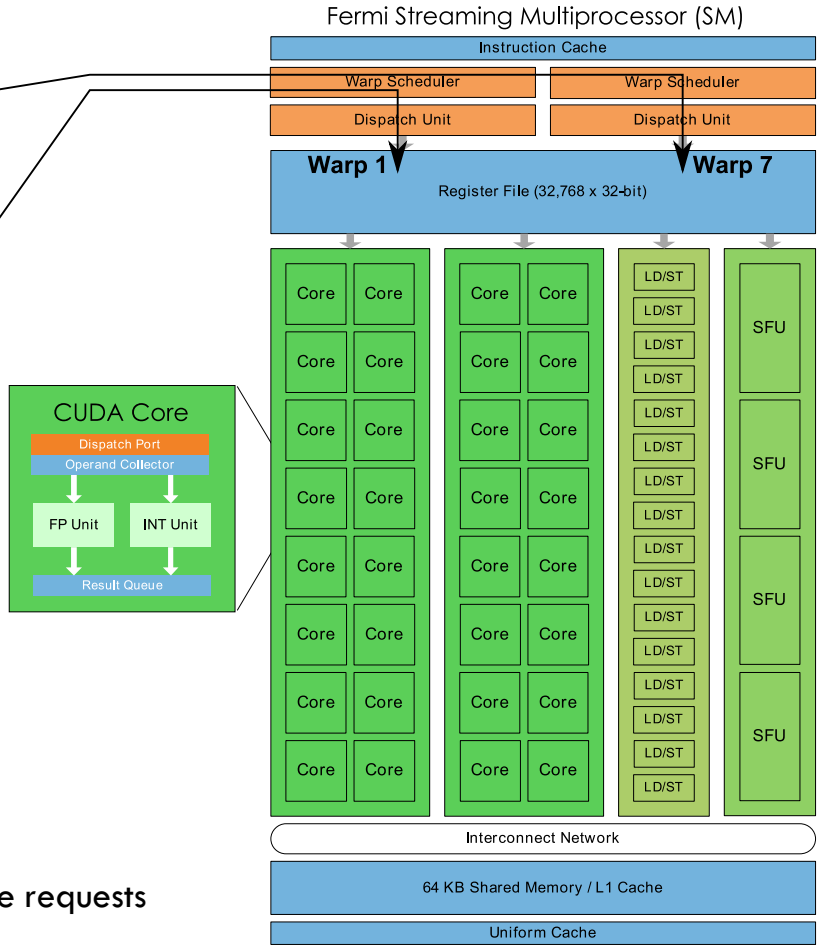


- **Each warp (group of 32 threads) is executed as**
  - 16 integer or FP units ➔ 2 cycles per warp
  - 16 LD/ST units ➔ 2 cycles per warp
  - 4 SFUs ➔ 8 cycles per warp

- **However, in Fermi the CUDA cores work at twice the operating frequency**
  - This allows saving processing resources (area), but leads to a higher energy consumption

Memory Hierarchy



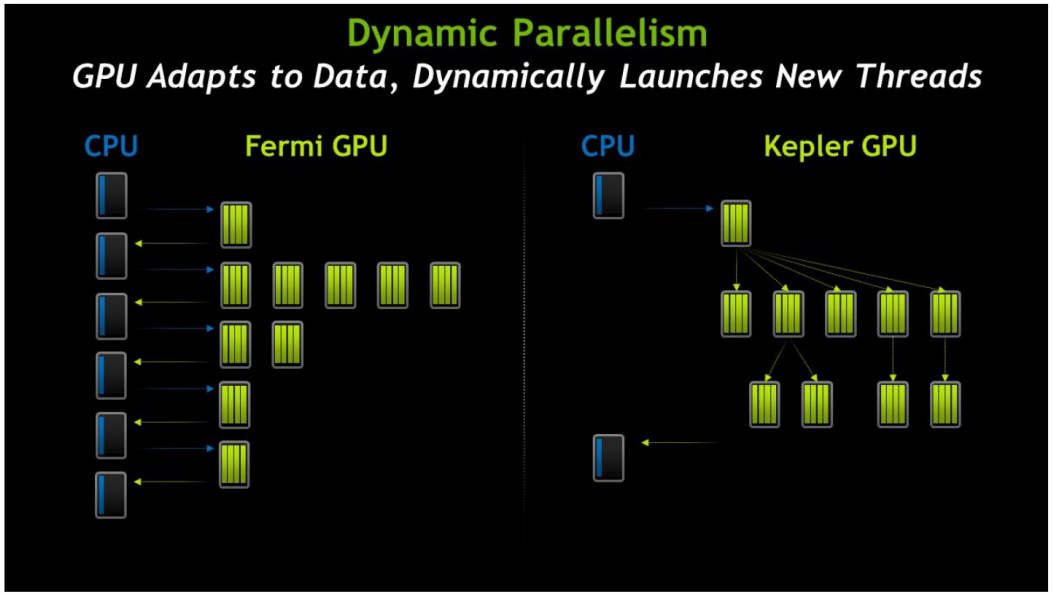


The SM memory hierarchy is composed of:

- An L1 texture cache memory
- A 64 KB L1 memory, which can be configured as:
  - 48KB for shared memory and 16 KB for L1 cache, or
  - 16KB for shared memory and 48 KB for L1 cache
- A 768KB of L2 cache, that serves all load, store and texture requests
- A global RAM memory used to store all data

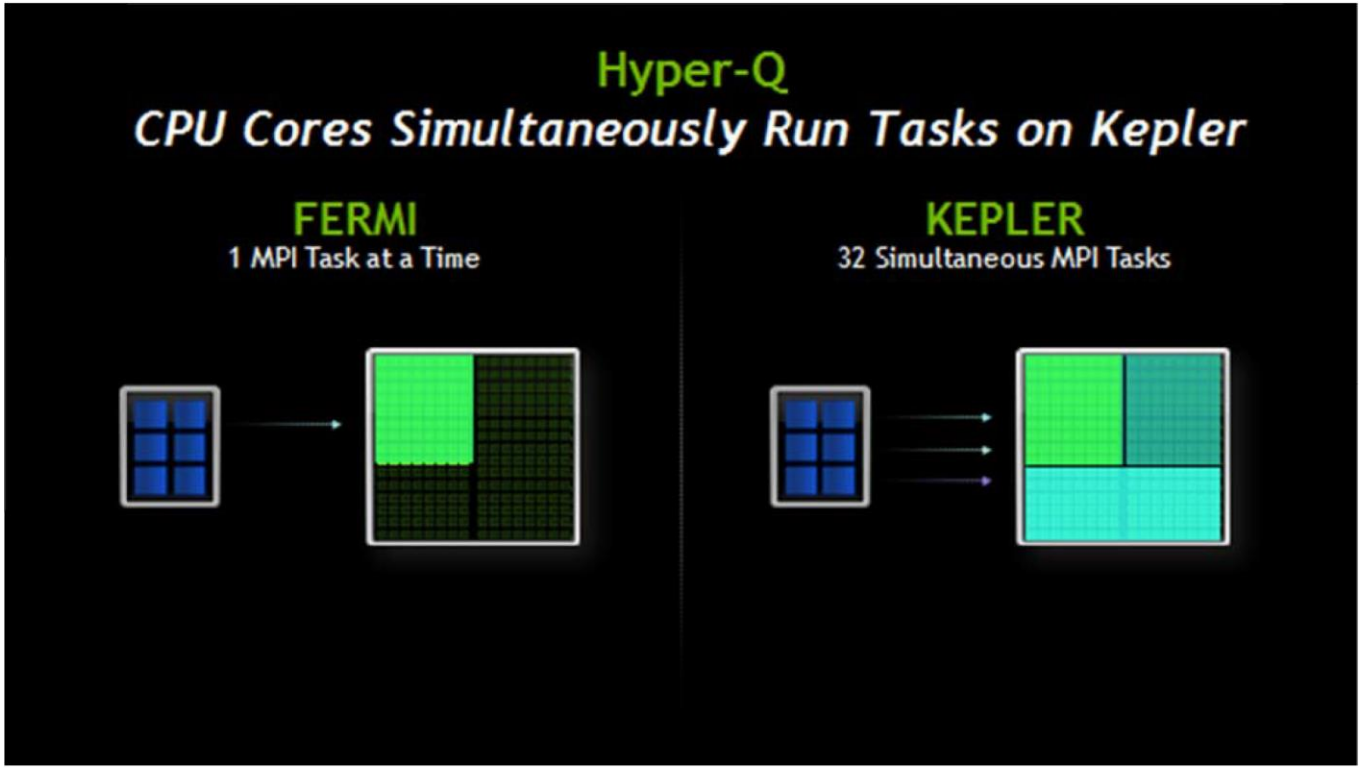
Design goals

- Introduces dynamic parallelism
  - Kernels (calls to GPU functions) can now generate new work, generating a new grid of blocks, which are subsequently sent for execution
  - Not only it allows to mitigate kernel call overheads, but also frees the CPU, allowing it to perform other tasks (task-level parallelism)
  - This also requires the Grid Management Unit to allow suspending the execution of ongoing work



Design goals

- Introduces dynamic parallelism
- Provide support for multiple work-queues, allowing different CPU cores to issue commands to the GPU on different CUDA streams.



Design goals

- Introduces dynamic parallelism
  - Kernels (calls to GPU functions) can now generate new work, generating a new grid of blocks, which are subsequently sent for execution
  - Not only it allows to mitigate kernel call overheads, but also frees the CPU, allowing it to perform other tasks (task-level parallelism)
  - This also requires the Grid Management Unit to allow suspending the execution of ongoing work
- Provide support for multiple work-queues, allowing different CPU cores to issue commands to the GPU on different CUDA streams.
- Allow for data transfers between different GPU devices without needing to go to the CPU/system memory.
- Improve the performance per Watt
  - Design of a new stream multiprocessor architecture (now named SMX)

Redesigned Simultaneous Multiprocessor (SMX)

- Large increase in the number of ALUs
  - 192 single-precision (or integer) CUDA cores (from 32 in Fermi)
  - 64 double-precision FP units (from 16 in Fermi)
  - 32 SFUs (from 4 in Fermi)
  - 32 LD/ST units (from 16 in Fermi)
- Each dispatch port is composed of 16 functional units
  - Warps are executed in 2 groups of 16 threads (half-warp)
  - Execution units now operate at the standard SM frequency
- The increase in the number of ALUs results in a linear increase in power consumption; however, the reduction in half of the core operating frequency reduces power consumption by 8x
  - Power increases linearly with area, but decreases cubically with operating frequency.

Redesigned Simultaneous Multiprocessor (SMX)

- Four warp schedulers and 8 dispatch units
  - Each warp scheduler now selects one warp per cycle, and issues up to 2 instructions (in-order) from that warp
  - Double precision instructions can now be paired with other instructions; however, double-precision instructions still require twice the RF bandwidth
    - A double-precision instruction probably still uses 2 dispatchers
  - Warp scheduling is aided by compiler information, which provides the warp scheduler with the latency for a result to be generated

Improved ISA

- New ISA
  - Each thread can now use up to 255 registers
    - Reduces the number of register spills to memory
  - New shuffle instruction that allows swapping thread values within a warp
  - New and improved atomic operations:
    - atomicMin
    - atomicMax
    - atomicAnd
    - AtomicOr
    - AtomicXor
  - Improvements in Texture Memory to improve performance

Memory Hierarchy

- Improvements at the memory hierarchy:
  - Introduction of a new 48KB cache for constant data
    - In Fermi the L1 constant cache could only be accessed through the Texture path
    - Supports unaligned memory accesses
  - The shared/L1 memory now supports an additional configuration:
    - 16KB shared memory + 48KB L1 cache
    - 32KB shared memory + 32KB L1 cache
    - 48KB shared memory + 16KB L1 cache
  - Double data access bandwidth
    - From 128B/clock in Fermi (16 LD/ST units x 32 bits)
    - To 256B/clock in Kepler (to support 32 LD/ST units)
  - 2x increase in the L2 cache:
    - From 768KB (Fermi) to 1536KB (Kepler)

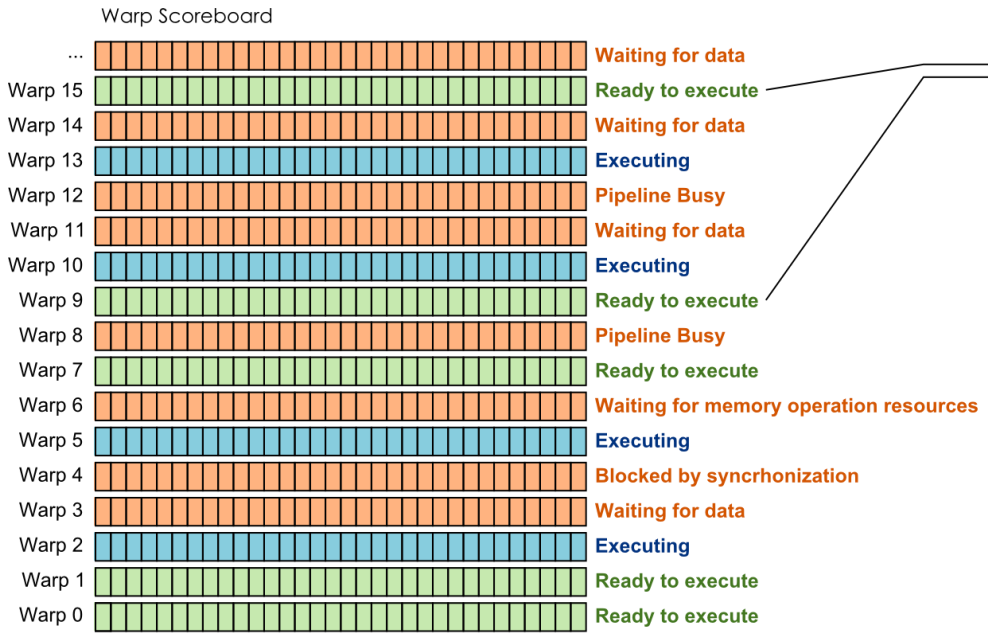
Comparison

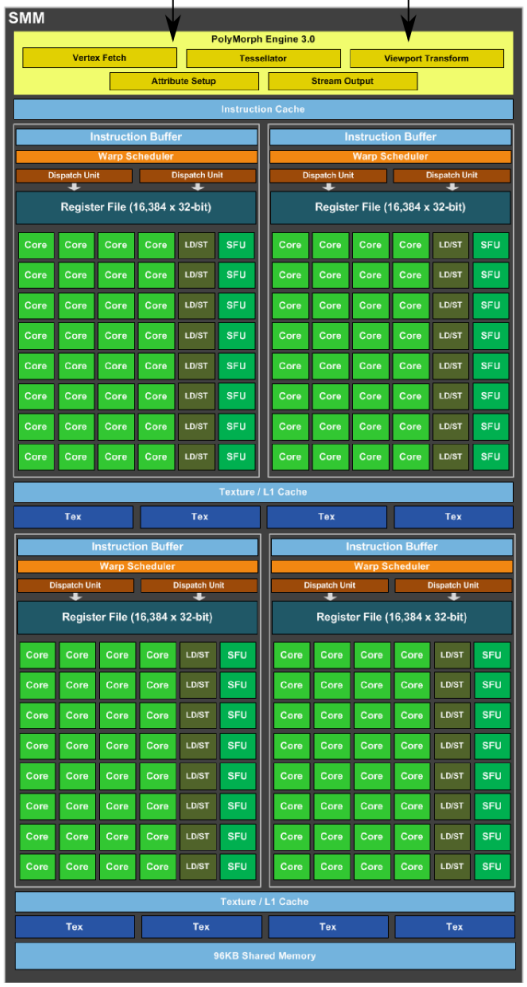
	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
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

NVIDIA Maxwell Micro-architecture

- The Simultaneous Multiprocessor was redesigned
  - Named SM (Fermi), SMX (Kepler), SMM (Maxwell)
- The new SMM was redesigned to optimize the used resources. Hence, the Maxwell SMM features a reduced number of 128 CUDA cores (vs 192 in Kepler)
- The L2 cache increased to 2MB, distributed in 4 blocks of 512KB (one per Graphics Processing Cluster)

Simultaneous Multiprocessor Arch. (SMM)

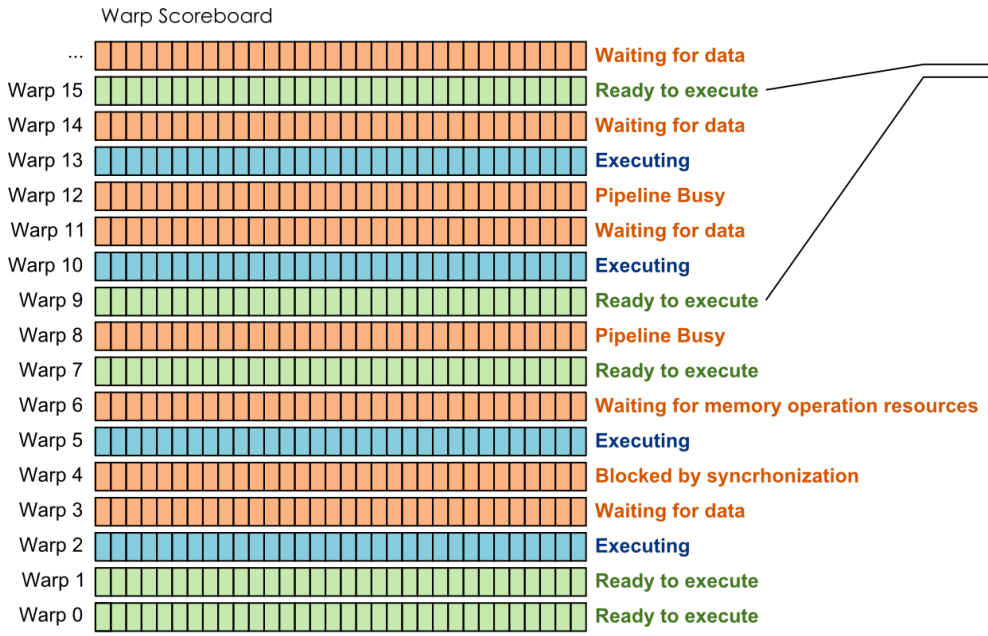


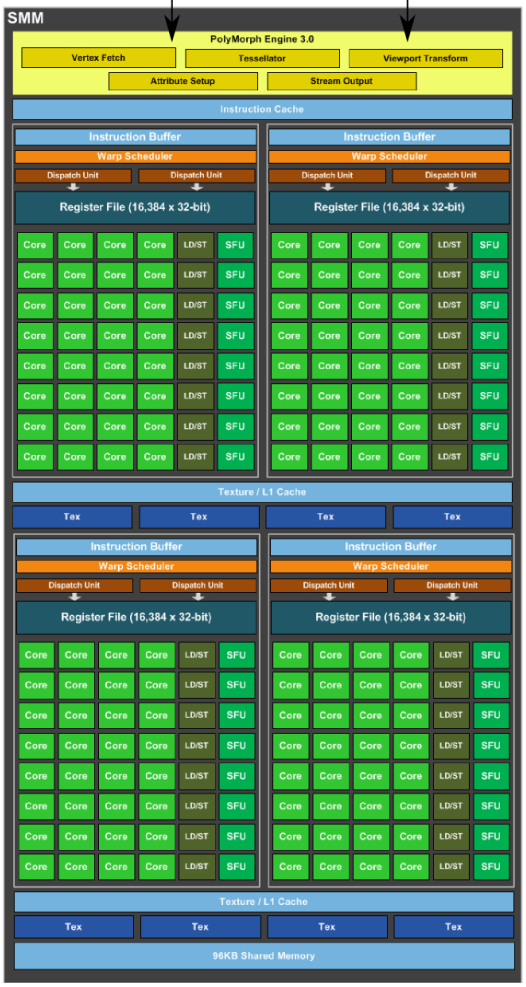


The SM was redesigned in order to simplify control:

- While in previous designs the schedulers shared execution resources, each scheduler now has dedicated execution units
  - Greatly simplifies control
  - Requires additional hardware resources for the execution pipeline, which are now generally less utilized
  - However, this likely decreases the critical path allowing for a increase in operating frequency

Warp scheduling





Several improvements related with the warp scheduling mechanisms

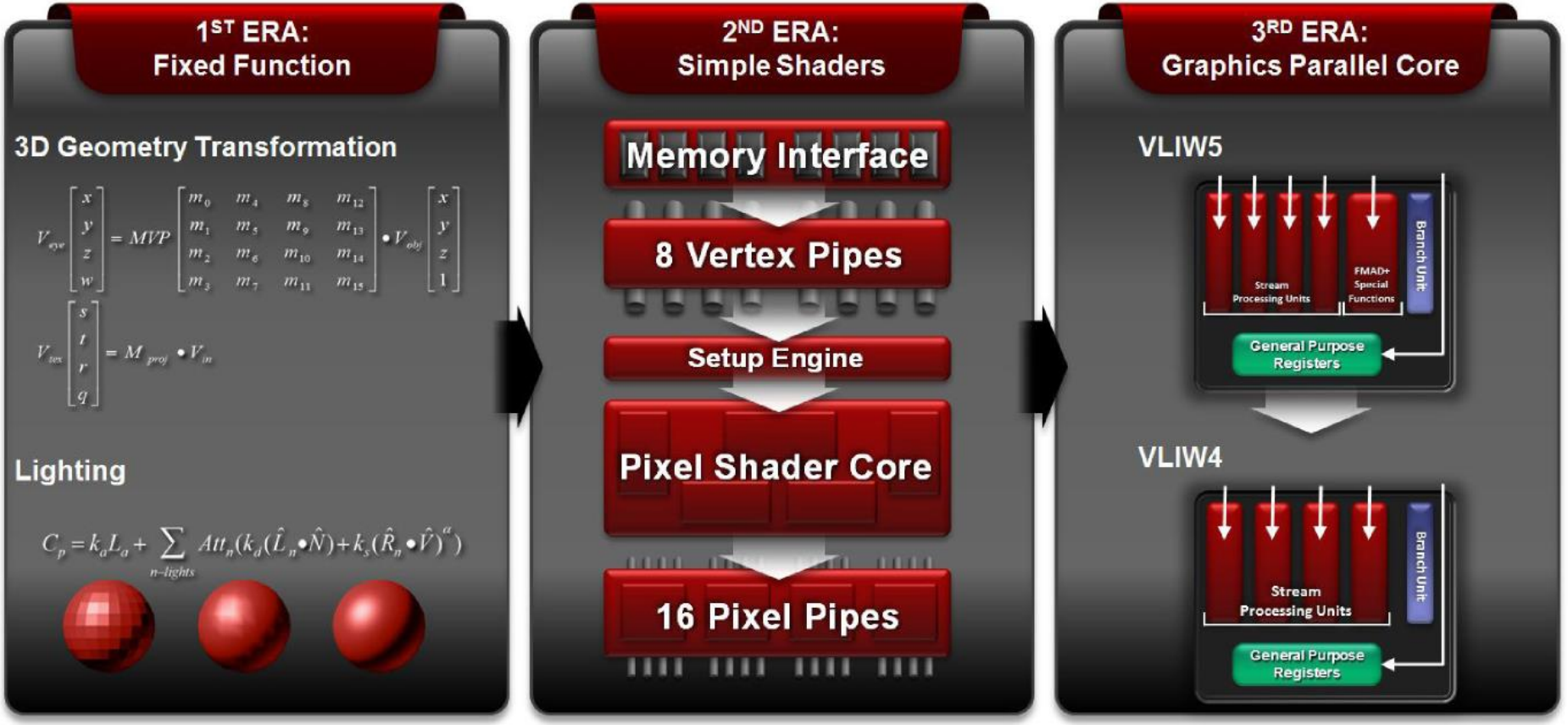
- Stalls down the pipeline are marked at the scoreboard (unlike in Fermi)
  - This suggests a more centralized mechanisms that is able to predict whether the selection of a given warp (which has all operands ready) would result in a pipeline stall

NVIDIA Pascal

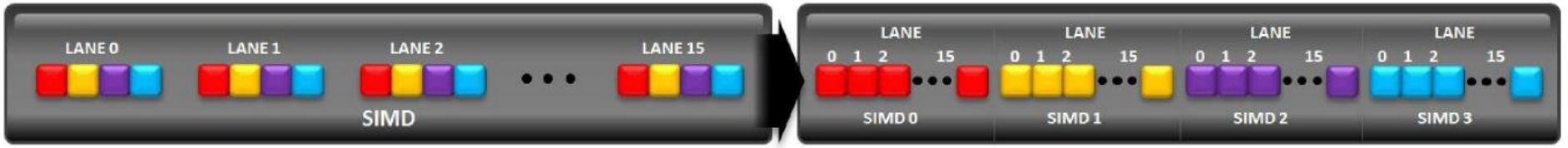
GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

AMD GPU micro-architectures

Recent history



VLIW4 SIMD vs Quad SIMD



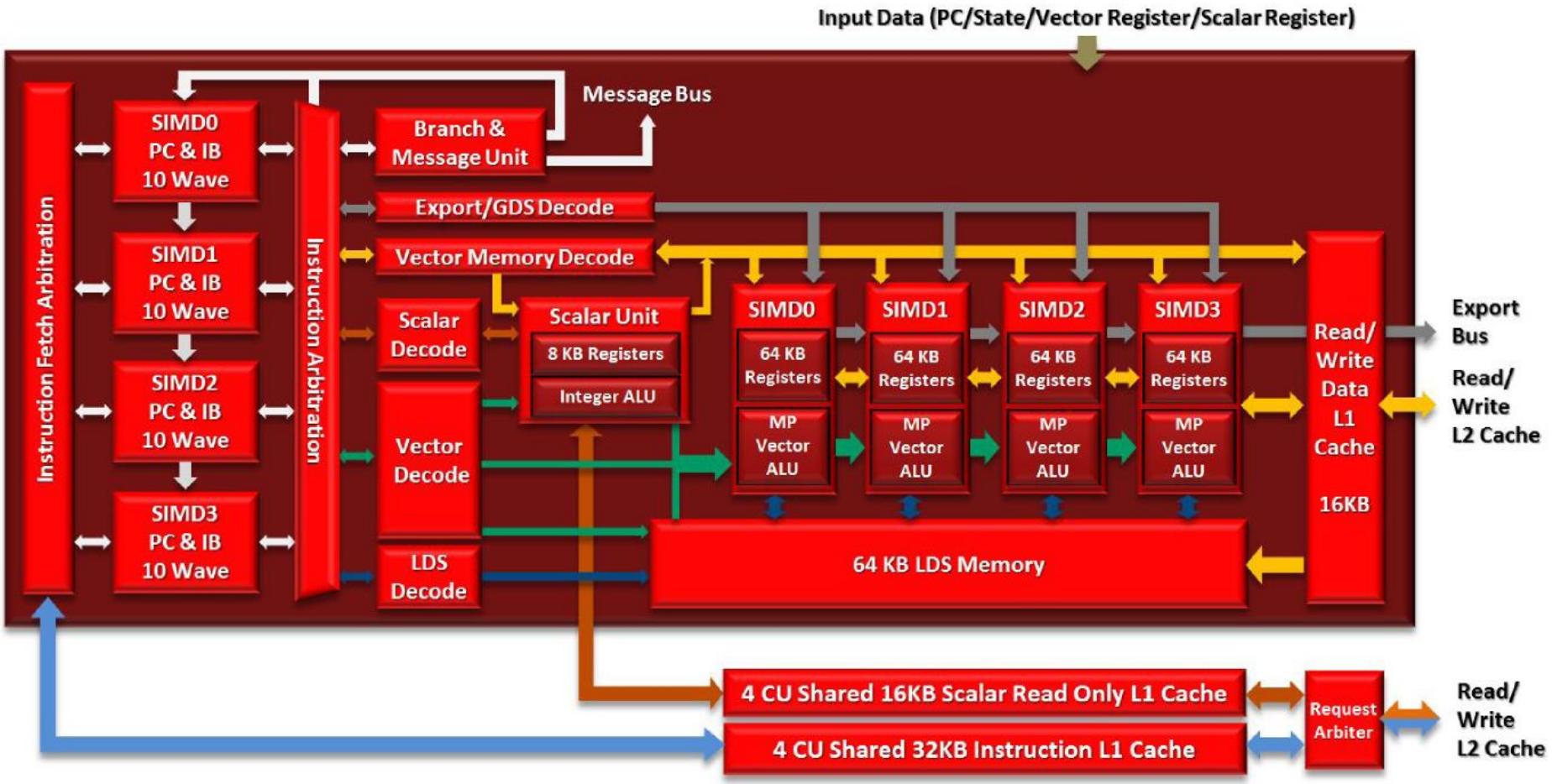
VLIW4 SIMD

- 64 Single Precision multiply-add
- 1 VLIW Instruction × 4 ALU ops → dependency limited
- Compiler manages register port conflicts
- Specialized, complex compiler scheduling
- Difficult assembly creation, analysis, and debug
- Suited for graphics, less flexible for compute
- Careful optimization required for peak performance

GCN Quad SIMD

- 64 Single Precision multiply-add
- 4 SIMDs × 1 ALU op → occupancy limited
- No register port conflicts
- Standardized compiler scheduling & optimizations
- Simplified assembly creation, analysis, and debug
- Simplified tool chain development and support
- Stable and predictable performance

GCN Compute Unit



Architecture comparison

Accelerator	GTX 560 Ti	GTX 660 Ti	GTX 780 Ti	GTX 980	R7 265	R9 290X	5110P
Manufacturer	NVIDIA	NVIDIA	NVIDIA	NVIDIA	AMD	AMD	Intel
Release	2011	2012	2013	2014	2014	2013	2012
Architecture	Fermi	Kepler	Kepler	Maxwell	GCN	GCN	Knights Corner
Cores	384	1344	2880	2048	1024	2816	60
SMs / CUs	8	7	15	16	16	44	
Cores per SM / CU	48	192	192	128	64	64	
Global Memory (MB)	1024	1024	3072	4096	2048	4096	8192
GPU clock (MHz)	1660	1058	1046	1278	925	1000	1053
Memory clock (MHz)	2004	3004	3500	3500	2800	2500	
Peak Memory Bandwidth (GB/s)	128.3	144.2	336	224	179.2	320	320
SP Peak Performance (GFlops/sec)	1263	2843	5040	4612	1894	5632	2022
DP Peak Performance (GFlops/sec)	105	947	1680	144	118	704	1010
L1 Cache Size (KB)	16	16	16	16	16	16	32
L2 Cache Size (KB)	254	393	1536	2097	512	1024	512/core