GPUs and Heterogeneous Systems
(programming models and architectures)

GPU architecture: from the graphics pipeline to the modern architecture

References

- Slides mainly based on:
  - Slides by Prof. Kayvon Fatahalian (Standford University) http://graphics.stanford.edu/~kayvonf/
  - Slides by Prof. Patrick Cozzi (University of Pennsylvania) https://cis565-fall-2020.github.io
  - E. Lindholm and others, “NVIDIA Tesla: A unified graphics and computing architecture”, MICRO 2008
  - NVIDIA whitepapers, http://www.nvidia.com
  - D. Kaeli and others, “Heterogenous Computing with OpenCL 2.0”
  - AMD whitepapers, https://www.amd.com

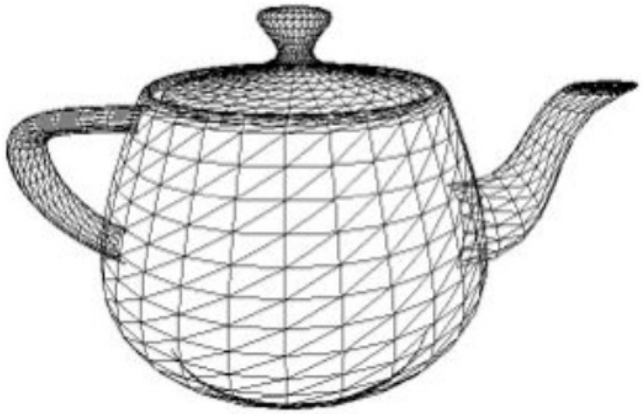
Main steps in GPU evolution

- Real-time graphics pipeline for 3D rendering
- GPU as a HW accelerator for graphics pipeline
- GPU for general purpose computing
- GPU architecture evolution till today

We here focus on NVIDIA architectures, but other vendors followed a similar path

3D rendering

- **3D rendering**: process of converting a 3D model into a 2D image
  - How does each triangle contribute to each pixel in the image?
  - Performed at real-time or offline depending on the application



3D model of the scene (geometry, materials, lights, camera, …)

2D image displayed on the screen

3D rendering

- 3D rendering requires various steps to be executed in sequence:
  - Definition of single objects in the scene
  - Placement of the various objects in the overall scene
  - Definition of the viewpoint, lighting, …
  - Identification of portion of the scene actually seen by the viewpoint
  - Deletion of all hidden surfaces
  - Definition of the color of each point seen by the viewpoint
  - Mapping the seen scene on the 2D screen

Graphics pipeline for 3D rendering

- **Graphics pipeline**: conceptual model describing what steps need to be performed in 3D real-time rendering
  - Standardized around OpenGL/Direct3D APIs
    - Libraries used to separate 3D rendering from application logic
    - Possibly accelerated in HW

System stack:

Application
(e.g., videogame)

↓

Scene model
(objects, materials, lights, …)

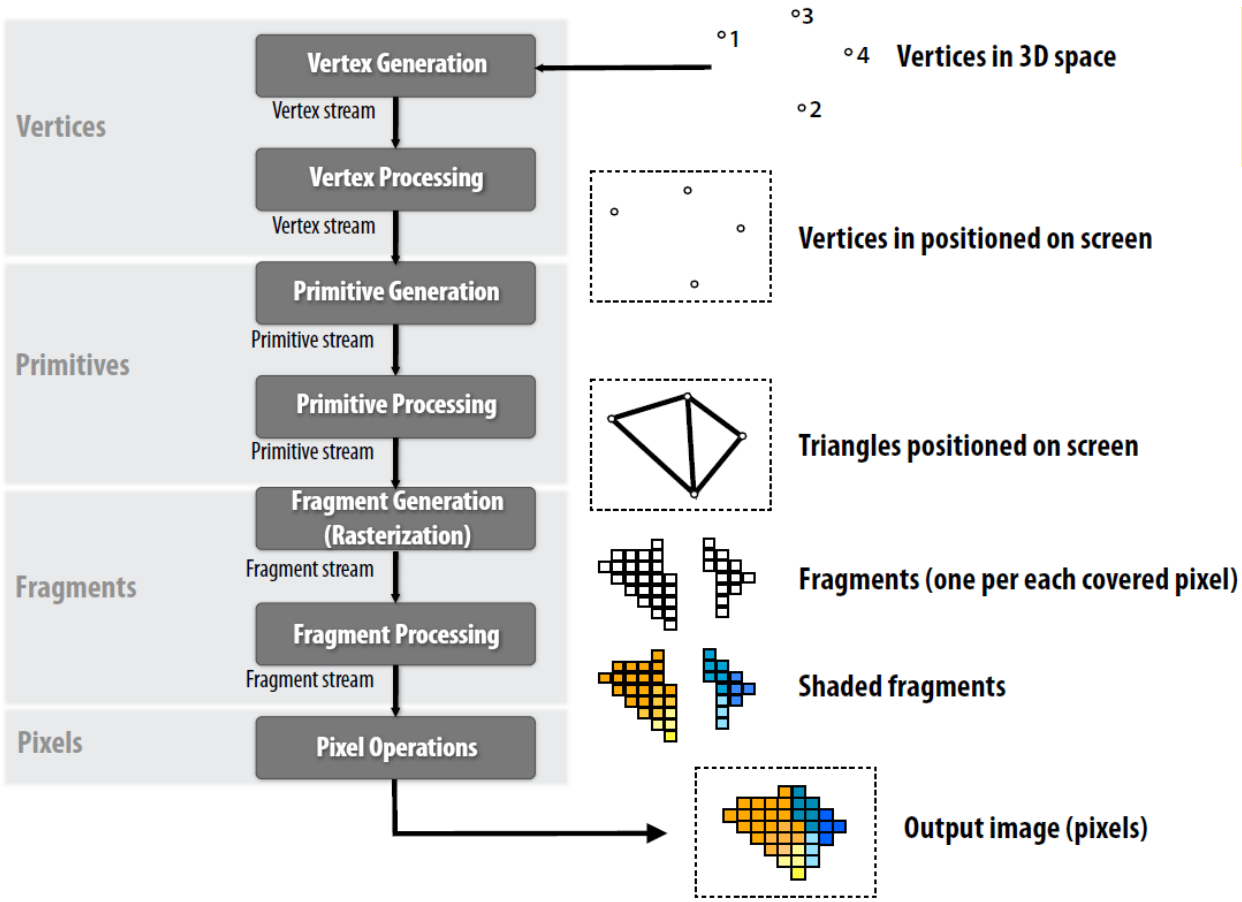
↓

Graphics pipeline
(OpenGL/Direct3D)

↓

Graphics pipeline implementation
(software driver + GPU)

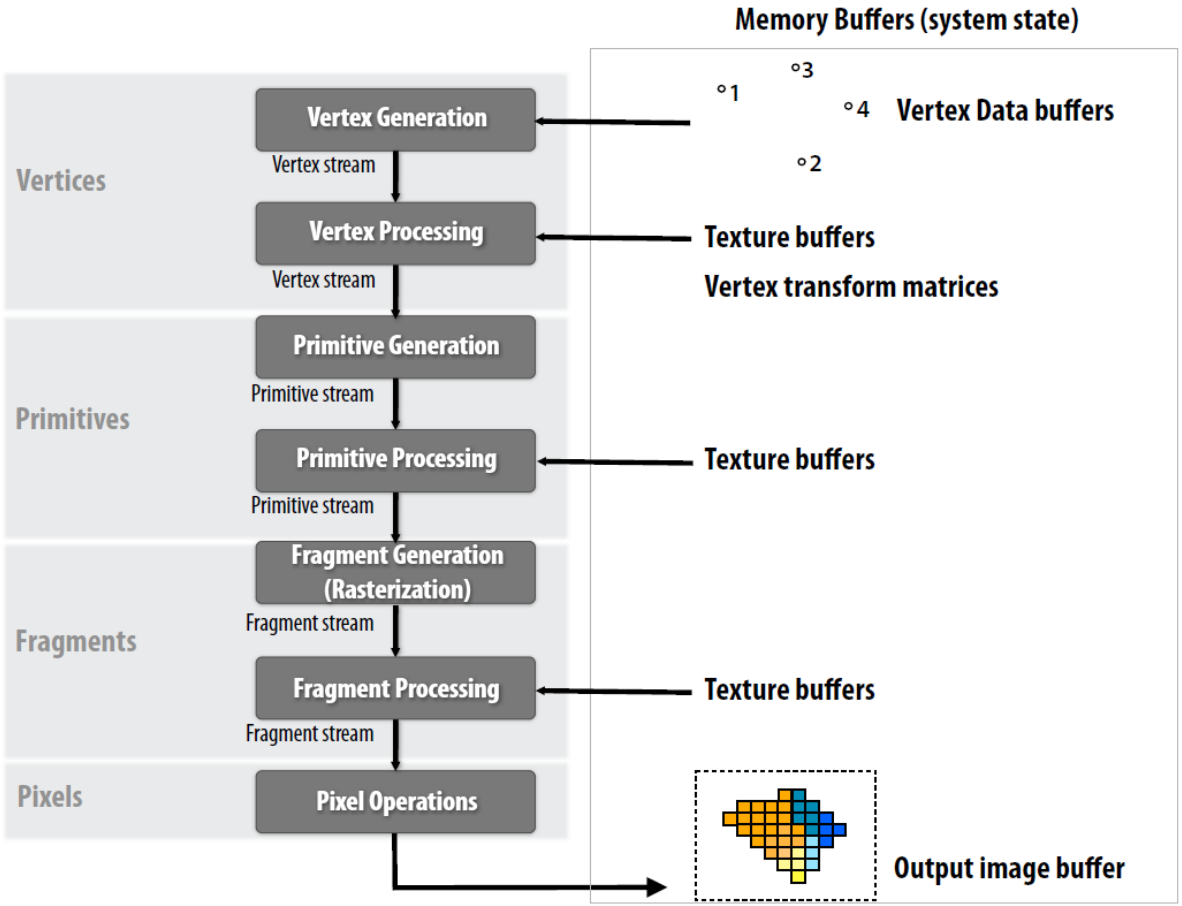
Real-time graphics pipeline



Sequence of operations on several types of objects

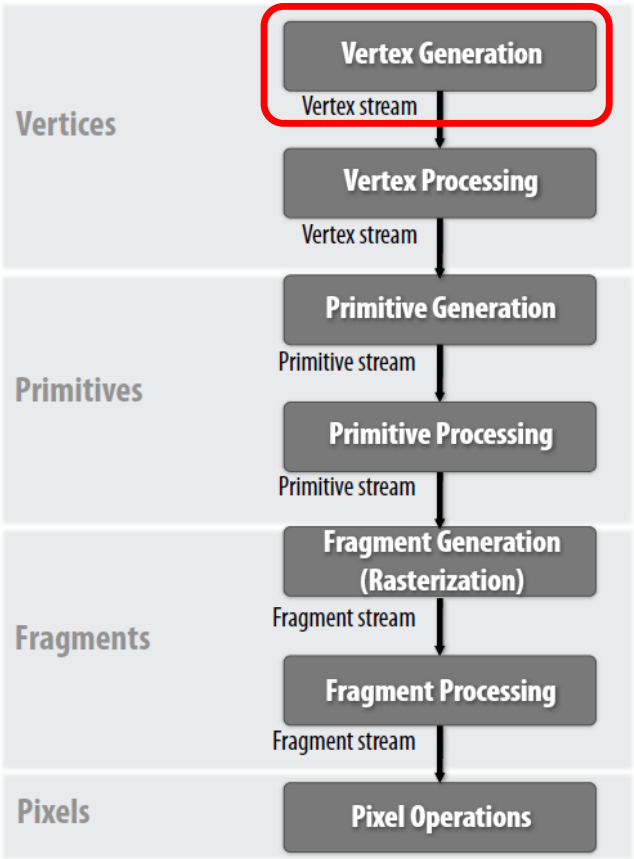
This is the graphics pipeline implemented by OpenGL before 2007

Real-time graphics pipeline



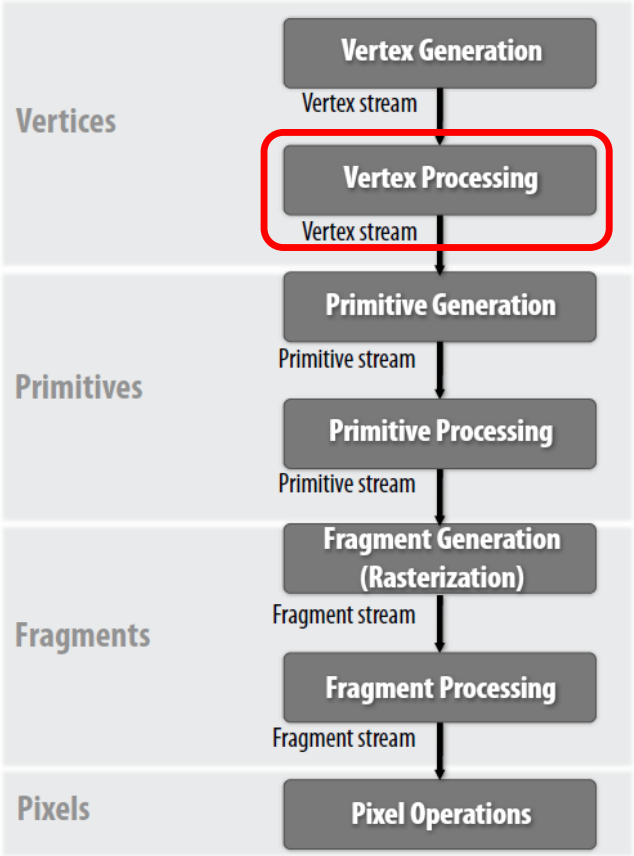
Each operation accesses different data buffers

Vertex generation



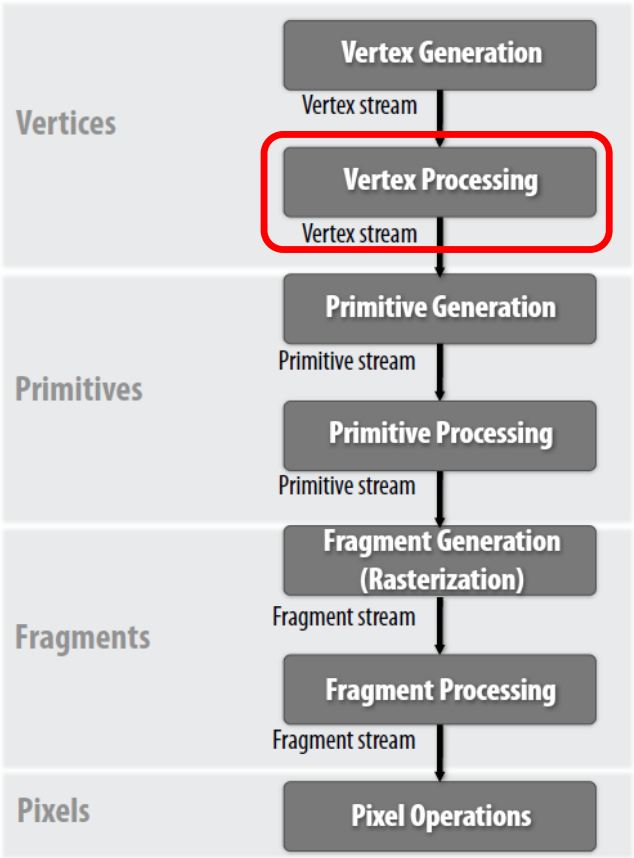
- The host interface is the communication bridge between the CPU and the graphics unit (e.g., GPU)
- It receives commands from the CPU and pulls geometry information from system memory
- It outputs a stream of vertices in object space with all their associated information (normals, texture coordinates, per vertex color, etc.)

Vertex processing

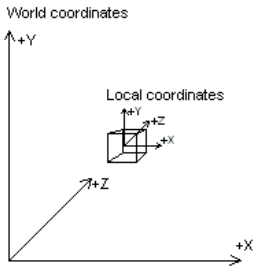


- The vertex processing stage receives vertices from the host interface in object space and outputs them in the clip space

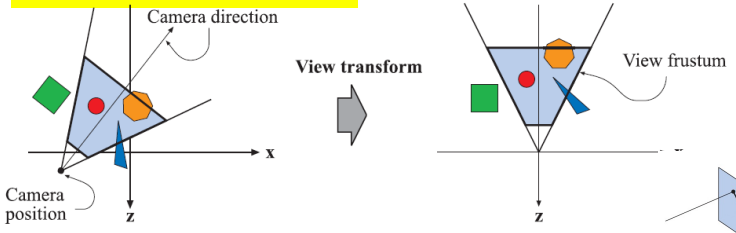
Vertex processing



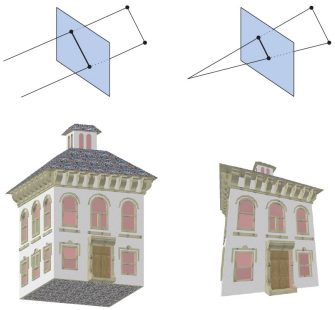
1. Model to world coordinates



2. World to eye coordinates

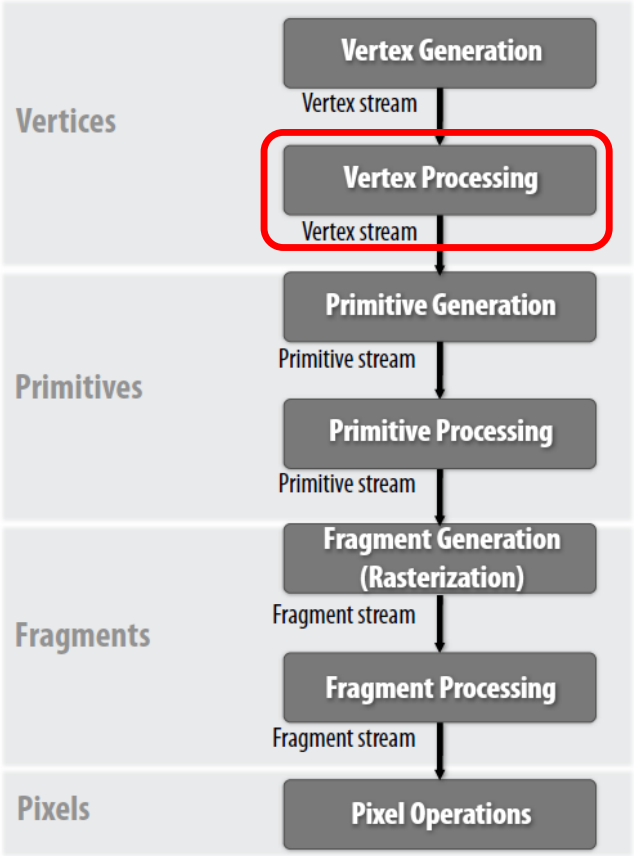


3. Eye to clip coordinates

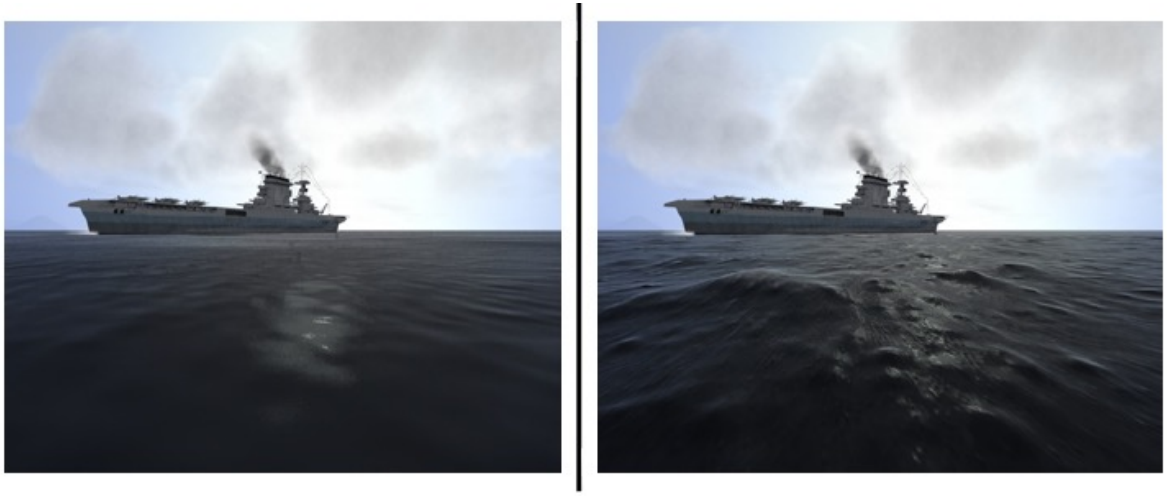


Matrix multiplications on each vertex

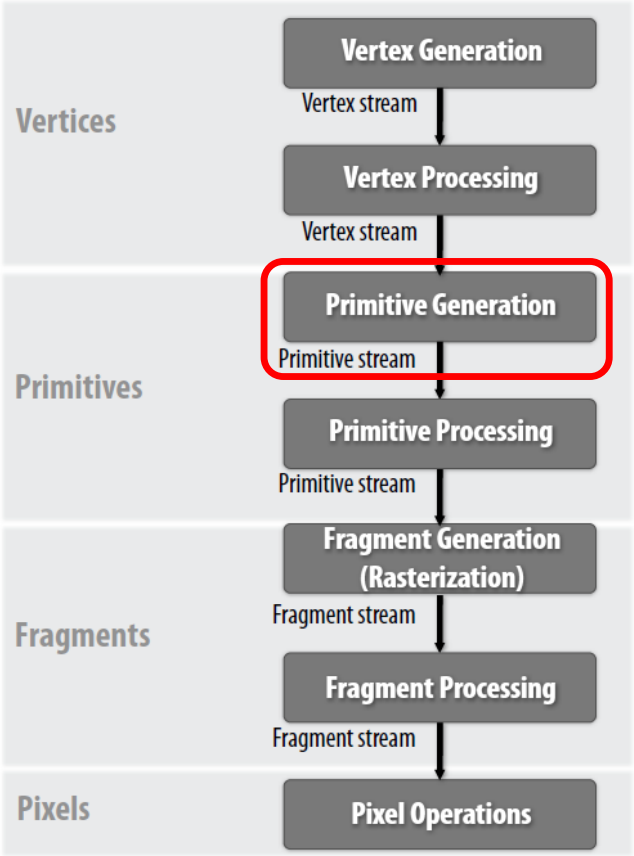
Vertex processing



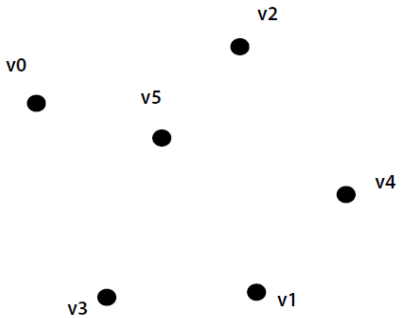
- Textures may be also used for advanced transformations on vertices (e.g., displacement mapping)

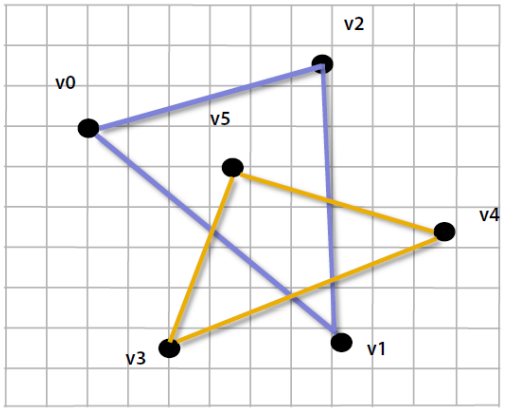


Primitive generation

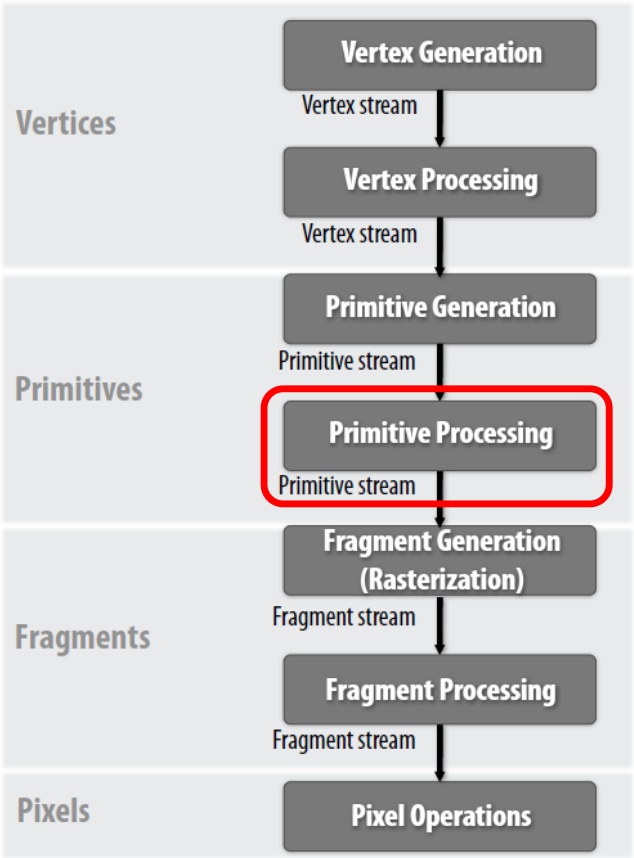


- The primitive assembler groups vertices forming one primitive (i.e., a triangle)

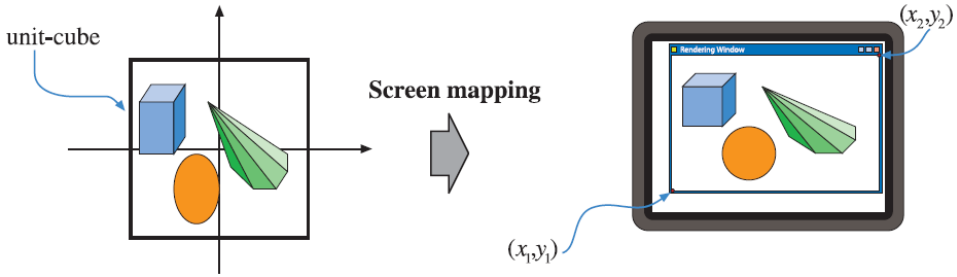




Primitive processing

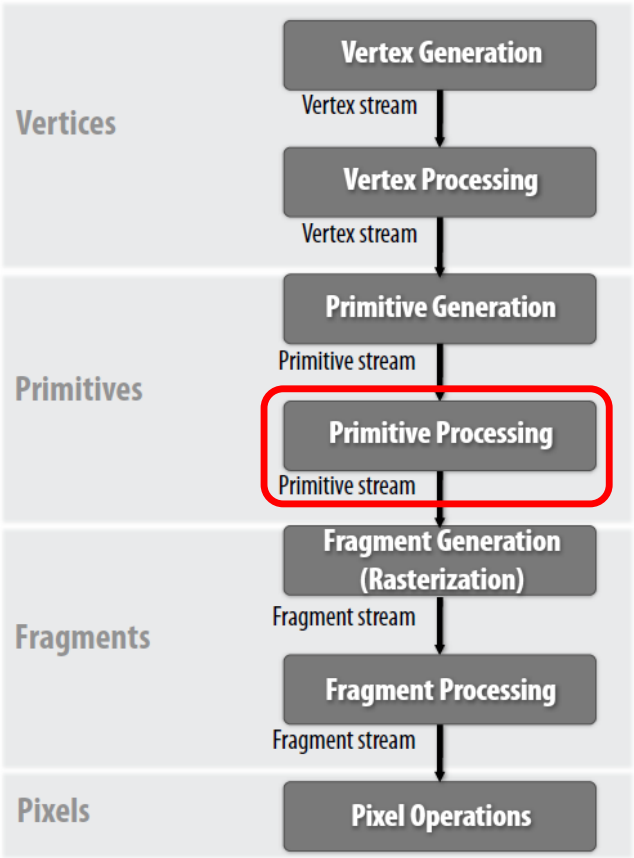


- Various elaborations are performed
  - Perspective division and viewpoint transformation

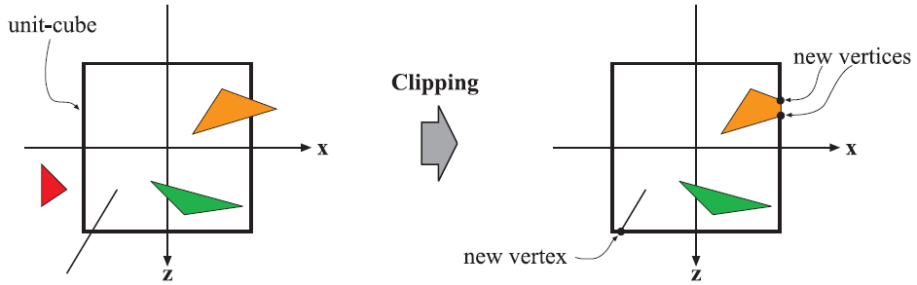


Scalar math operations and matrix multiplications on each vertex

Primitive processing

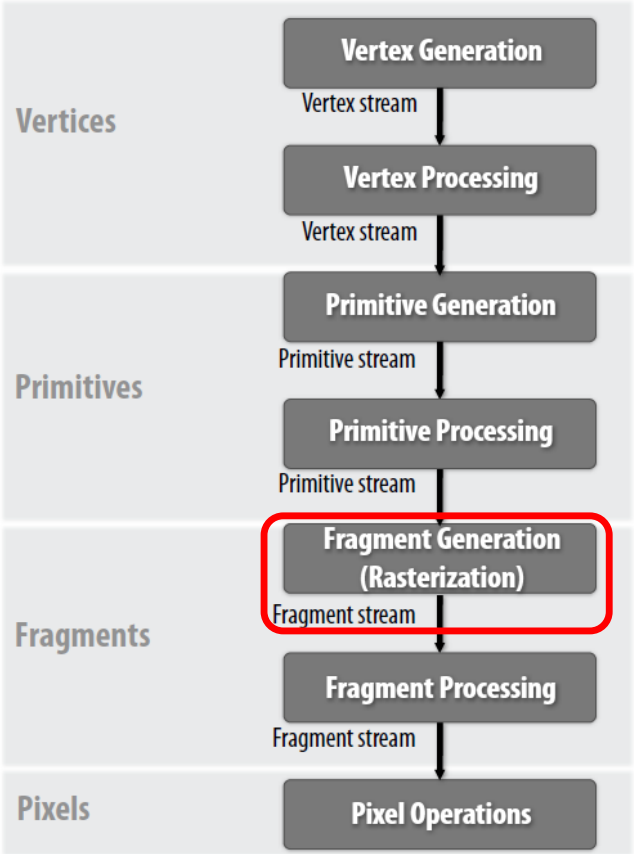


- Various elaborations are performed
  - Clipping and backface culling

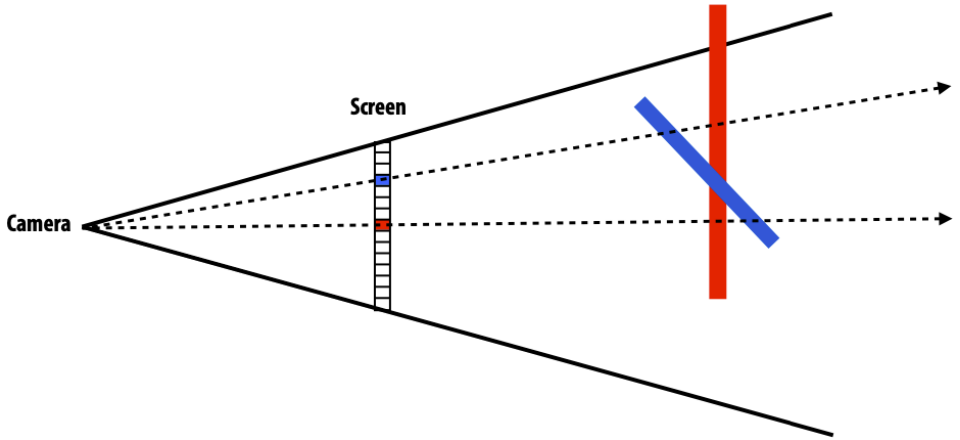


More advanced inclusion tests on primitive coordinates and math operations on vectors

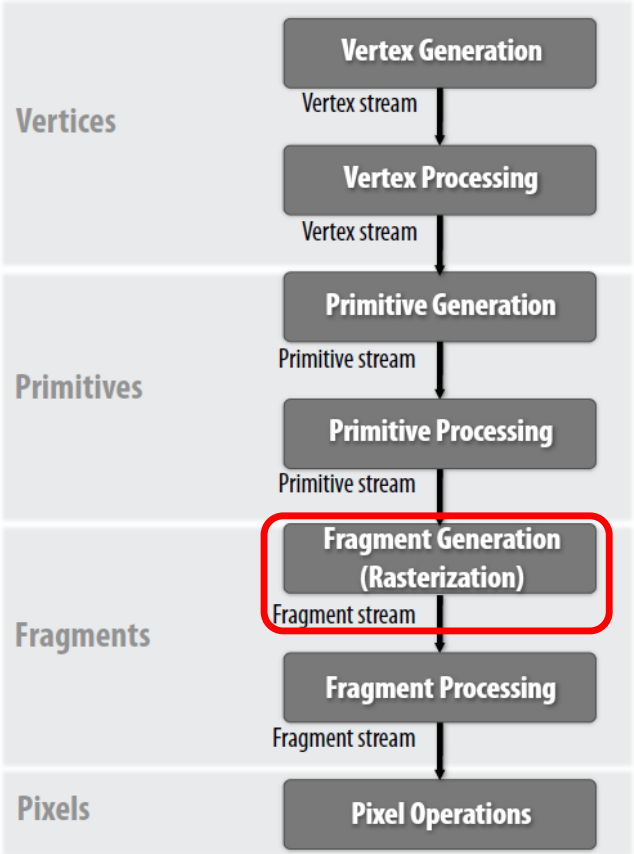
Fragment generation



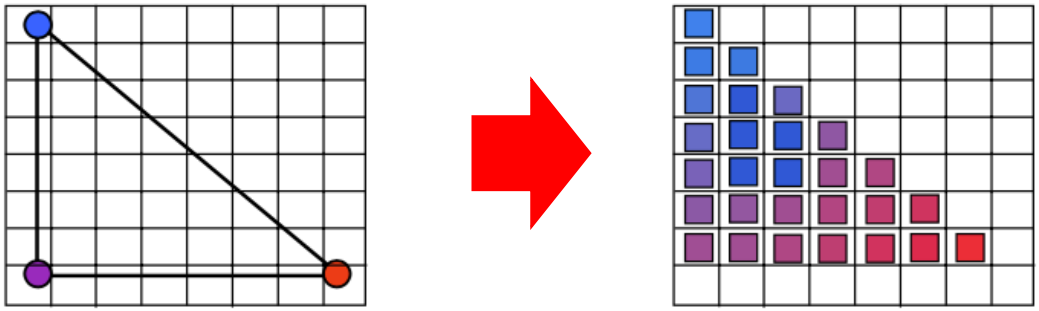
- Rasterization converts each primitive into fragments
  - Determine which pixels a primitive overlaps
  - Computes preliminary fragment color



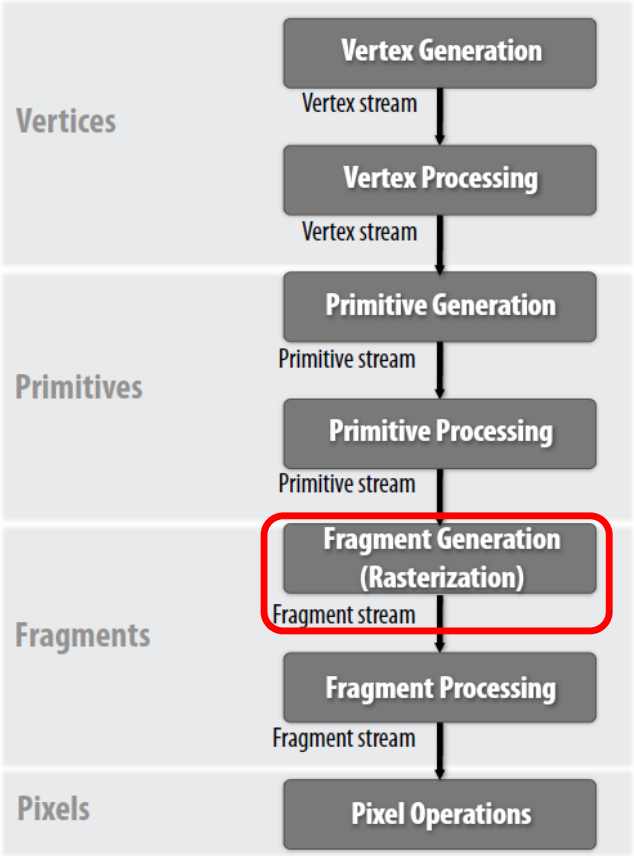
Fragment generation



- Rasterization converts each primitive into fragments
  - Determine which pixels a primitive overlaps
  - Computes preliminary fragment color

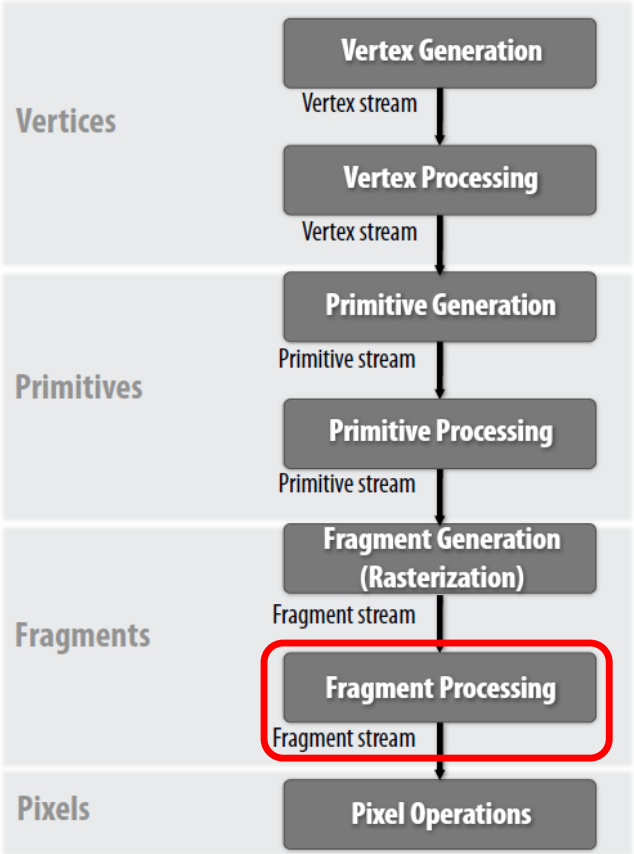


Fragment generation

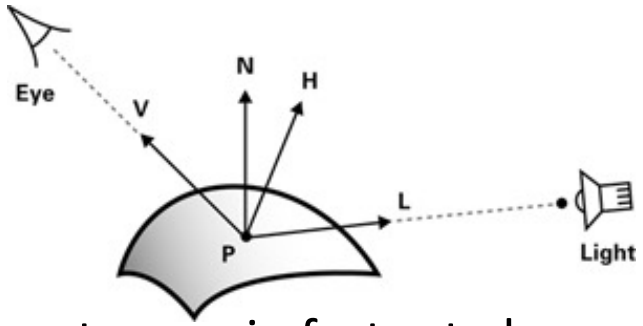


- A fragment contains data to determine how a primitive contributes to color 1 pixel
  - Screen coordinates
  - Depth coordinate of the primitive at sample point
  - Surface normal
  - Texture coordinates

Fragment processing



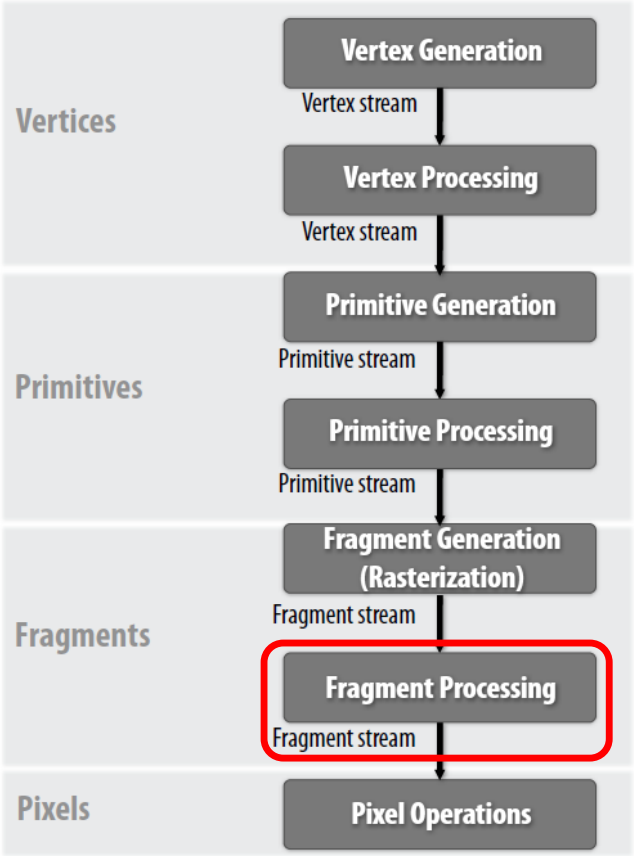
- Assign final colors to fragments
  - Shades the fragment by simulating the interaction of light and materials



Computes a gain factor to be multiplied against the color of the pixel

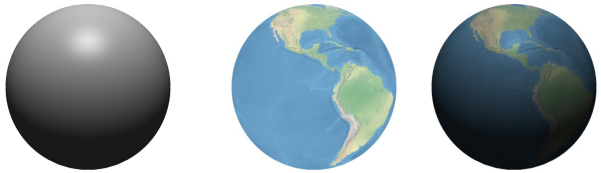
Math operations on vectors

Fragment processing



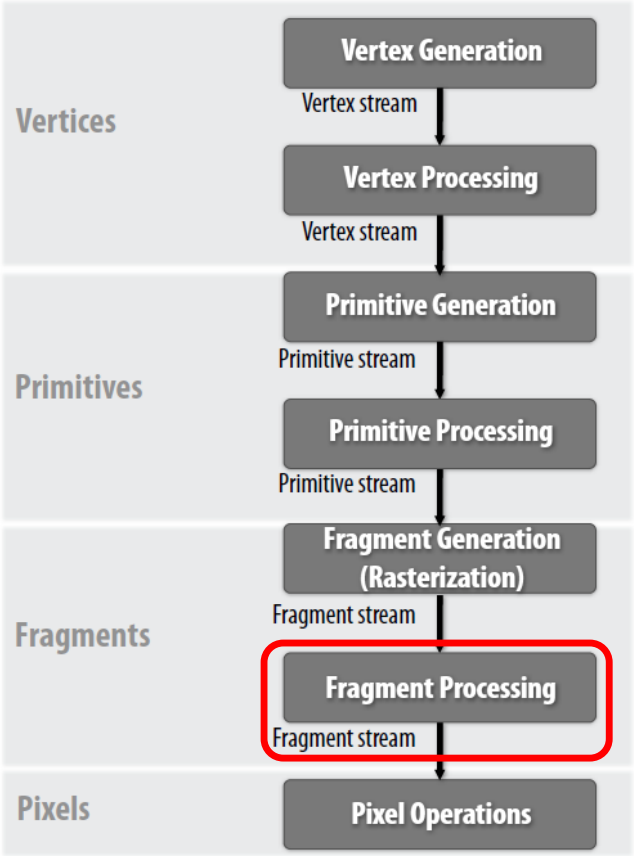
- Texture mapping

- Lightning and texture

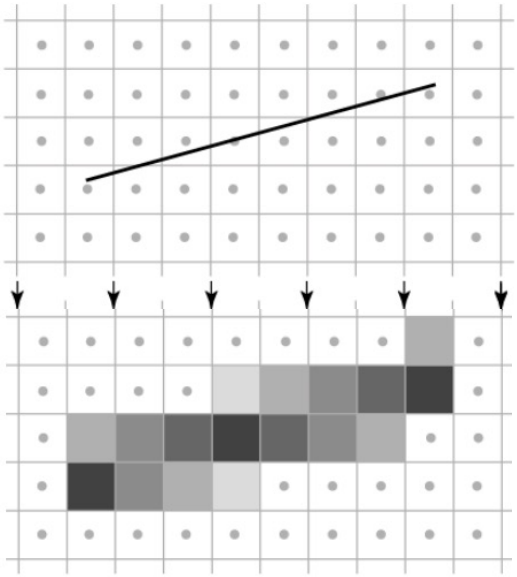


Math operations on vectors and scalars

Fragment processing



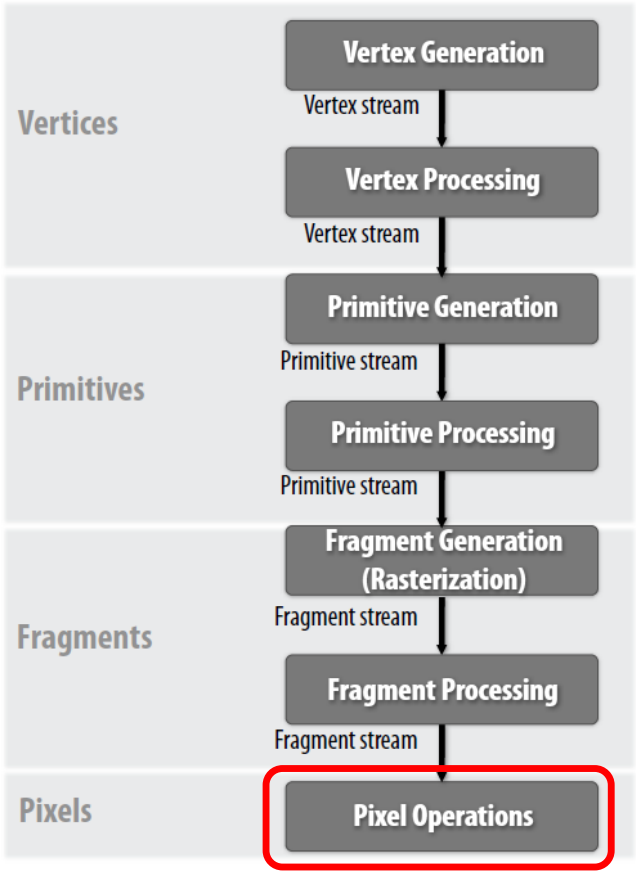
- Anti-aliasing



- …and other elaborations…

Math operations on vectors and scalars

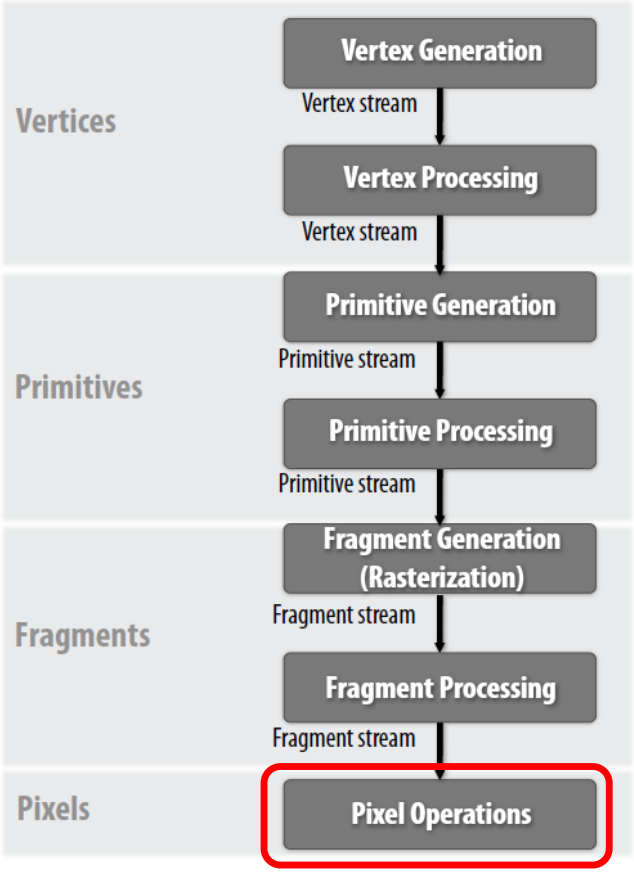
Pixel operations



- Fragments are finally processed to compute final color of each screen pixel
  - Blending: consider transparencies in pixel coloring
  - Z-buffer test: reject hidden fragments
  - Other tests…

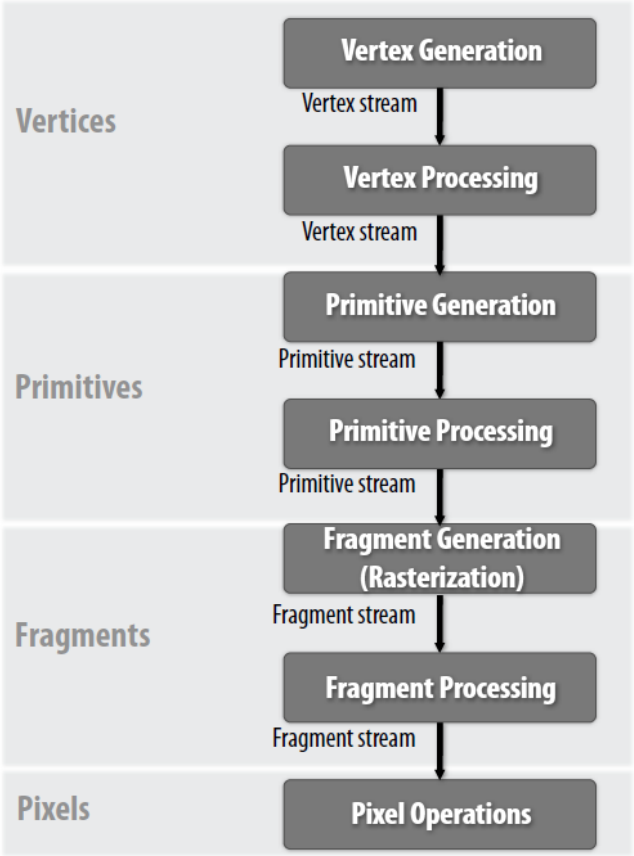
More advanced tests on fragments and math scalar operations

Pixel operations



- Finally, pixels are copied in the framebuffer (the memory space connected to the screen controller)

Considerations on the real-time graphics pipeline



- Huge amount od data to be processed
- Single-stage elaborations can be performed in parallel on each chunk of data (vertex, fragment, pixel,…)
- Single-stage elaborations are mainly data-intensive, based on matrix elaborations and other arithmetical operations
- Few branch instructions
- Some data dependencies

PARALLELISM!

Quest for accelerating graphics pipeline processing

- Hardware accelerators have been designed to accelerate the graphics pipeline

- Evolution of graphics pipeline accelerators
  - Pre-GPU era
  - Fixed-function GPU
  - Programmable GPU
  - GPU based on unified shared processors

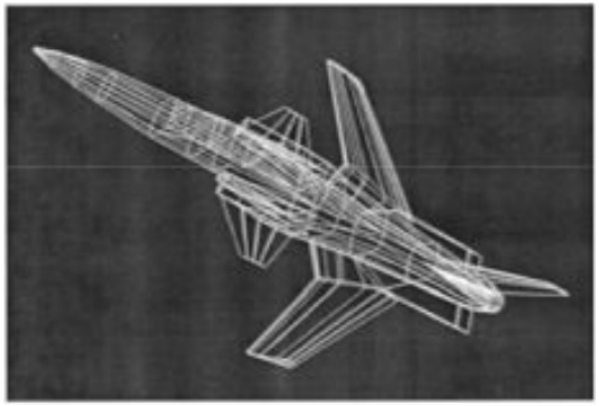
Pre-GPU era

- In the 70/90s graphics supercomputers have been designed to accelerate the graphics pipeline
- Such solutions are mainly targeted specific market sections such as computer simulation, digital content creation, engineering and research
  - Target customers: animation/movie industry, aerospace/military companies, universities, …
- Expensive solutions with a confined market!

Pre-GPU era

- 70s: Machines have been built for flight simulators
- 80s: ASIC accelerators have been designed for geometry processing

Prior to 1987
wireframes

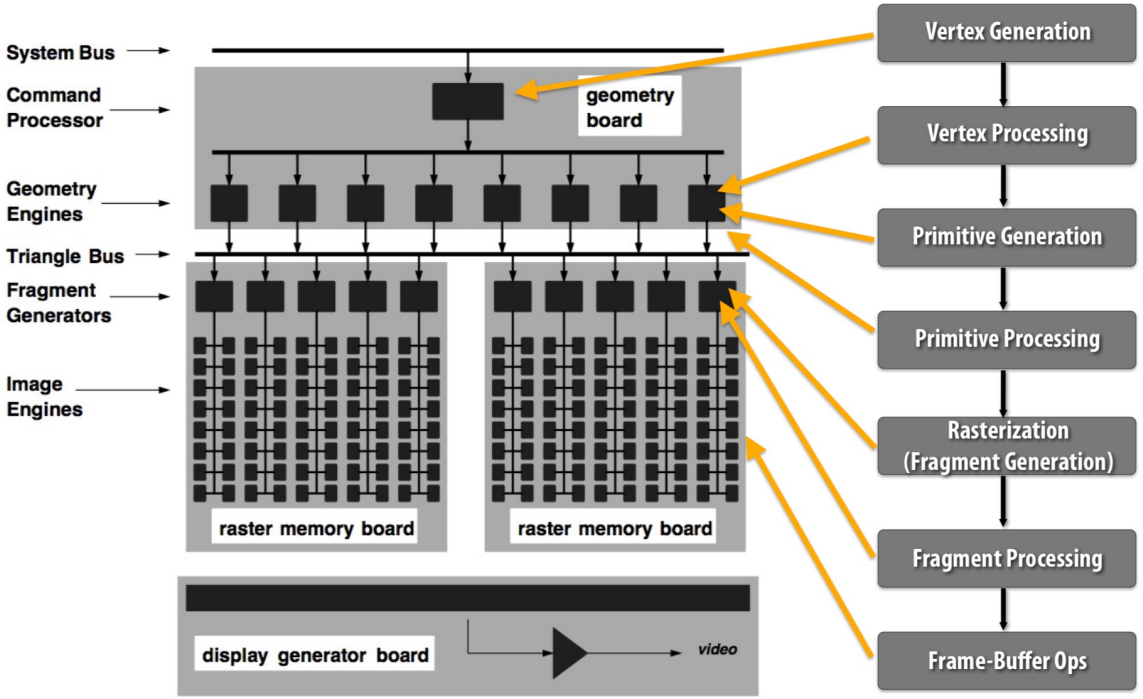


1987 - 1992
Shaded solids

After 1992
Texture mapping

Pre-GPU era

- SGI RealityEngine (1993)
  - Composed of various boards
  - Comprising both ASIC accelerators and general-purpose processors
  - Programmable
  - Compliant with OpenGL



Pre-GPU era – early 90s

- Interactive software rendering (no GPUs yet)
- Note: SGI was building interactive rendering supercomputer, but this was the beginning of interactive 3D graphics on PC

Wolfenstein 3D, 1992

Doom, 1993

Pre-GPU era – early 90s

- Gaming market was growing very fast
  - Quest for more realism -> need for more computational power to perform 3D rendering

- Idea of designing an accelerator to be integrated in the desktop computers to process the graphics pipeline
  - Standardized around OpenGL/Direct3D

Goals of GPUs?

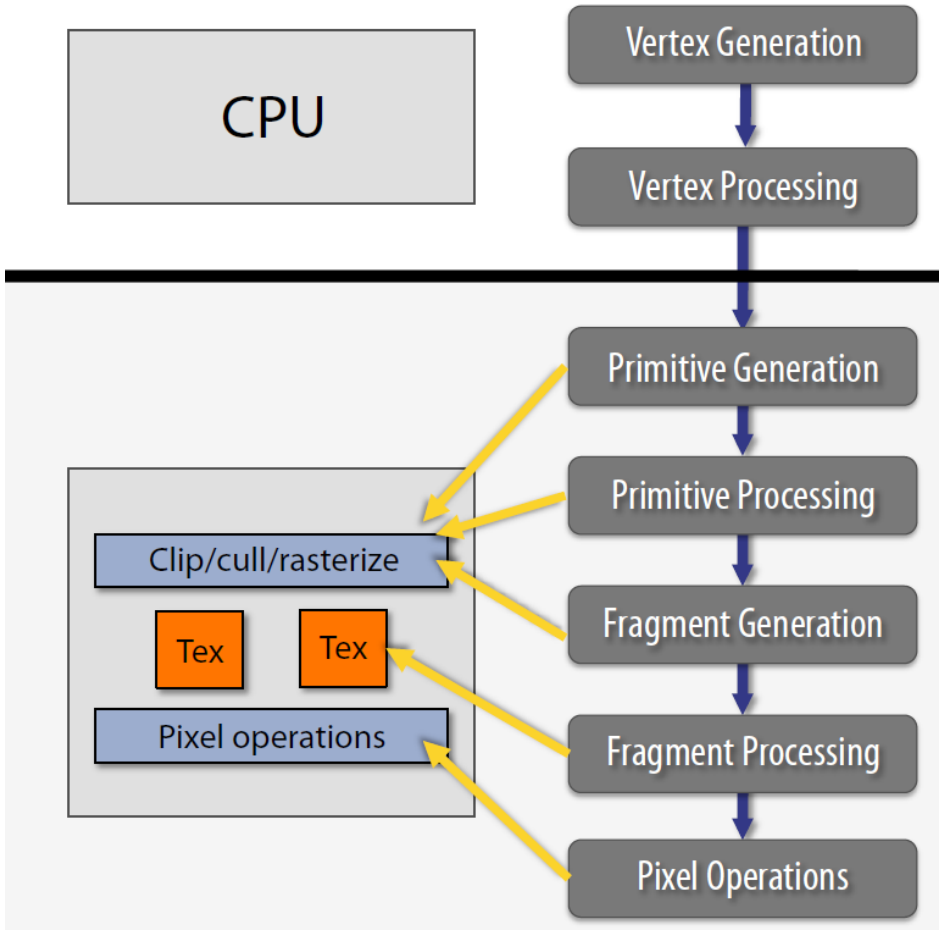
- The Graphics Processing Unit (GPU) is a HW accelerator (initially designed) for the graphics pipeline

- Exploit parallelism
  - Among the pipeline steps
  - In the data processing of the single step
  - By running different tasks on CPU and GPU in parallel

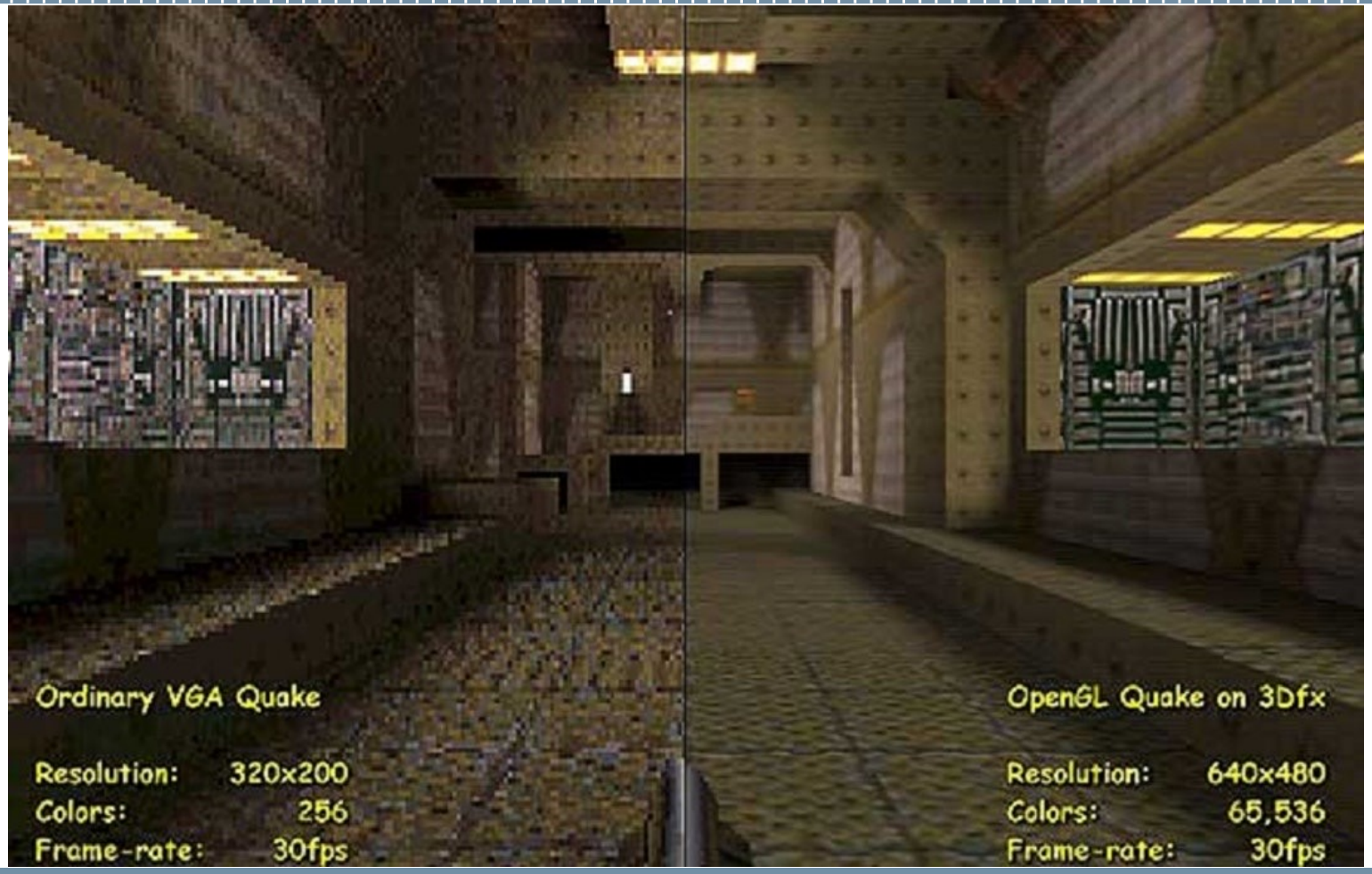
- Use ad-hoc hardware accelerators for recurrent functions
  - Texture filtering, rasterization, MAC, sqrt, ...

3dfx voodoo (1996)

- Fixed function rasterization, texture mapping, depth testing, etc.
- Required separate VGA card for 2D



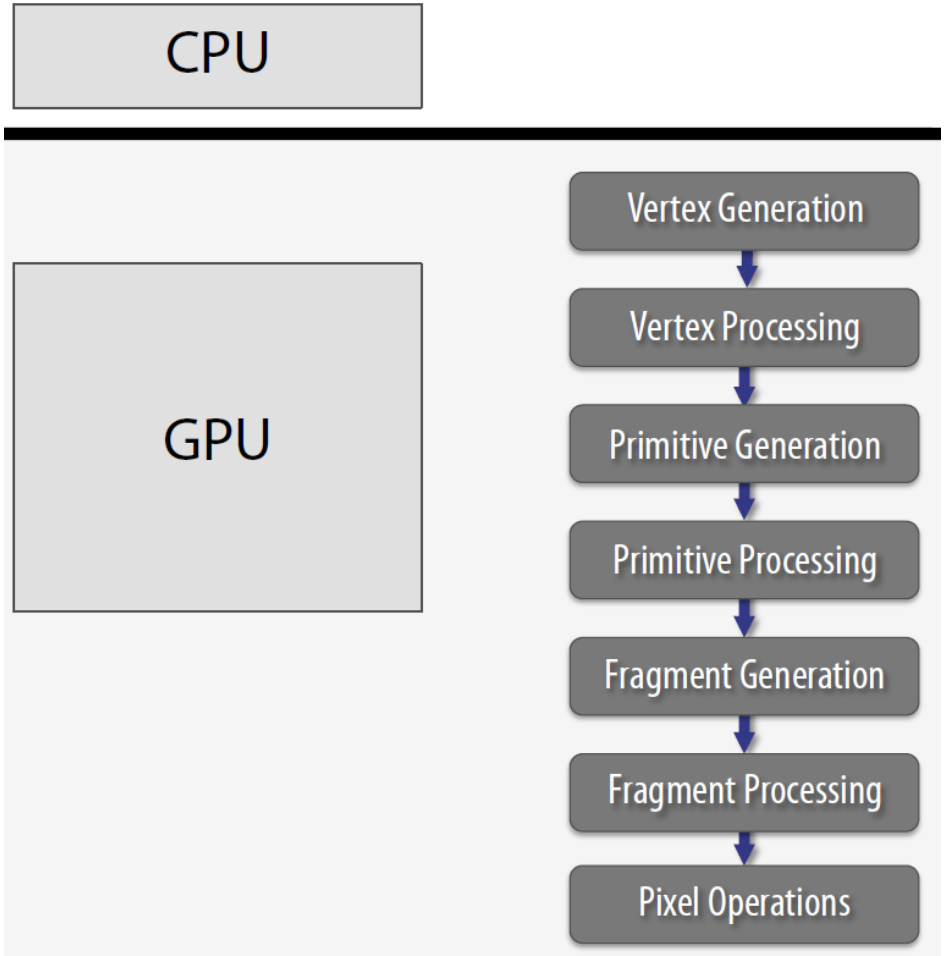
3dfx voodoo (1996)



NVIDIA GeForce 256 (1999)

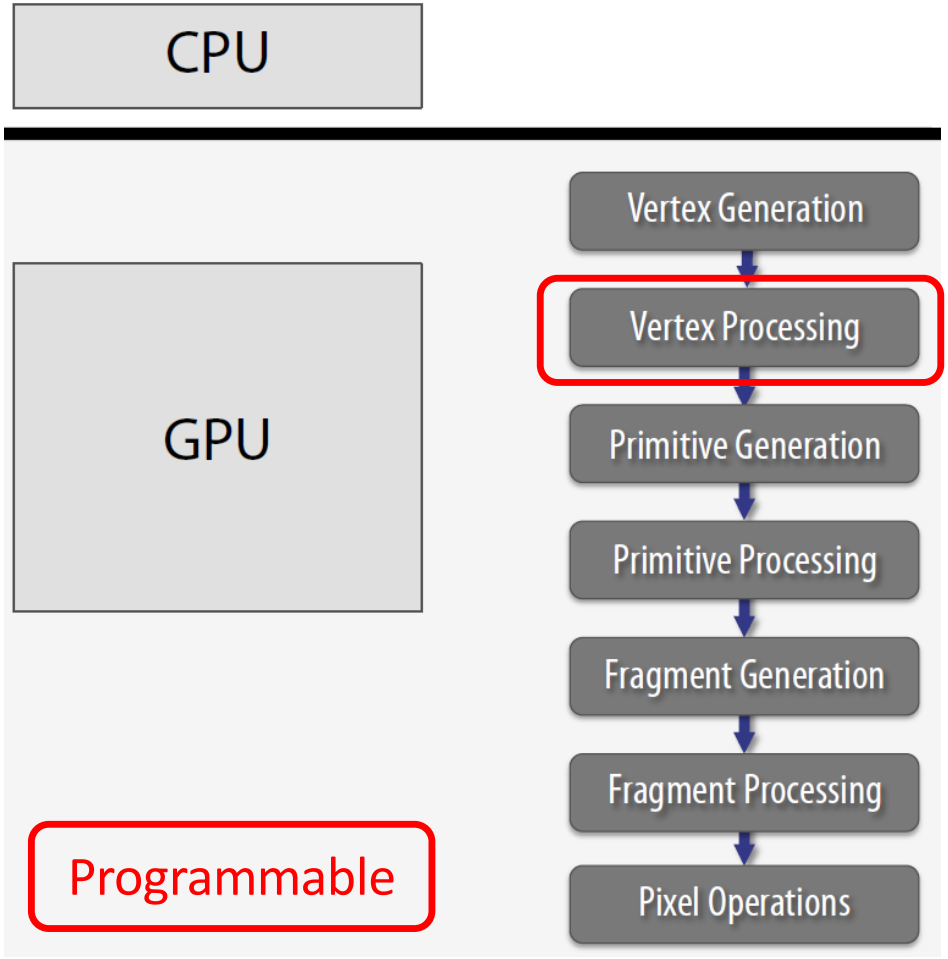
- All stages implemented in hardware
- Fixed function rasterization, texture mapping, depth testing, etc.

NVIDIA used for the first time the term "Graphics Processing Unit" (GPU)



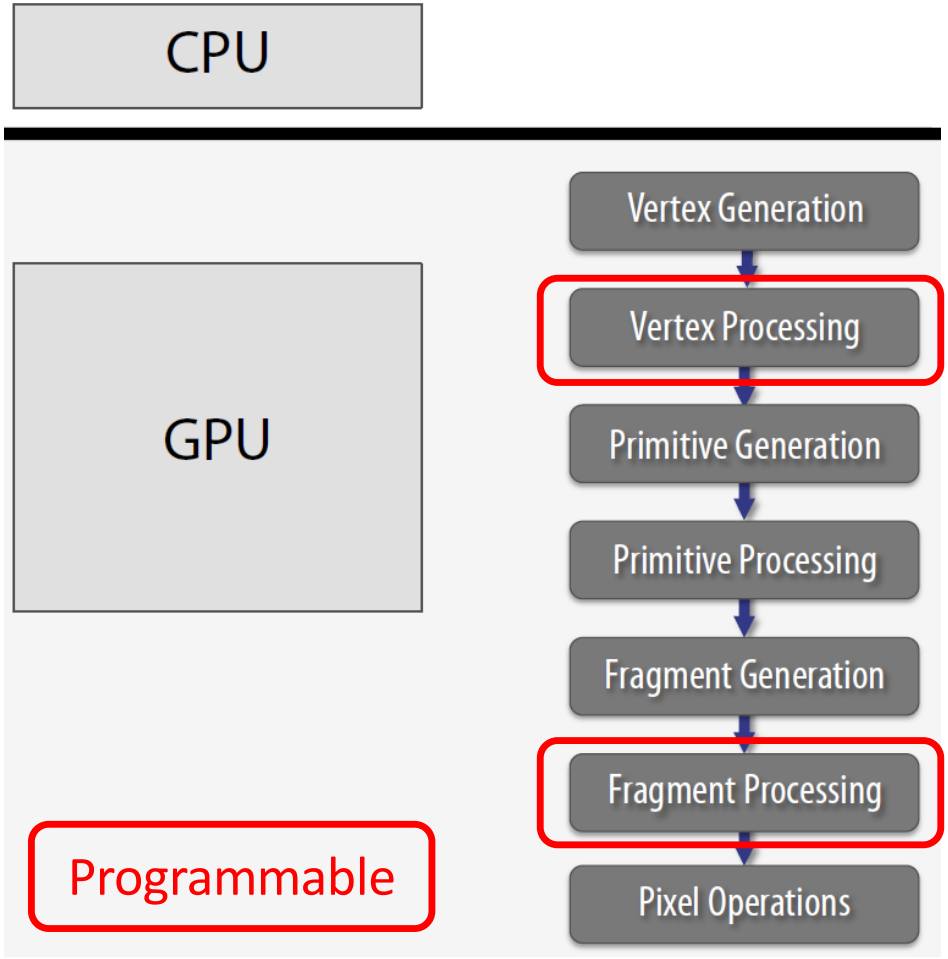
NVIDIA GeForce 3 (2001)

- Optionally bypass fixed-function with a programmable vertex shader
- Shader: a “mini-program” defining the logic of a pipeline stage
  - A specific shading language is used (OpenGL or Direct3D)



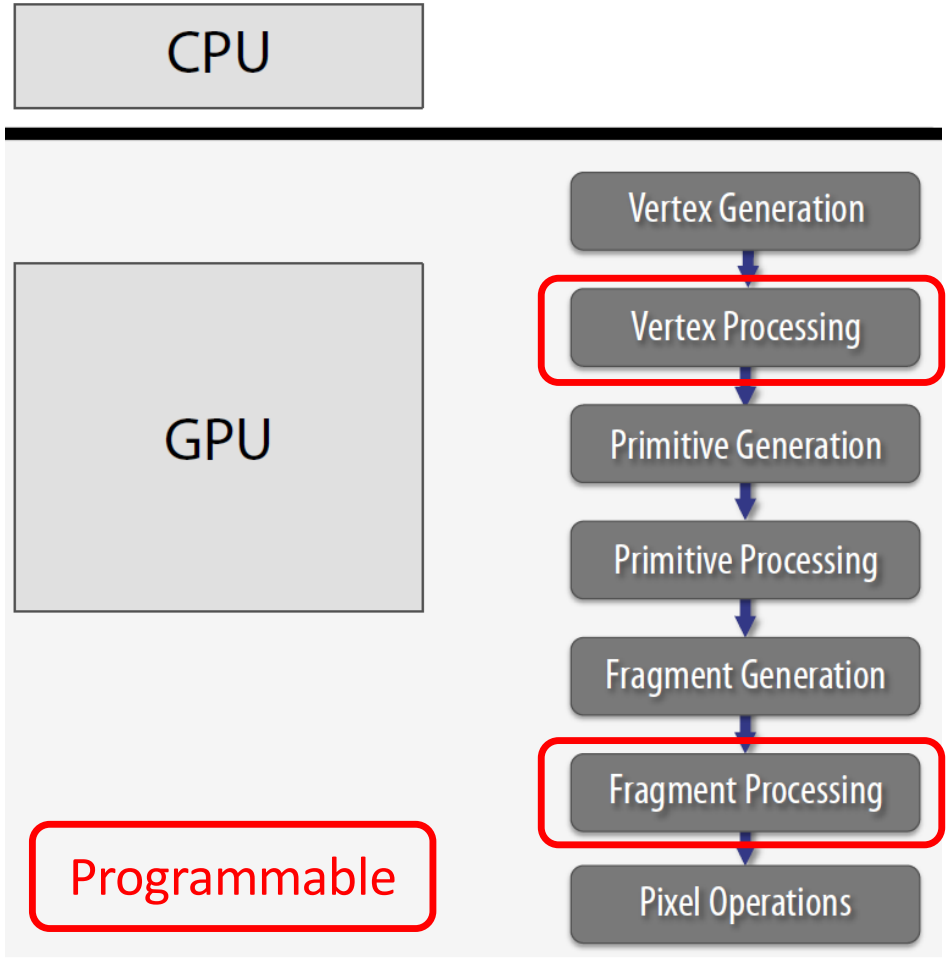
NVIDIA GeForce 6 (2004)

- Improved programmability in fragment shader
- Vertex shader can read textures
- Dynamic branches



How to run code on a GPU

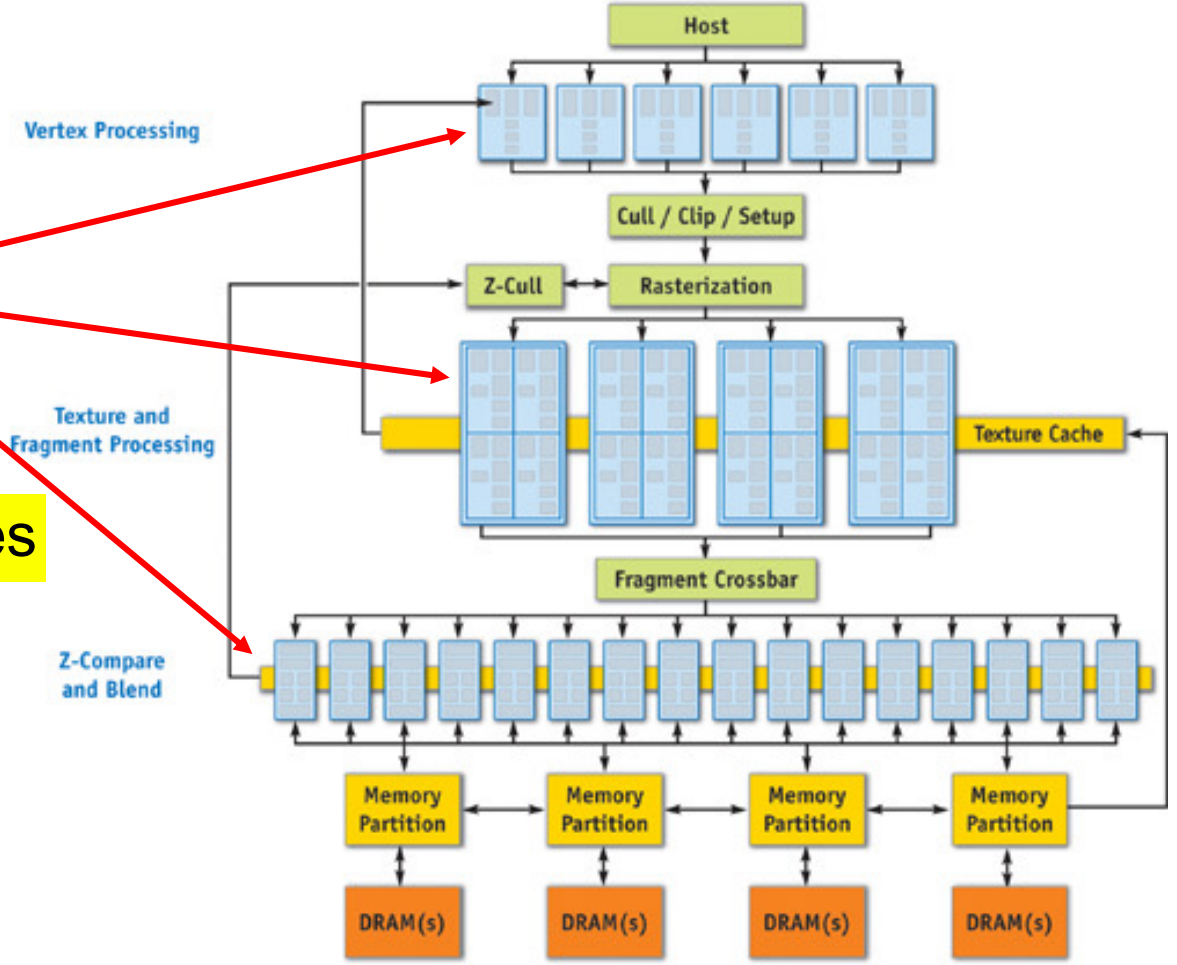
- OpenGL/Direct3D workflow:
  - Application provides shader program binaries to the GPU
  - Application sets graphics pipeline parameters
  - Application provides to the GPU a buffer of vertices
  - Application send to the GPU a drawing command



NVIDIA GeForce 6 (2004)

- Pipelined architecture
- Multiple cores for each stage
- Programmable stages

- The introduction of programmable stages requires fetch and decode units

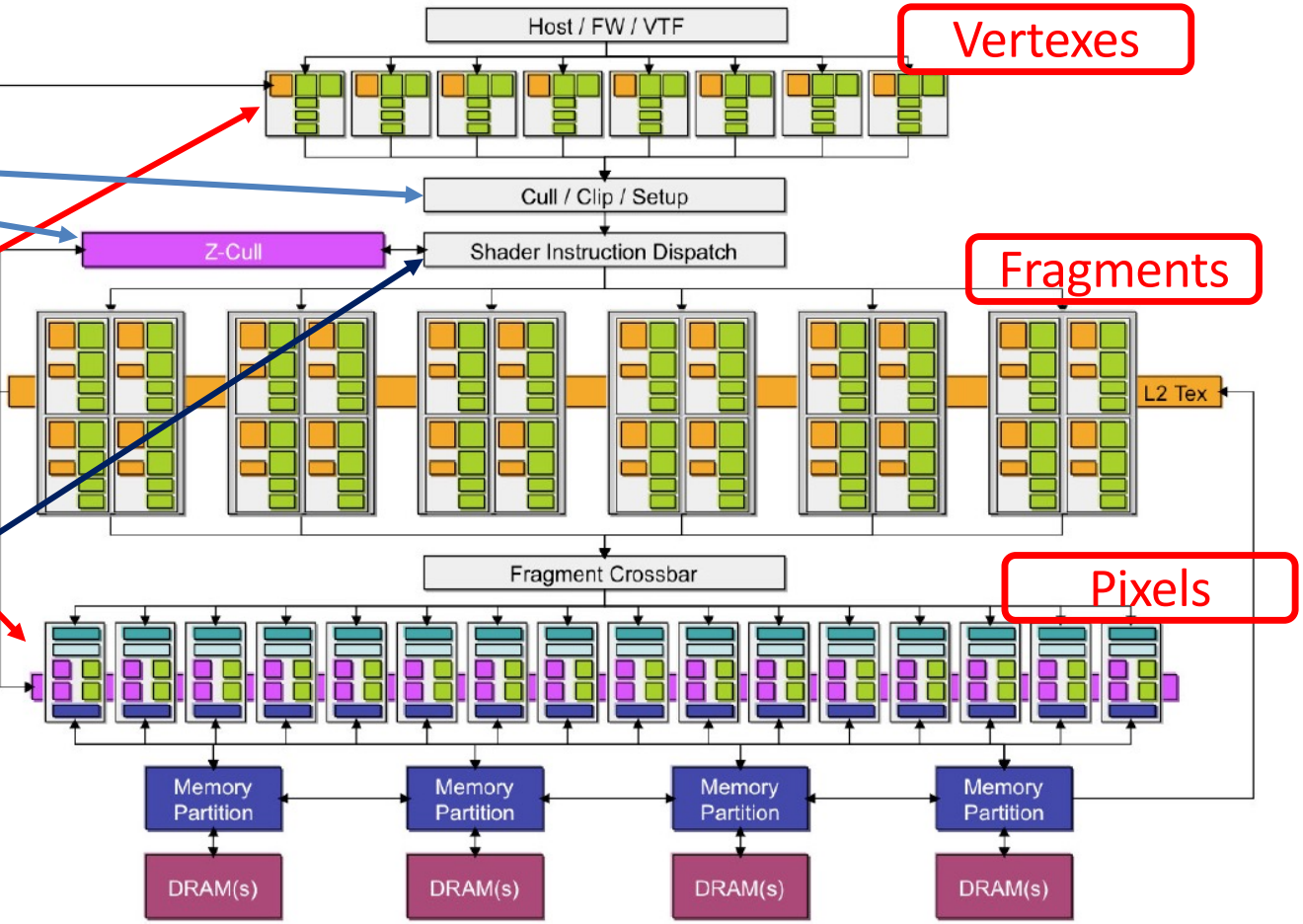


NVIDIA GeForce 7800 (2005)

Fixed stages

Programmable stages

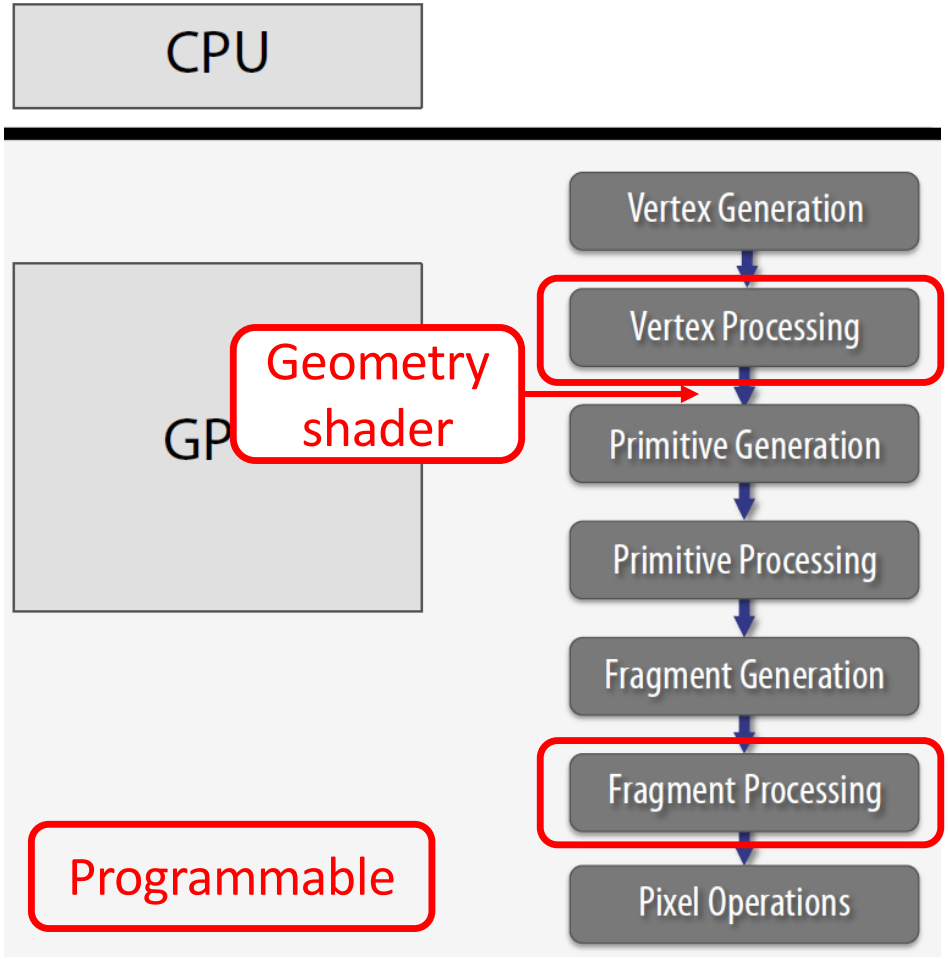
The introduction of programmable stages requires fetch and decode units



NVIDIA GeForce 8 (2006)

- Ground-up architecture redesign
- New geometry shader after the vertex shader
- Introduction of the unified shader processor

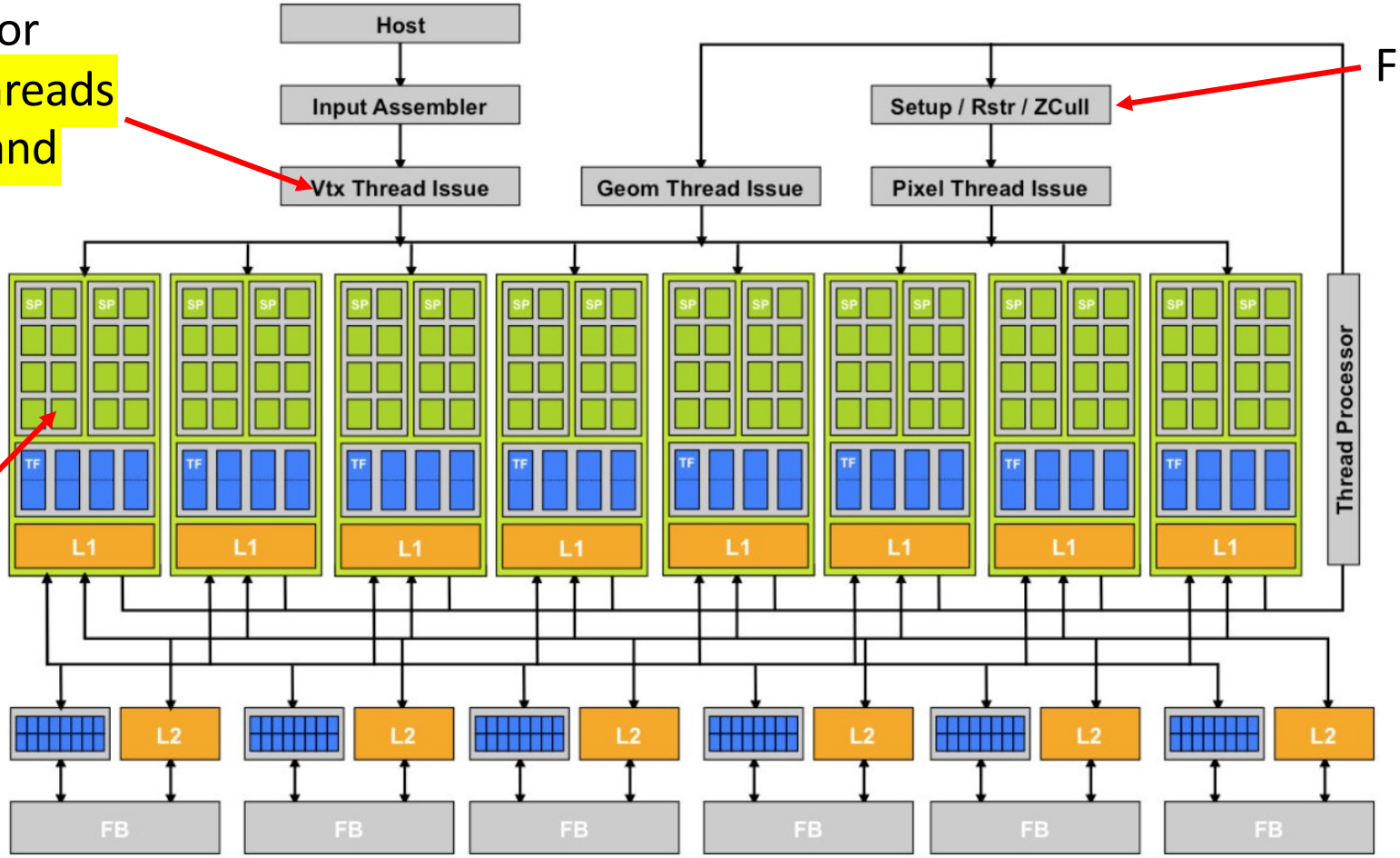
- Introduction of CUDA
  - Employment of GPU for general-purpose computing on GPU (GPGPU)



NVIDIA GeForce 8800 - Tesla (2006)

Issue Units for managing threads generation and scheduling

Grid of unified shader processors



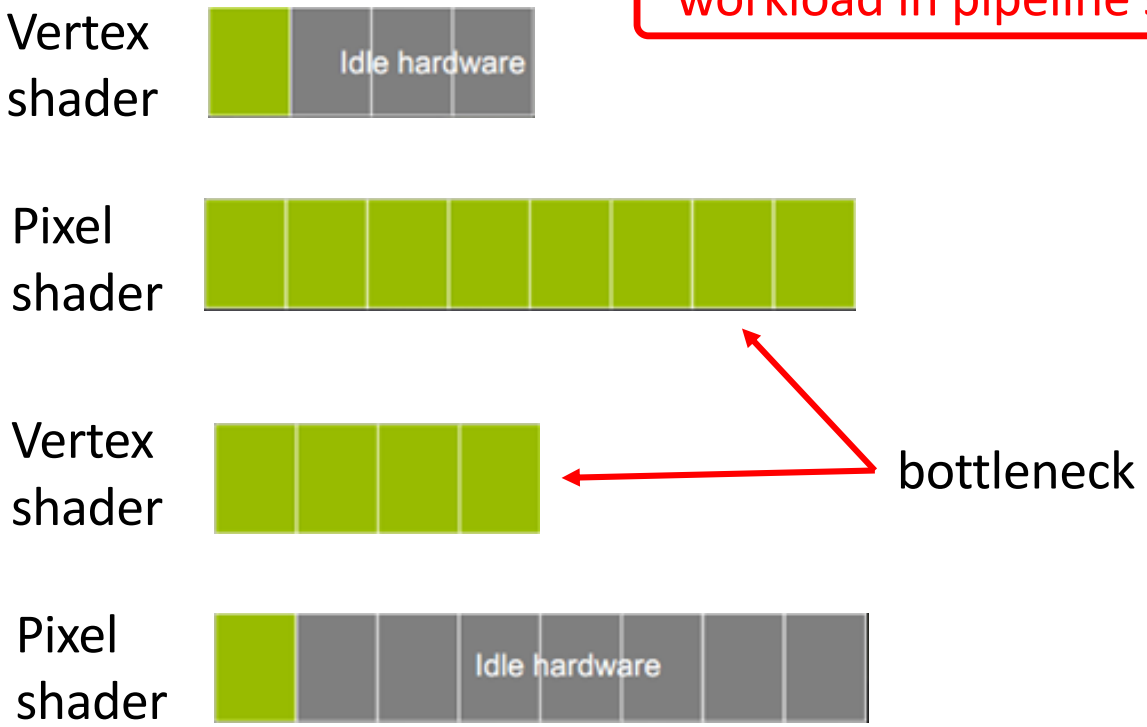
Why a single shader processor?

- Non-unified shader processors

Problems in balancing workload in pipeline stages

Heavy pixel workload

Heavy geometry workload



Why a single shader processor?

- Unified shader processors

Optimal usage of processing resources

Heavy pixel workload

Unified shader

Heavy geometry workload

Unified shader

Unified shader processor

- How does the unified shader processor work?

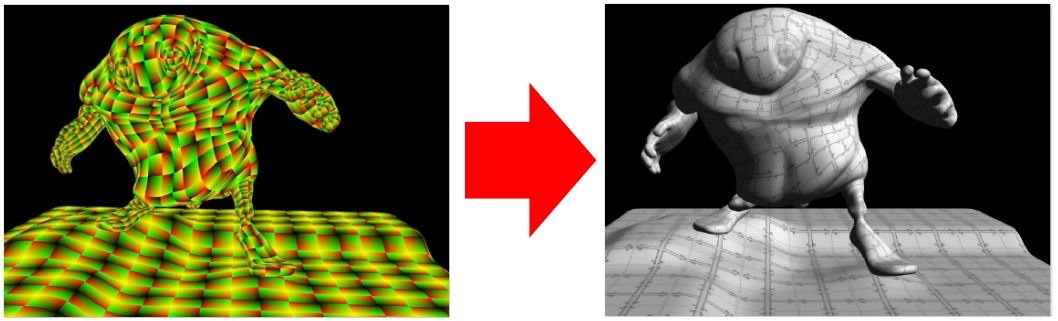
- Three key ideas:
  1. Instantiate many shader processors
  2. Replicate ALU inside the shader processor to enable SIMD processing
  3. Interleave the execution of many groups of SIMD threads

An example: a diffuse reflectance shader

- Shader programming model:
  - Fragments (or vertex or pixels) are processed independently
  - The function is written to work on a single fragment

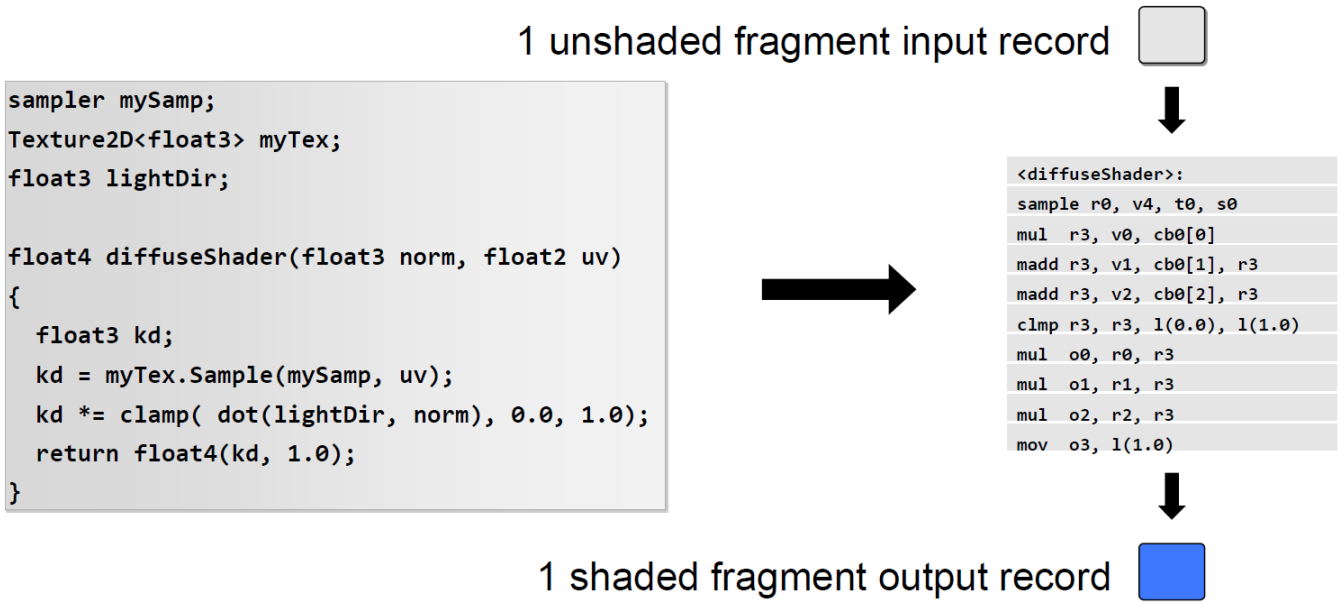
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

OpenGL code



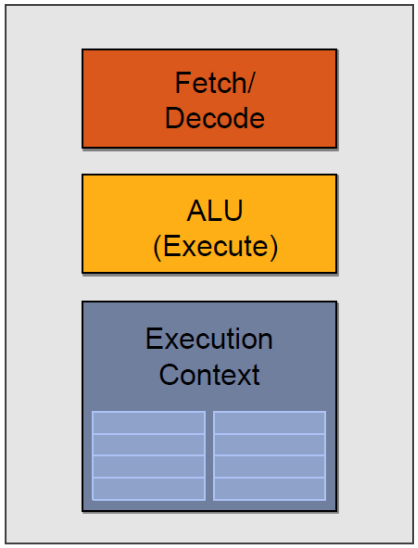
An example: a diffuse reflectance shader

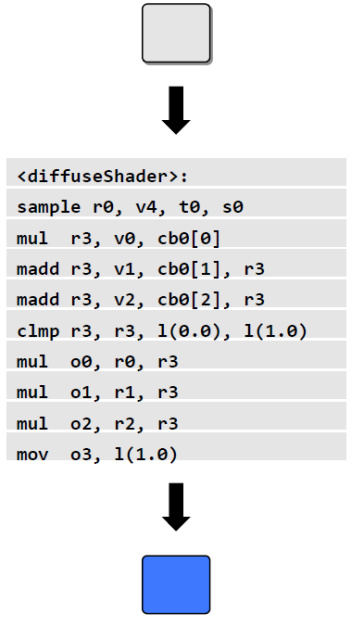
- Shader programming model:
  - Fragments (or vertex or pixels) are processed independently
  - The function is written to work on a single fragment



Unified shader processor

- Shader programming model: fragments (vertices, pixels, …) are processed independently
  - The function has to be executed for each fragment

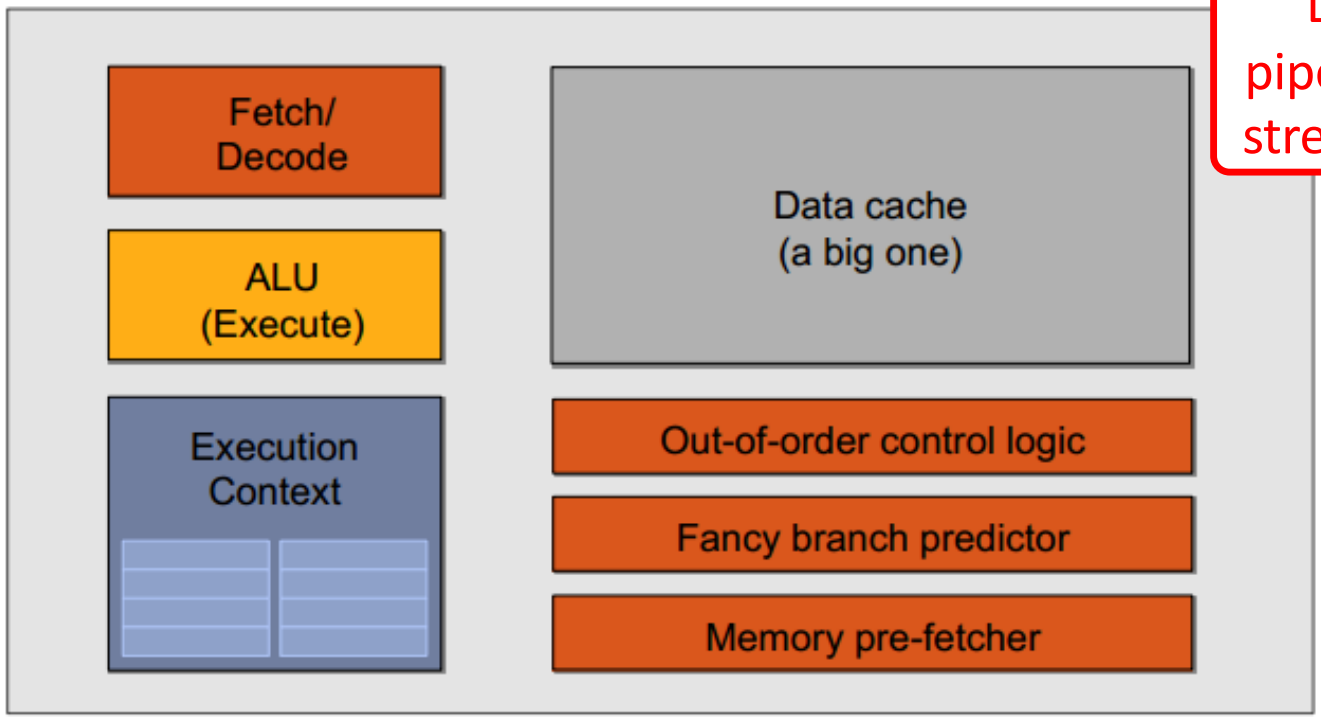




One instruction stream per fragment

Basic architecture of a modern CPU

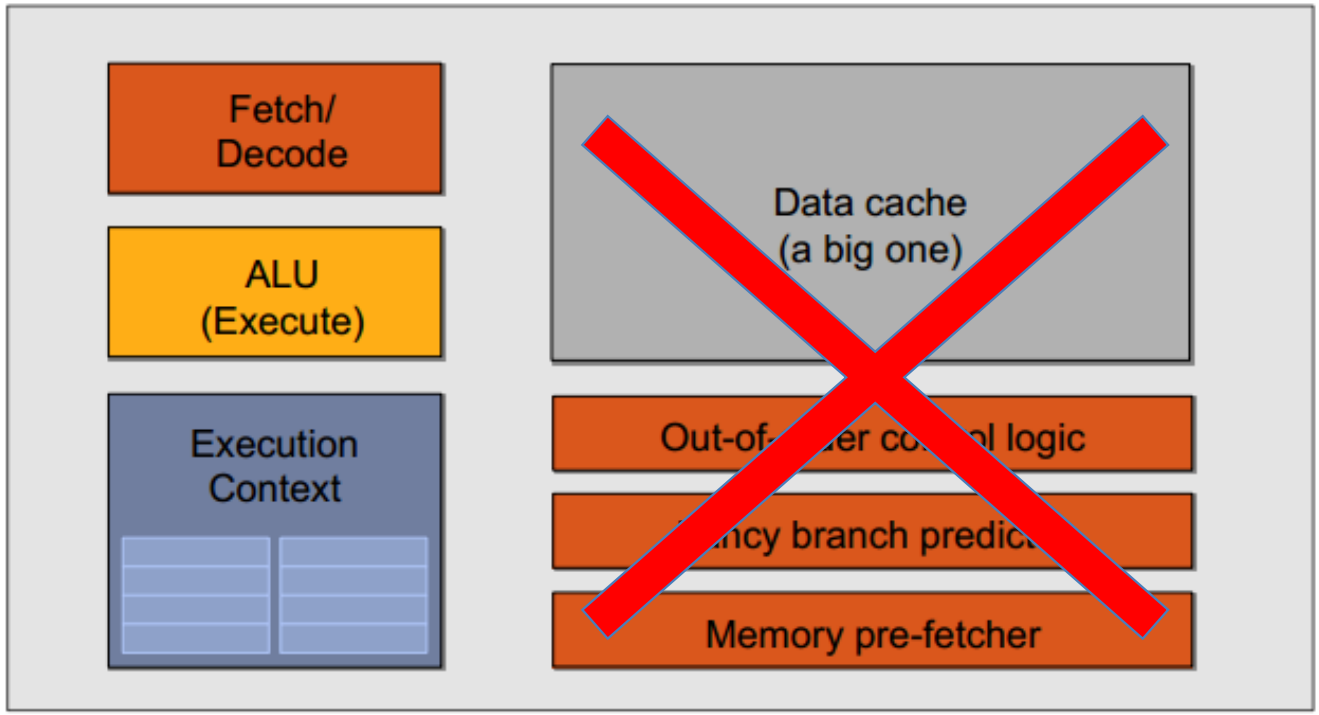
- The majority of the chip transistors are used to perform operations that help make a single instruction stream run faster



Does graphics pipeline instruction stream need them?

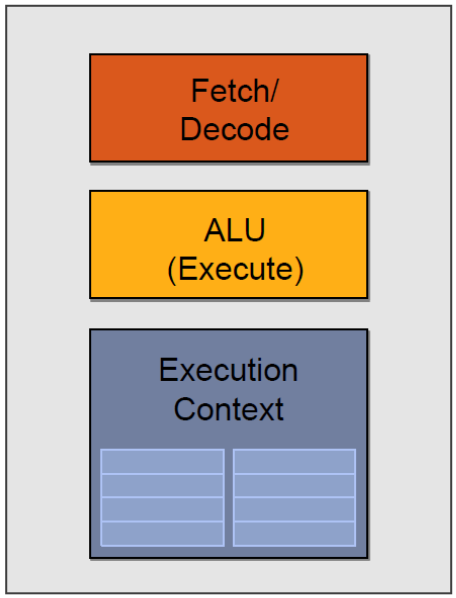
Basic architecture of a modern GPU

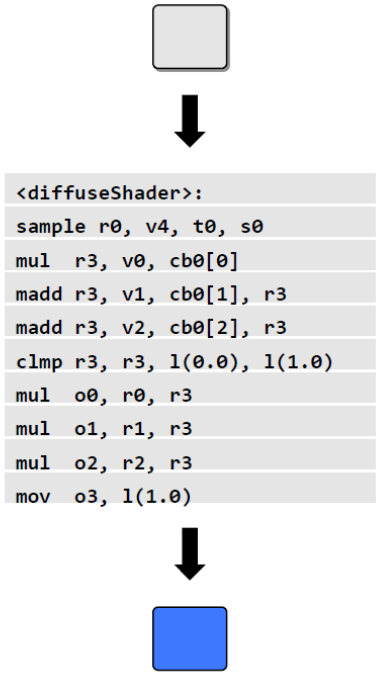
- Remove all components that help a single instruction stream run faster



Basic GPU core

- The result is a simple core executing an instruction stream

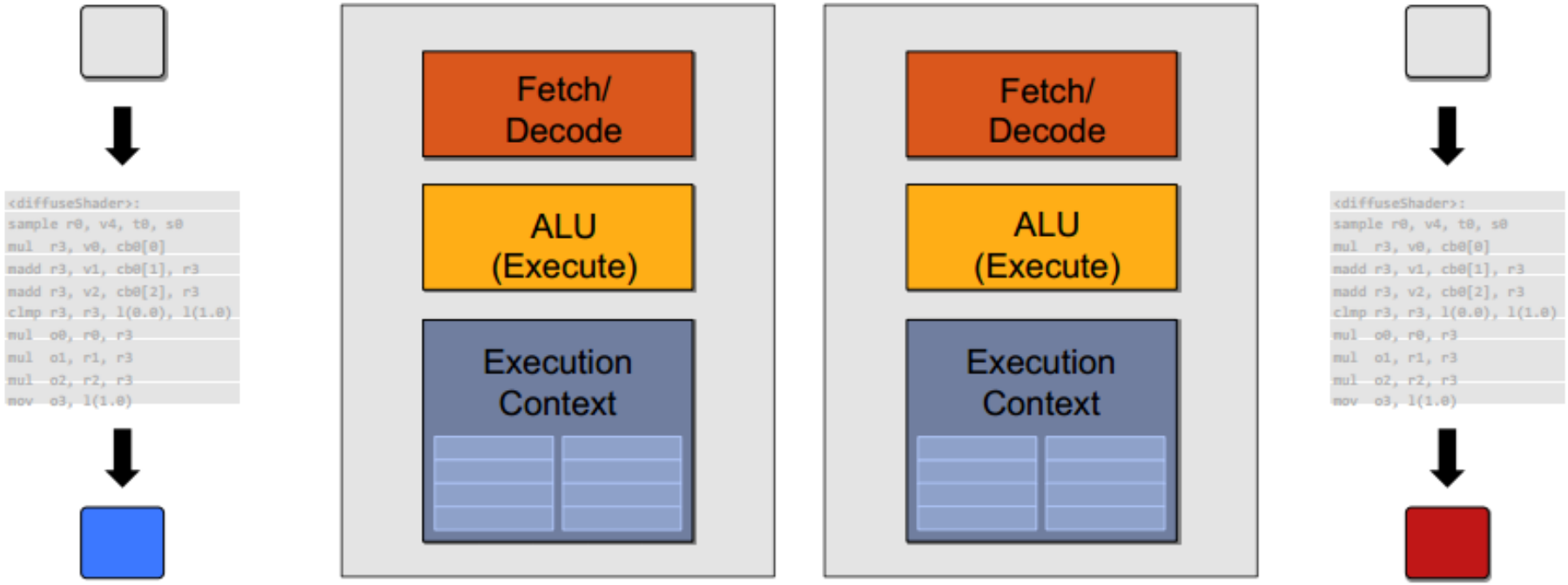




Replicate cores

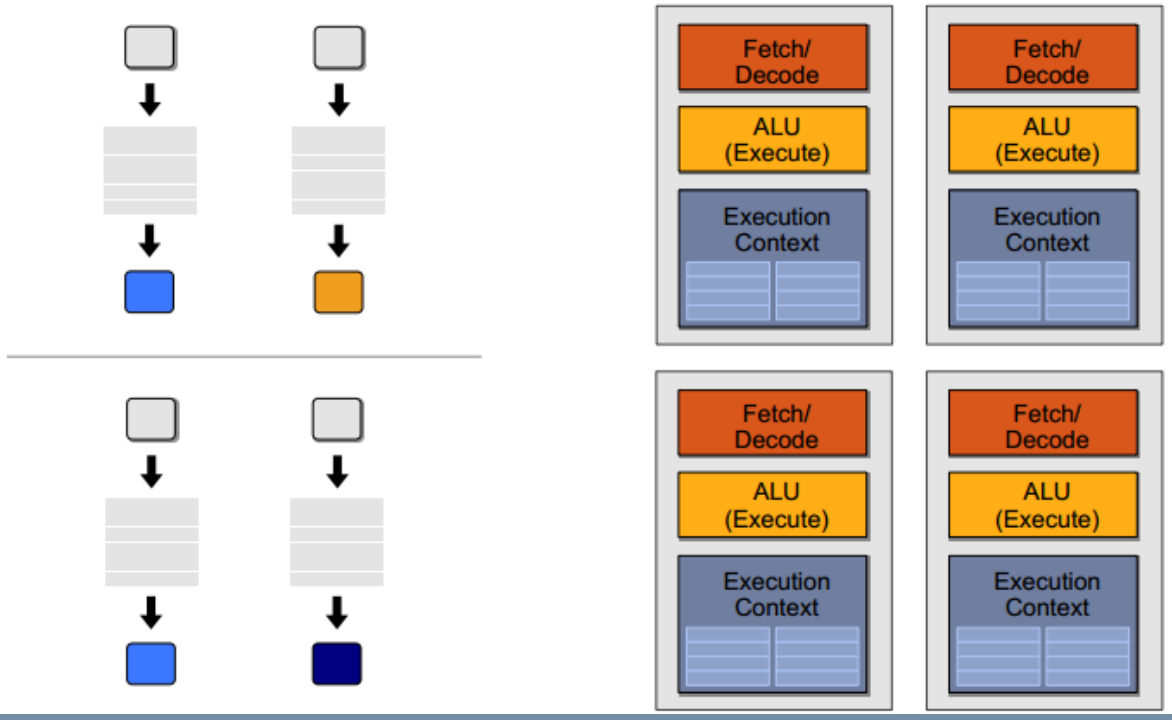
- Replicate cores to run several threads in parallel
  - 2 cores process 2 instruction streams in parallel

Two fragment processing threads are independent from each other



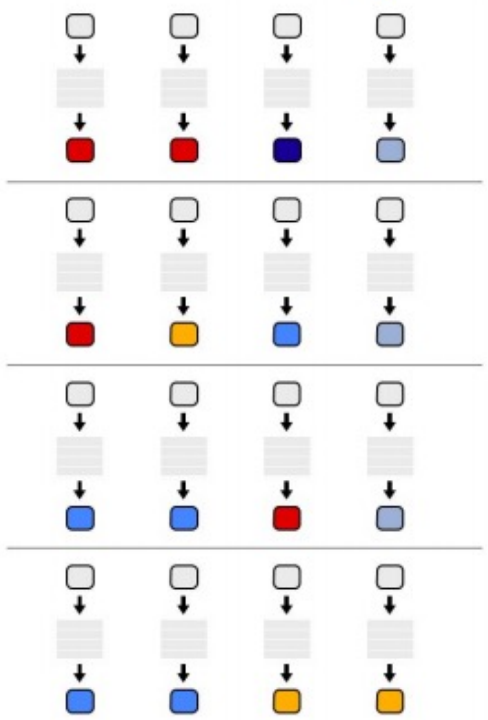
Replicate cores

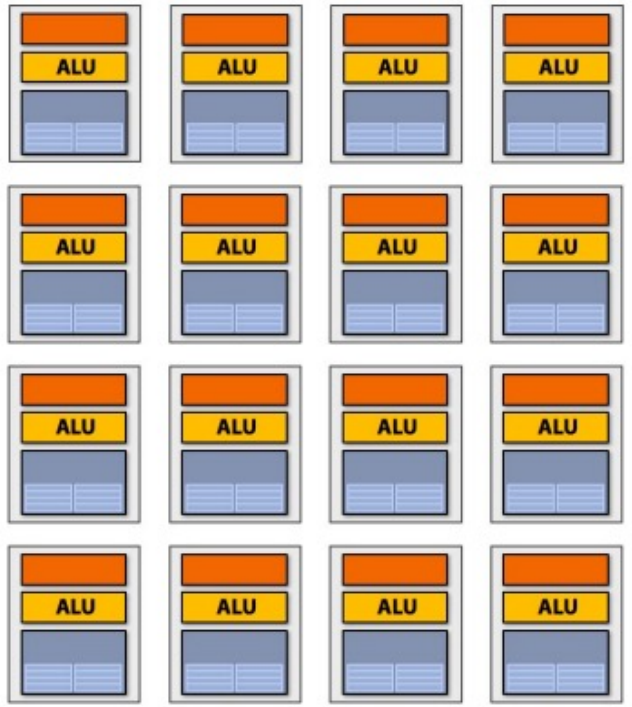
- Replicate cores to run several threads in parallel
  - 4 cores process 4 instruction streams in parallel



Replicate cores

- Replicate cores to run several threads in parallel
  - 16 cores process 16 instruction streams in parallel

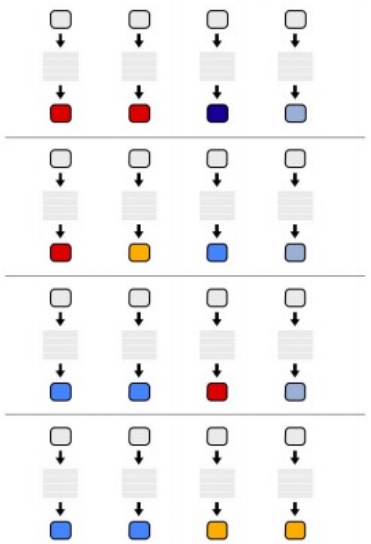


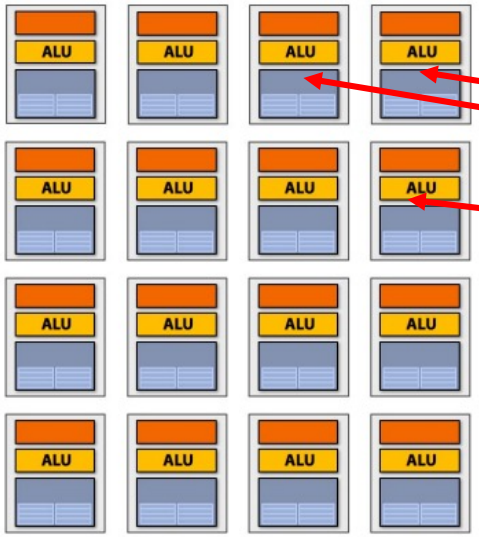


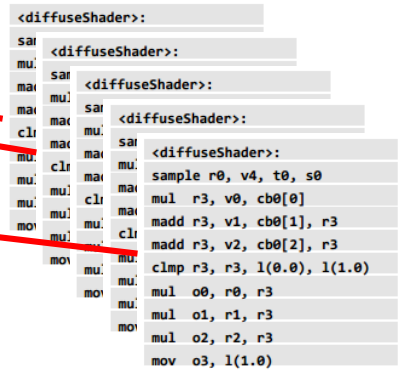
... and so on
and so forth...

Replicate cores

- ISSUE: since threads runs the the same instruction stream, should cores share it?
  - NO, since each unit has its own fetch and decode unit, it is simpler to run different instruction streams

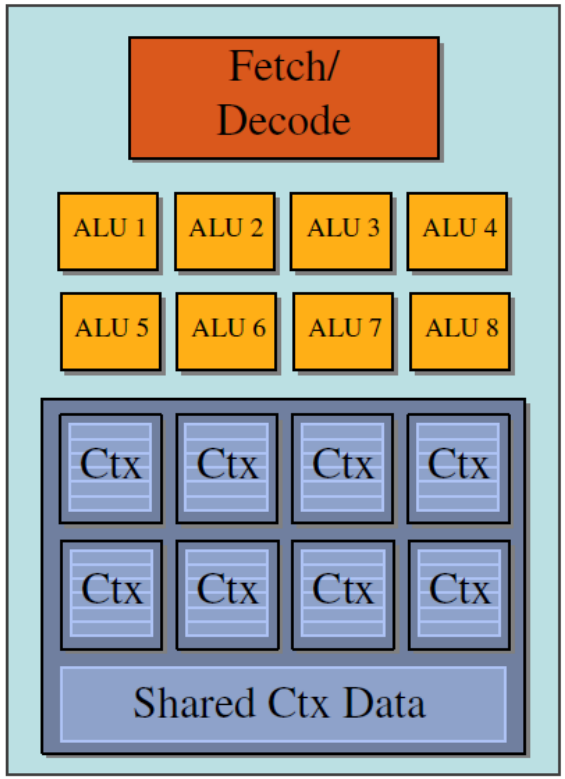






The result is a multicore architecture!

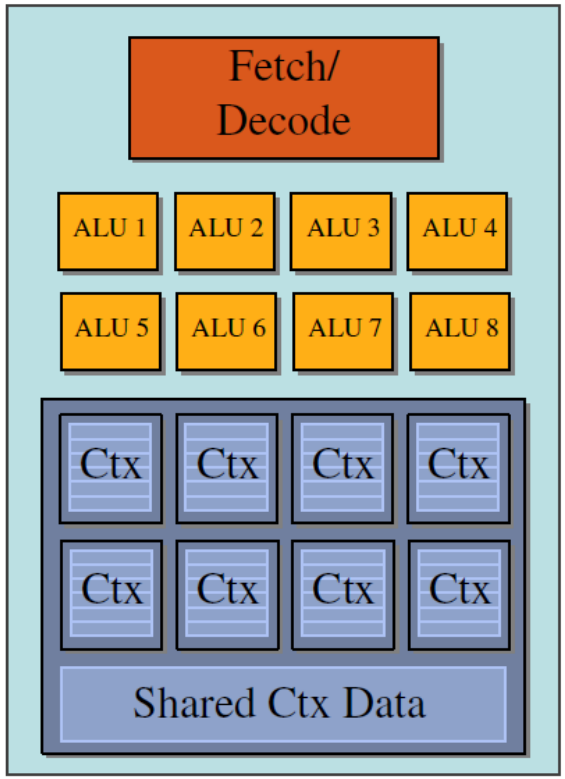
Replicate ALUs within the core



- Single fetch and decode unit
- Multiple ALUs
- Large register file storing several execution contexts

SIMD processing

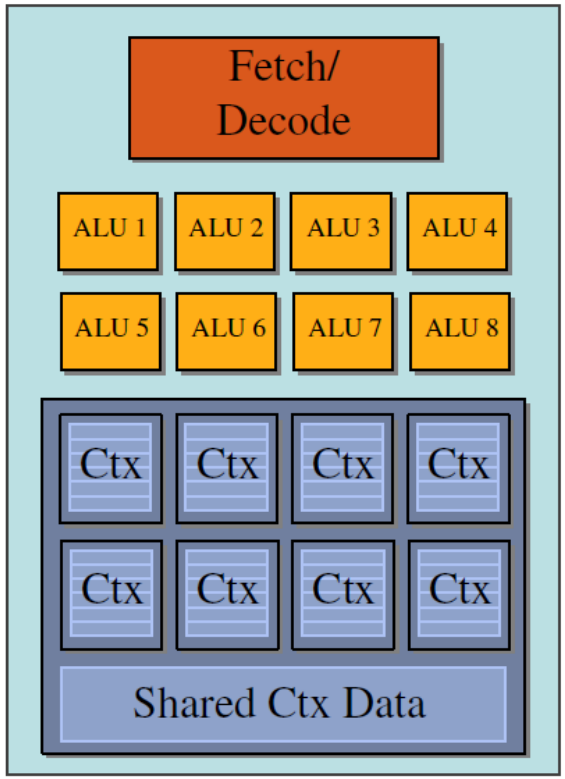
Replicate ALUs within the core



- …but the original compiled shader processes one item using scalar operations on scalar registers

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

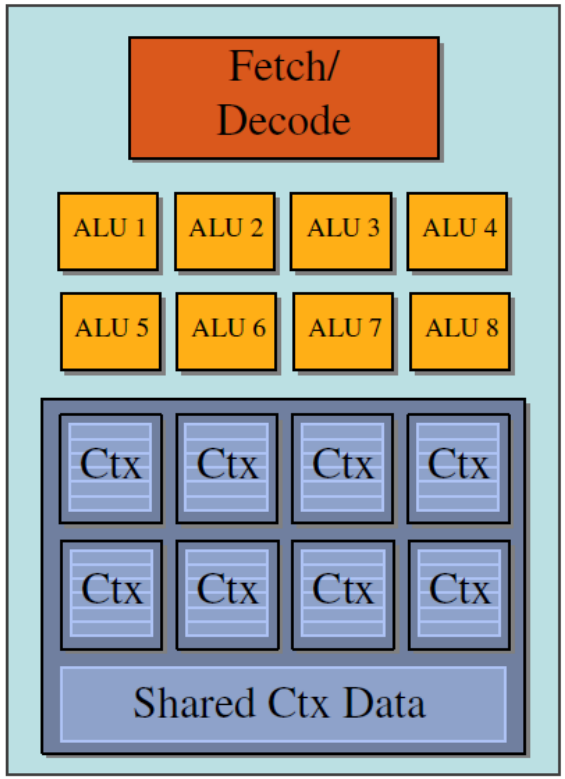
Replicate ALUs within the core



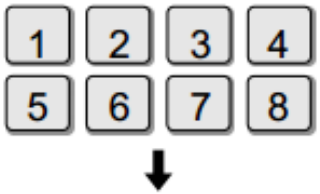
- First solution: the new compiled shader processes 8 items using vector operations on vector registers

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```

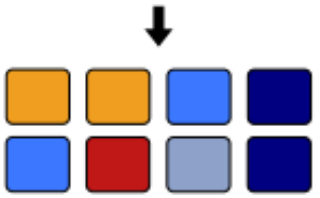
Replicate ALUs within the core



SIMD processing!




```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```



SIMD architecture based on vector instructions

- Vector instructions are generated by the compiler
  - Parallelism explicitly requested by programmer
  - Parallelism conveyed using parallel language semantics
  - Parallelism inferred by dependency analysis of loops

Explicit SIMD

- Examples:
  - Intel AVX2, Intel AVX512, ARM Neon instructions
  - Solutions generally used in CPUs

- SIMD parallelization is performed at compile time!

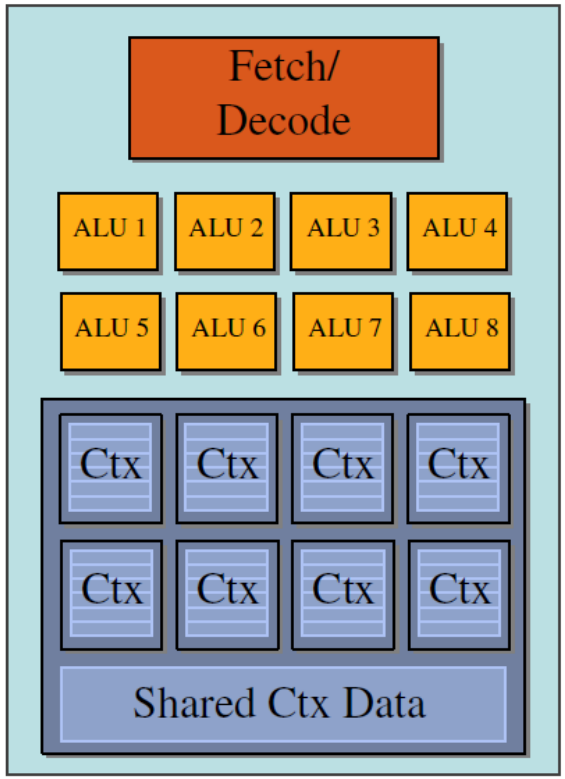
SIMT architecture

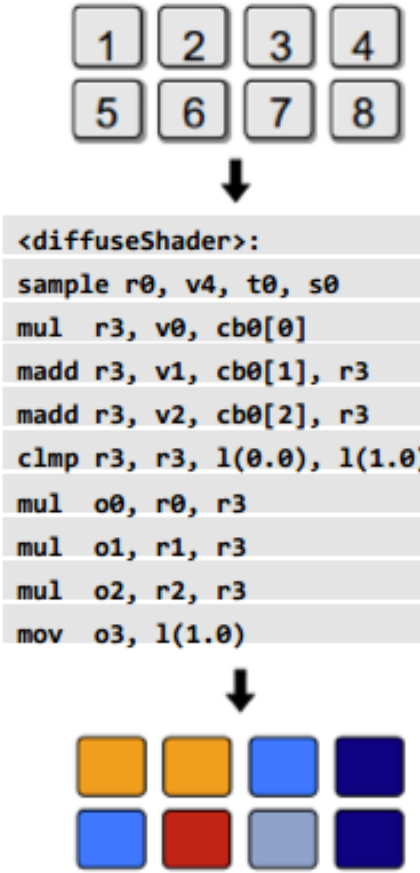
Implicit SIMD

- An alternative solutions is:
  - Compiler generates a binary with scalar instructions
  - N instances of the program are always run together on the SIMD core in lockstep
  - HW is responsible for simultaneous execution of the same instruction on different data of the N program instance on SIMD ALUs

- In NVIDIA terminology this is called **SIMT - single instruction multiple thread**

SIMT architecture

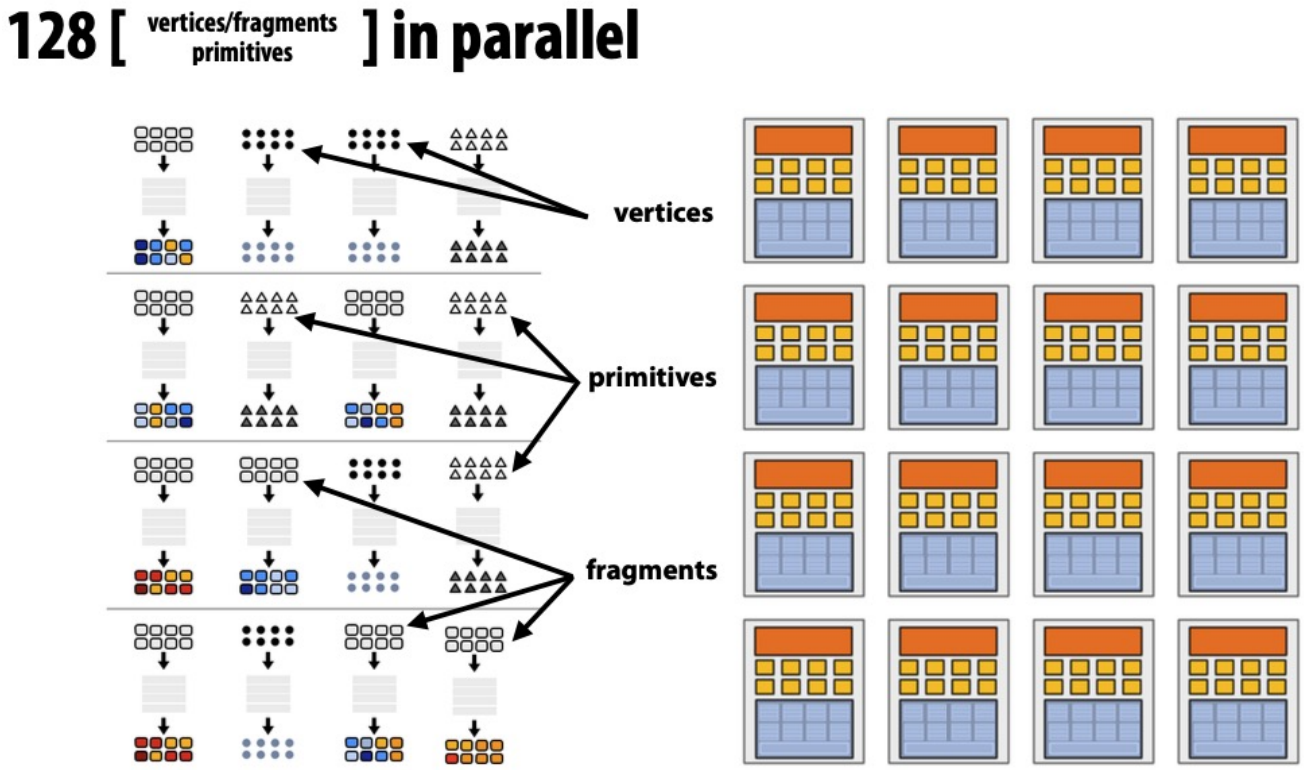




SIMT processing!

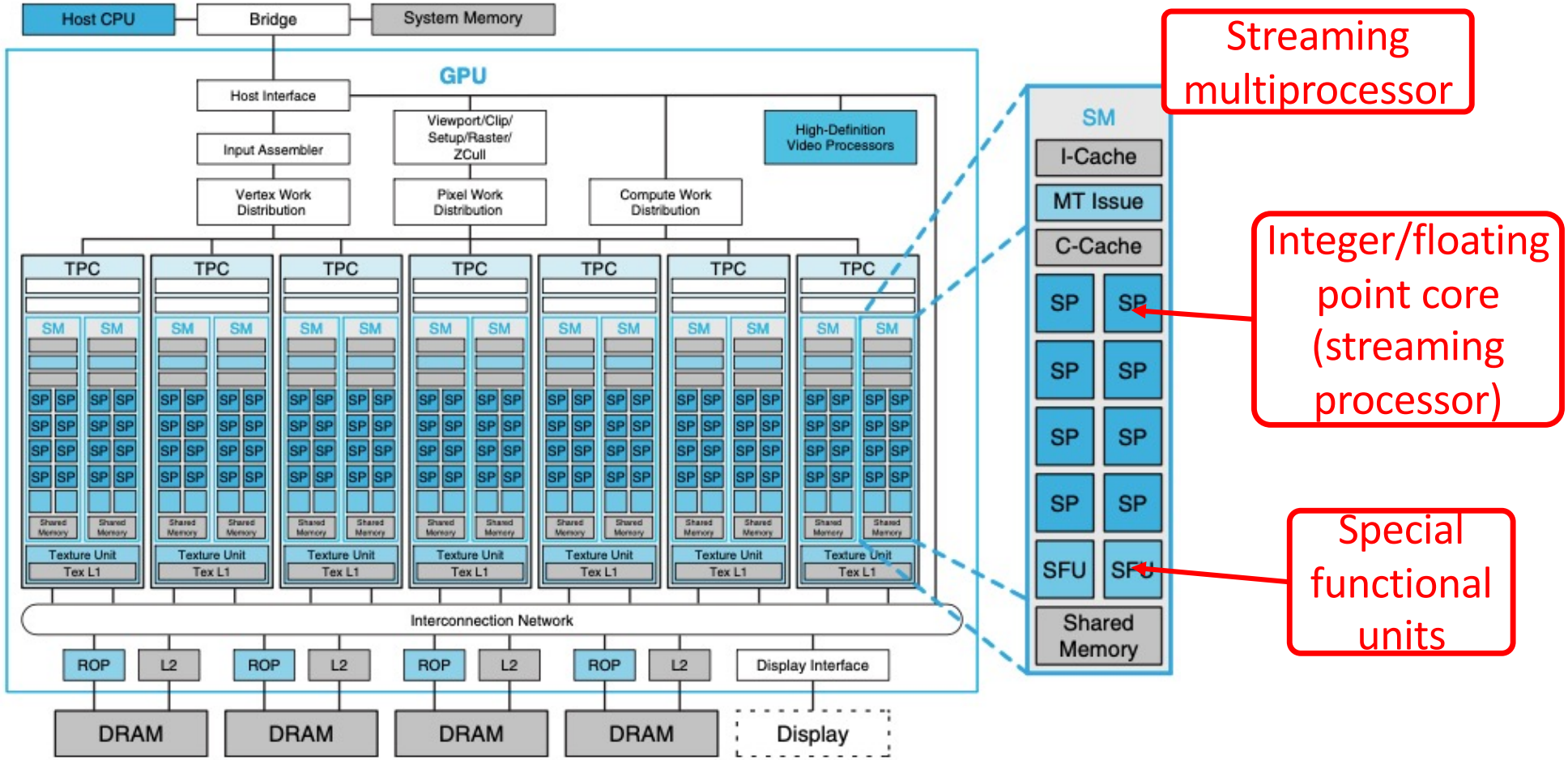
Merging two-level replications

- Result: multicore architecture where each core has a SIMD architecture

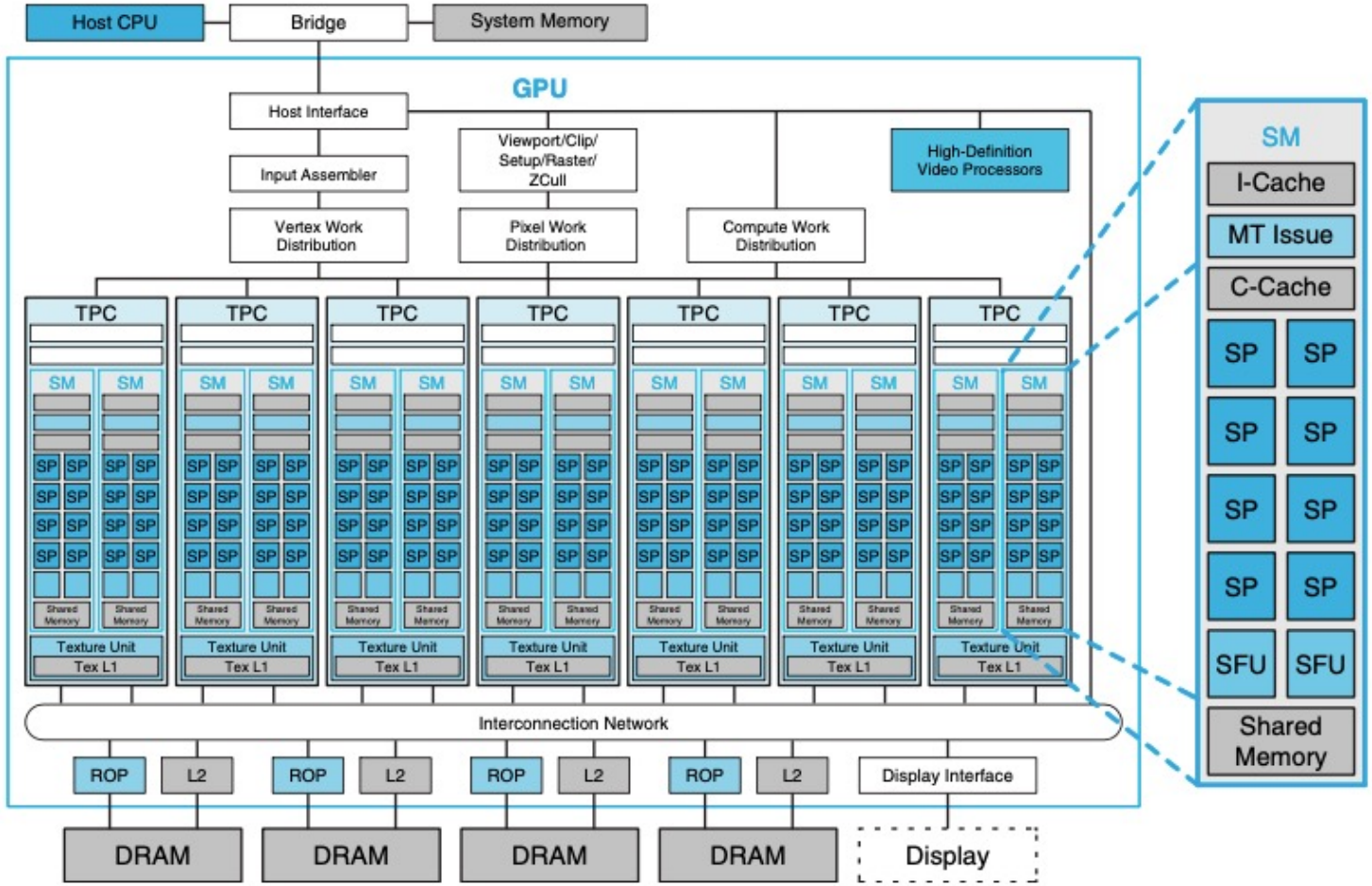


Final architecture is a manycore

A more detailed view on the Tesla architecture (2006)



General-purpose computing on GPU (GPGPU)



The final architecture is a manycore

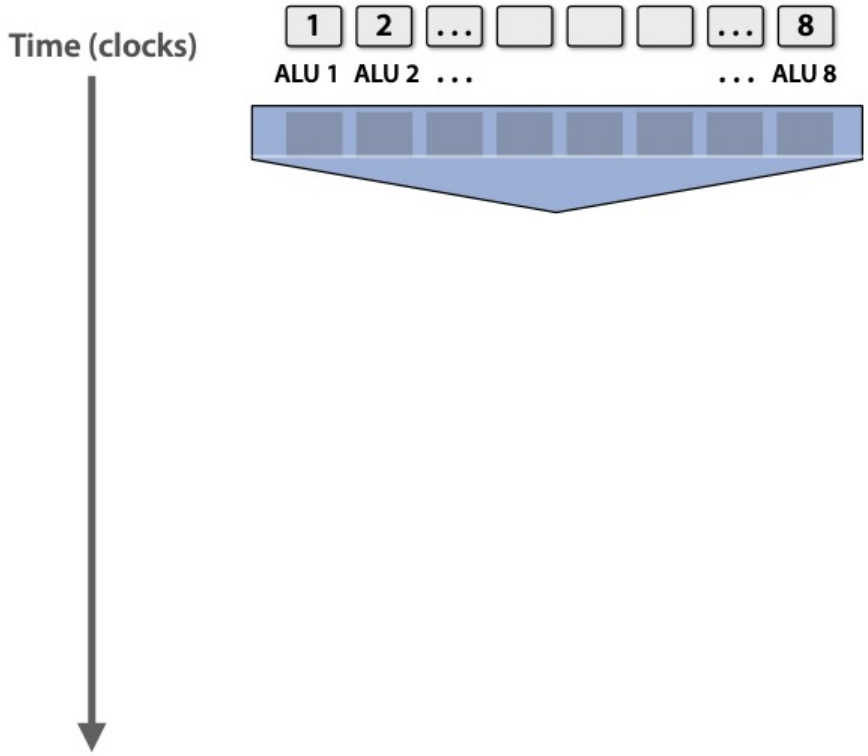
It can be used for general purpose computing

General-purpose computing on GPU (GPGPU)

- The GPU can be used to accelerate elaborations having similar structures/characteristics of the 3D rendering (large data parallelism, …)
- CUDA language (extension of C) can be used to code functions to be accelerate on the GPU
- CUDA program workflow:
  - Application (running on the CPU) sends data to the GPU memory
  - Application sends the binary of the **kernel function** to the GPU
  - Application tells the GPU to run the kernel function
  - Application copies back results from the GPU

Branches

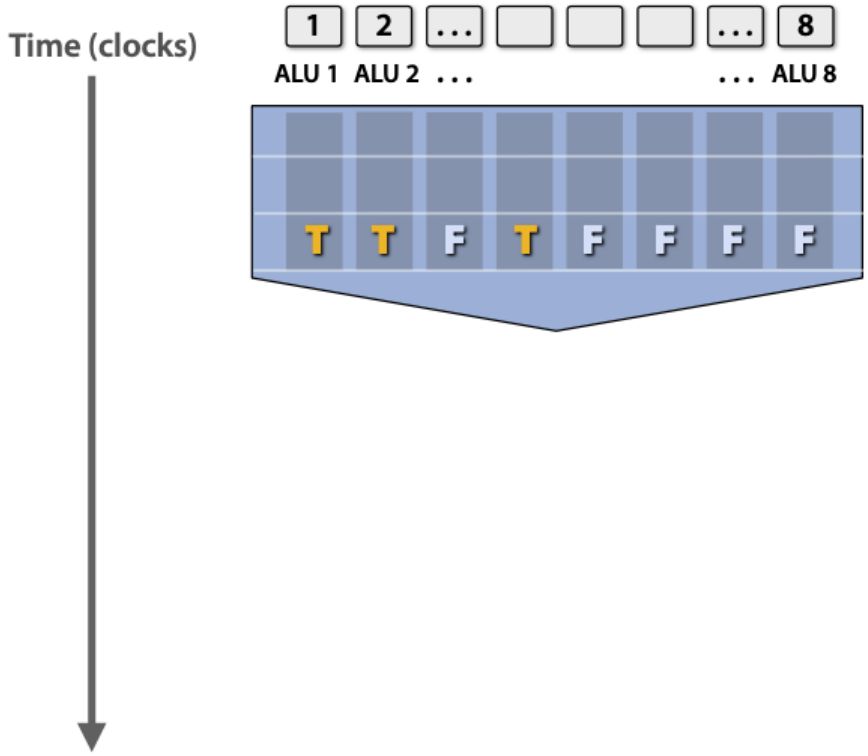
- What if there is a branch in a shader program?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Branches

- What if there is a branch in a shader program?

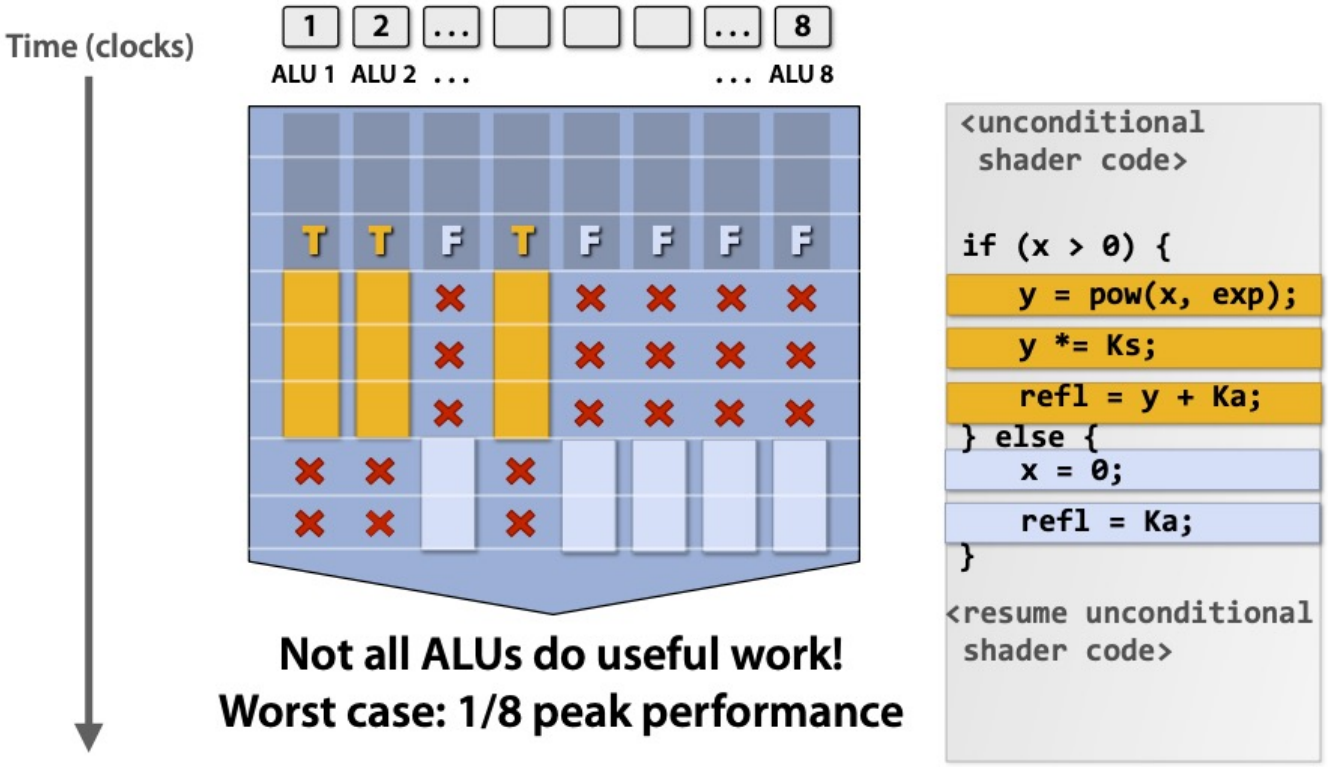



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Branches

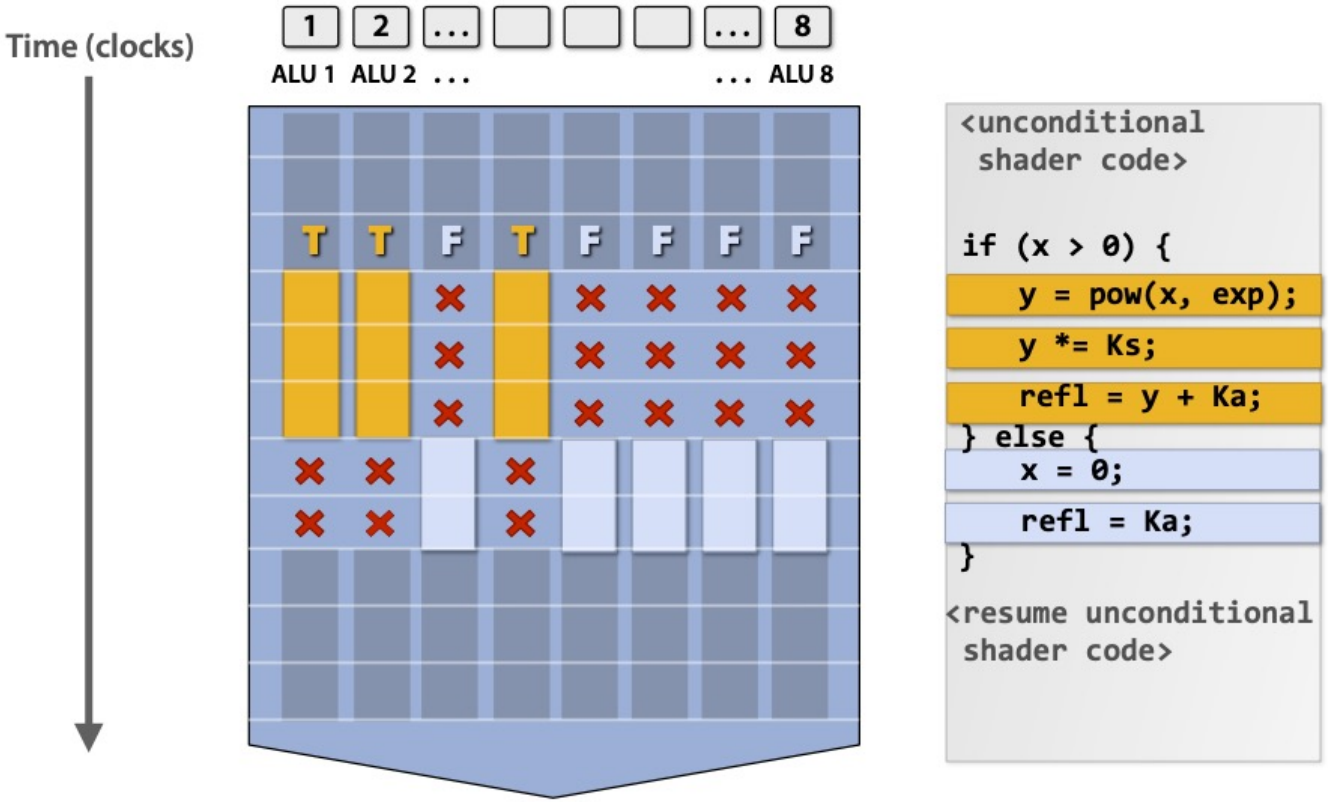
- What if there is a branch in a shader program?

Branch divergence!



Branches

- What if there is a branch in a shader program?



Coherent vs. divergent execution

- Instruction stream coherence ("**coherent execution**"): when the same instruction sequence applies to many data elements
  - Coherent execution **IS NECESSARY for SIMD** processing resources to be used efficiently
  - Coherent execution **IS NOT NECESSARY** for efficient parallelization **across different cores**
    - Different cores independently executes different instructions streams

- **Divergent execution**: a lack of instruction stream coherence

Stalls

- The execution of an instruction may have a data dependency with a previous one (still running) causing a **stall**!

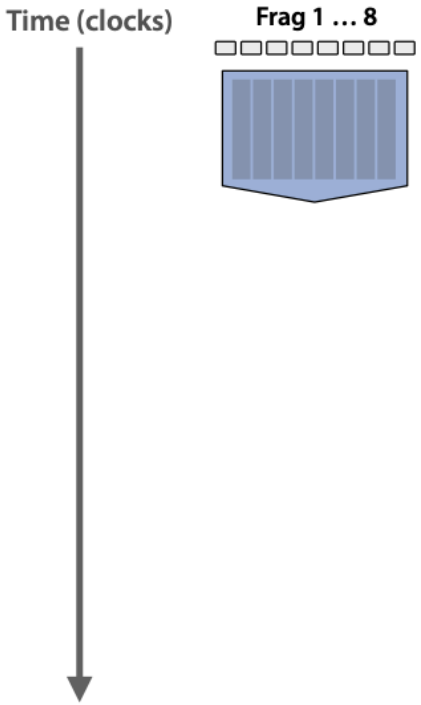
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Accessing memory is a major source of stalls - 100x slower than ALU instructions

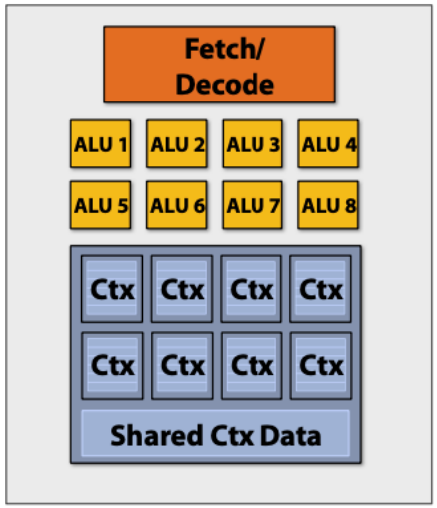
- Fancy caches and prefetching logic for avoiding stalls in CPUs have been removed in GPUs

Hiding stalls with instruction stream multi-threading

- Interleave processing of many streams on a single core to hide stalls caused by latency operations

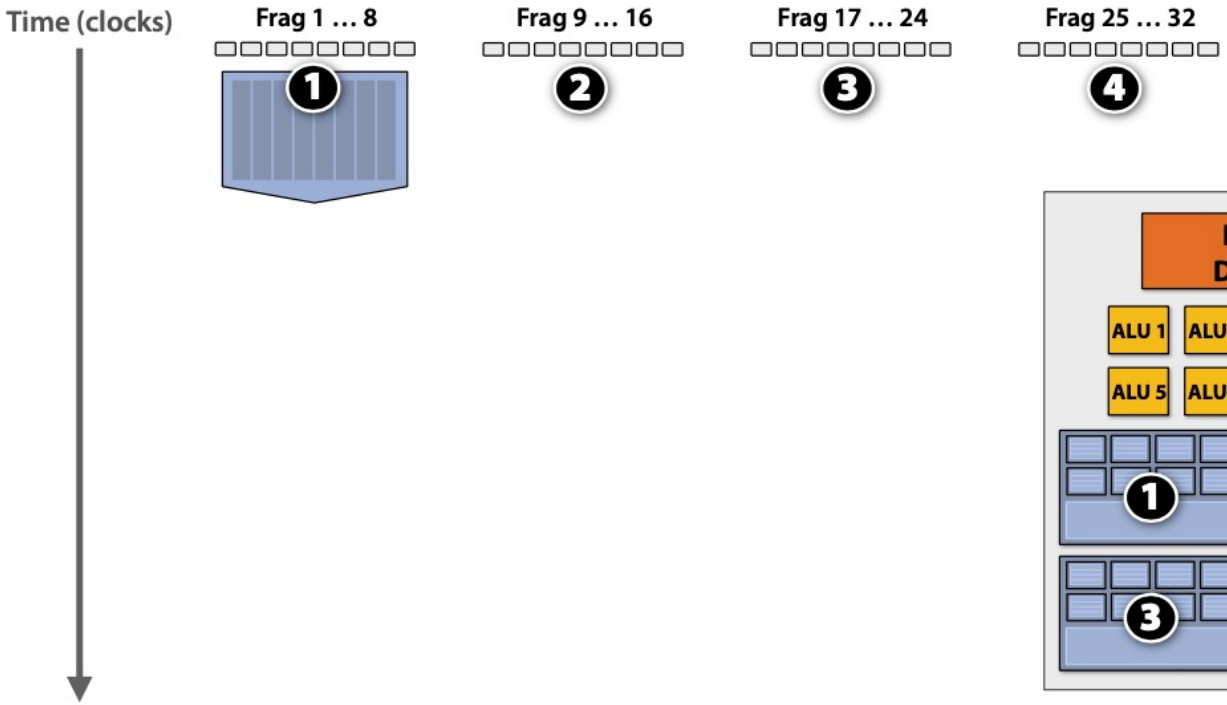


Former we had a single stream

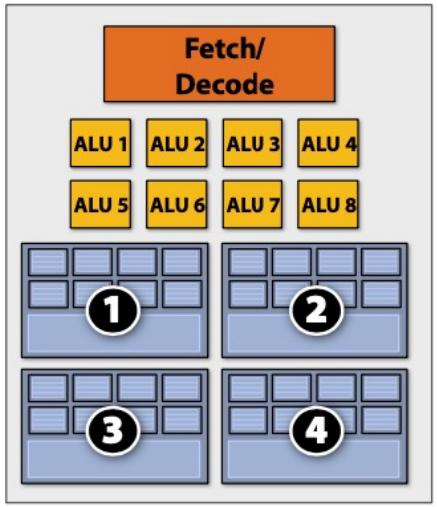


Hiding stalls with instruction stream multi-threading

- Interleave processing of many streams on a single core to hide stalls caused by latency operations

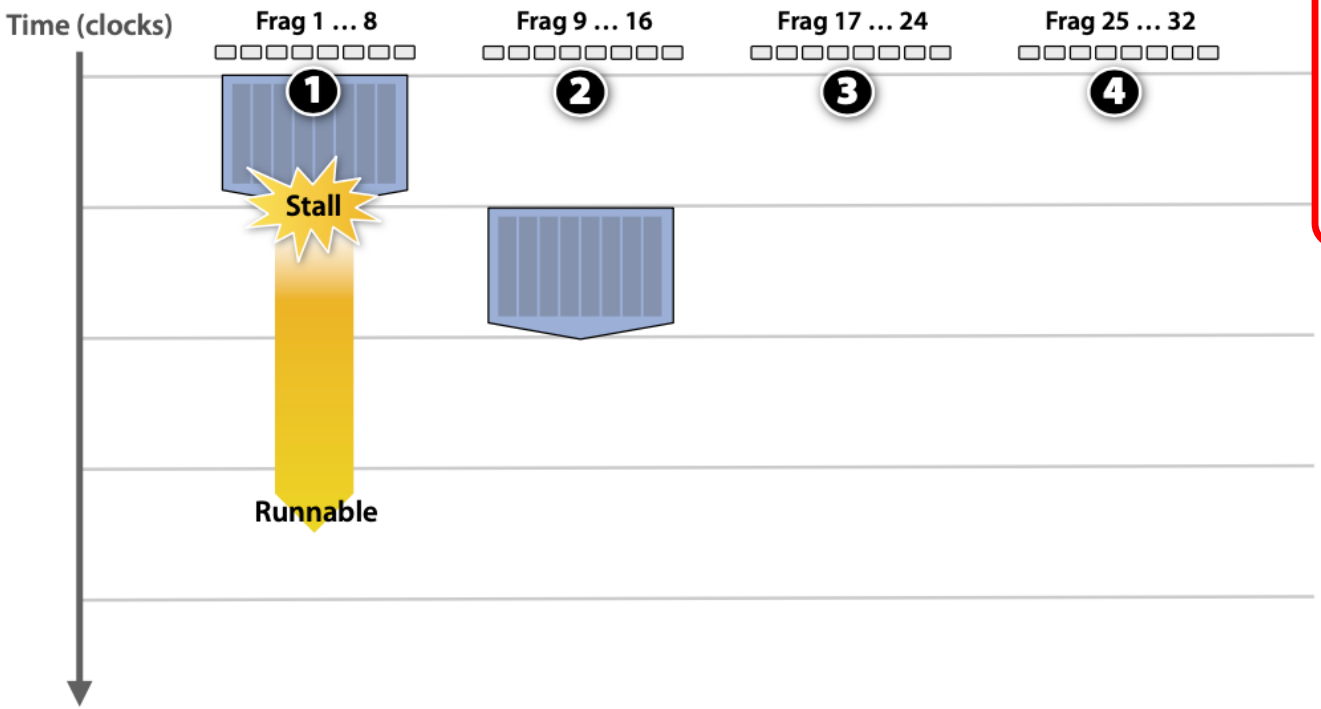


Interleave several instruction streams



Hiding stalls with instruction stream multi-threading

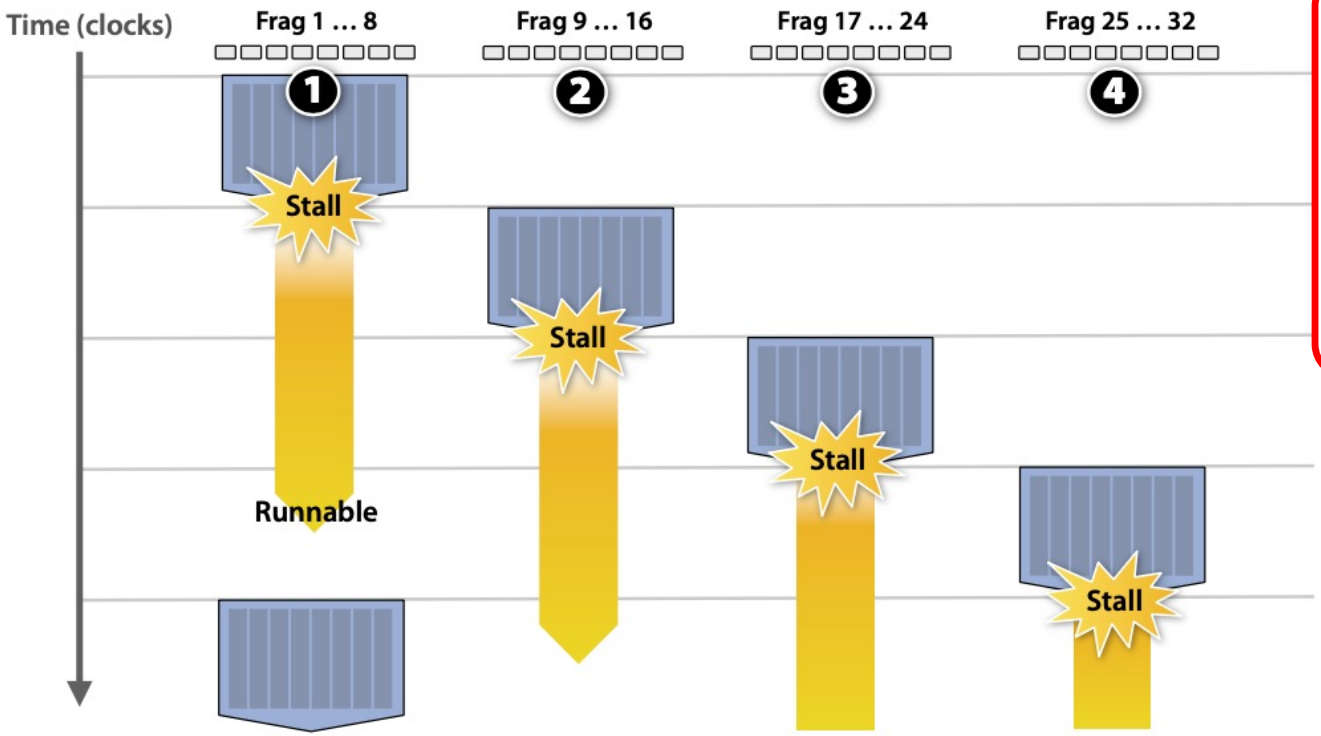
- Interleave processing of many streams on a single core to hide stalls caused by latency operations



Schedule another stream when the first one gets blocked

Hiding stalls with instruction stream multi-threading

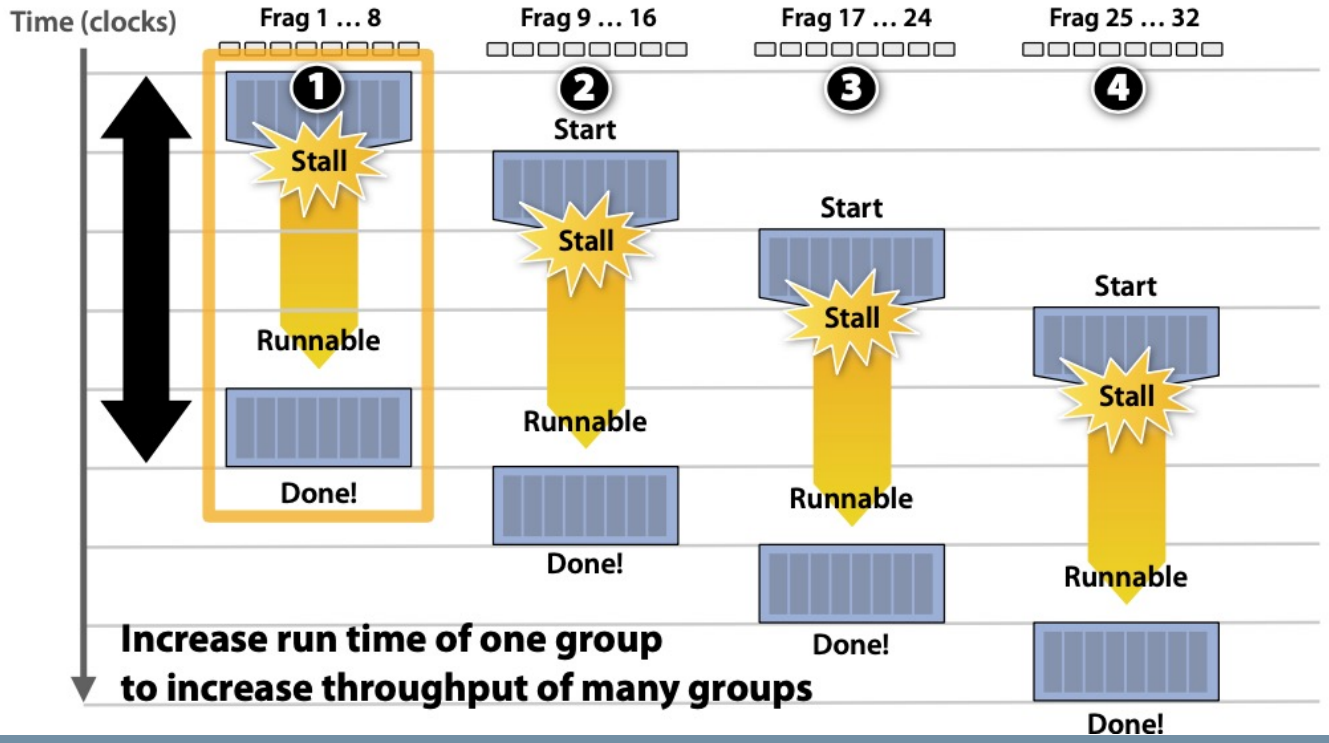
- Interleave processing of many streams on a single core to hide stalls caused by latency operations



Memory latency is hidden by executing arithmetic instructions of other streams

Hiding stalls with instruction stream multi-threading

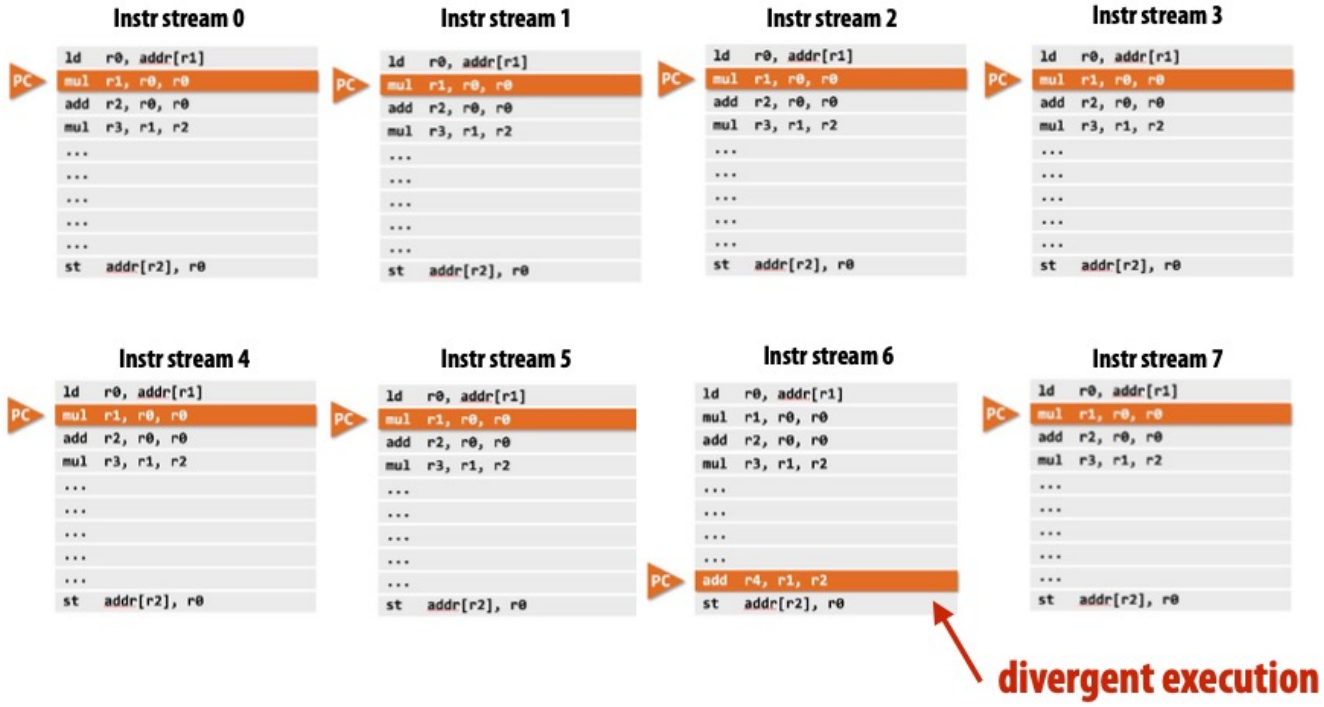
- Interleave processing of many streams on a single core to hide stalls caused by latency operations



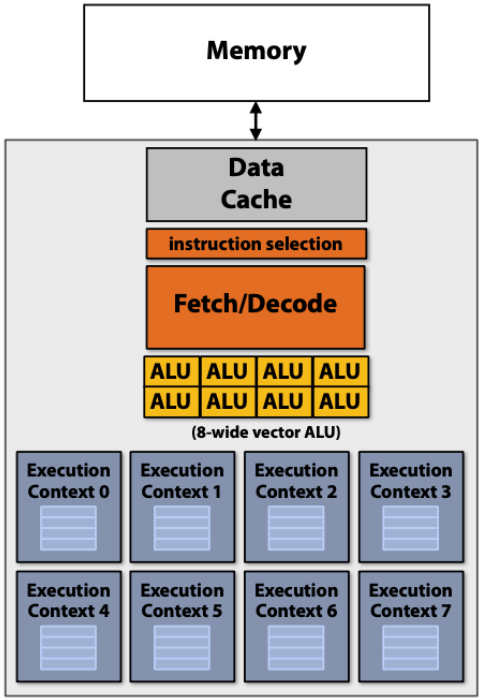
Interleaved multi-threading

Hiding stalls with instruction stream multi-threading

- Interleaved instruction streams are independent each other



Not only SIMT but also SPMD!



Throughput-oriented system

- We move from a latency-oriented architecture (i.e., the CPU) to a throughput-oriented one
  - Latency of the single stream is increased, but…
  - Overall throughput is increased
  - Consequently, the latency of the overall task execution is decreased!

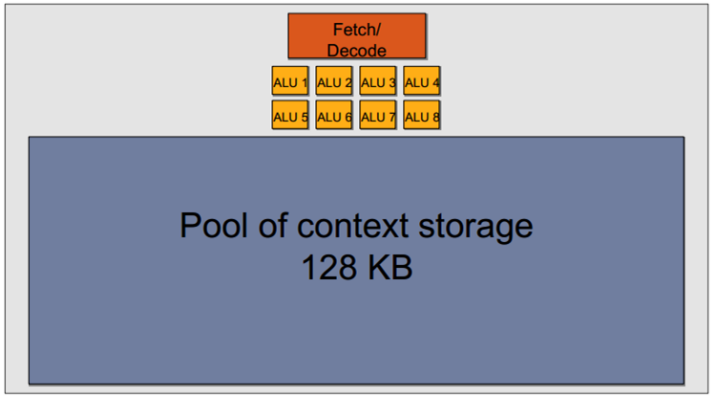
- Streams interleaving is managed by an HW scheduler and large register files to store execution contexts!

Takeaway

- The core is not stalled by memory accesses since running other **concurrent instruction streams**
  - The latency of the single instruction stream is worsened
  - The latency of memory operations are not positively affected by the proposed solution

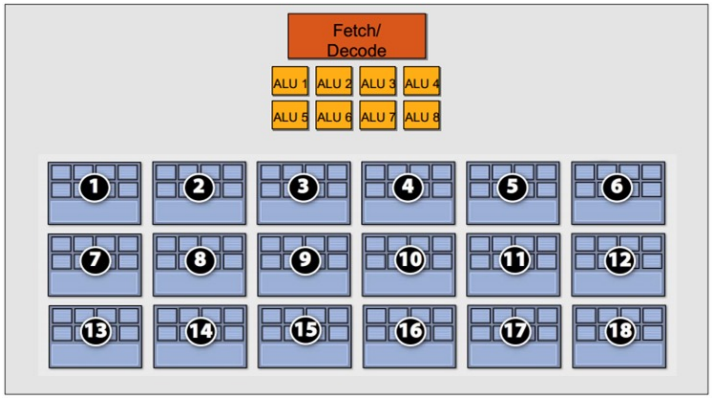
- The core is not stalled by running **arithmetic operations** of other instruction streams
  - How many concurrent instruction streams?
    - The larger the arithmetic/memory instruction ratio, the lower the number of streams to hide memory stalls

Context storage is another constraint

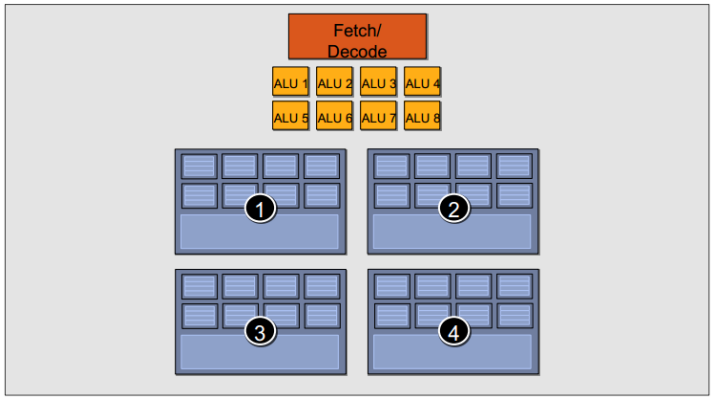


Resources for thread context are reserved for its entire execution (no swap memory!)

Threads require a small execution context ->
Many concurrent contexts ->
high latency hiding ability



Threads require a large execution context ->
Few concurrent contexts ->
low latency hiding ability

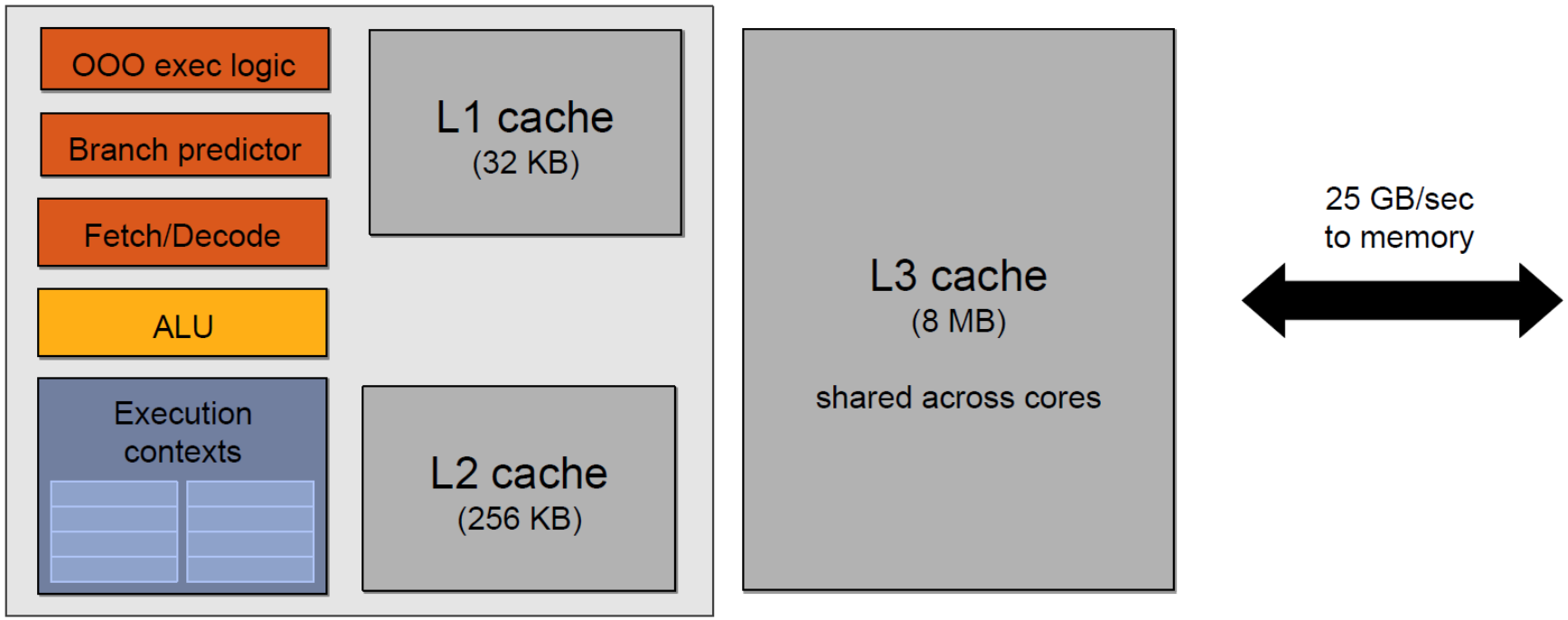


Basic architecture of a modern GPU - summary

- Use many “slimmed down” cores to run in parallel
  - Multicore of simplified processors
- Pack cores full of ALUs (by sharing instruction streams across group of data chucks)
  - Implicit SIMD execution managed by HW
- Avoid latency stalls by interleaving execution of many groups of instruction streams
  - When a group stalls, work on another group

CPU memory hierarchy

- CPU cores run efficiently when data is resident in cache
  - Caches reduce latency and proved high bandwidth



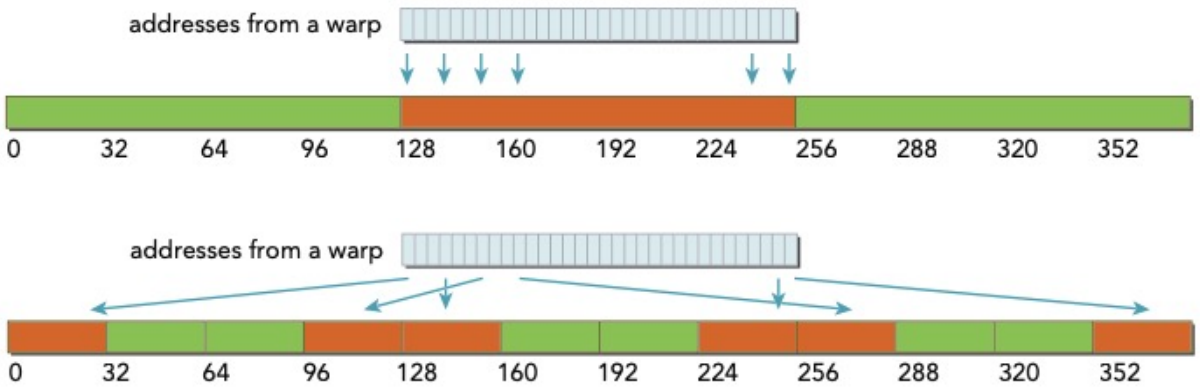
GPU memory hierarchy

- Initially GPU core was not provided with caches
- GPU has a high-bandwidth connection to memory
  - Memory is partitioned in banks each one with a separate port

GPU memory hierarchy

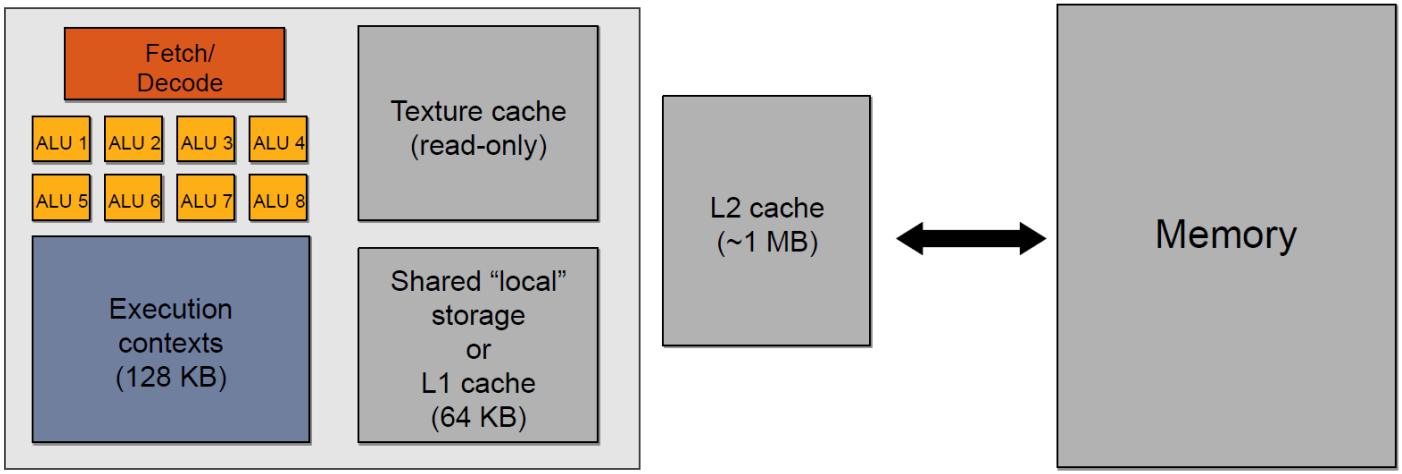
- GPU features sophisticated memory request order logic
  - Repack/reorder/interleave many buffered memory requests to maximize memory utilization

- Concurrent memory accesses are packed in transactions
  - Aligned memory accesses generates a single transaction
  - Non-aligned memory accesses generates multiple transactions



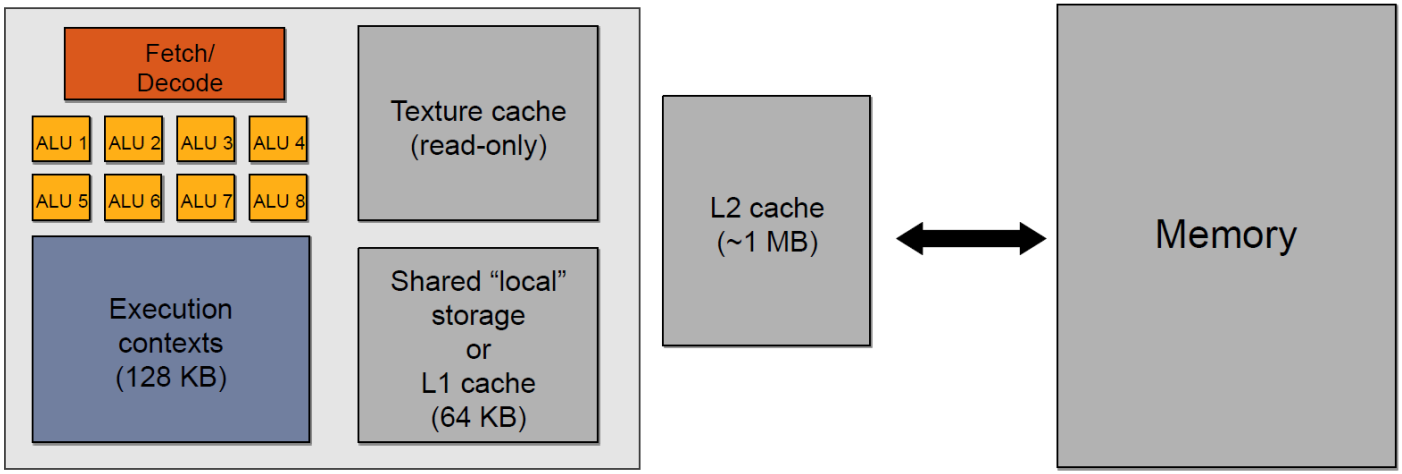
GPU memory hierarchy

- Modern GPUs are provided with
  - Local scratchpad memories (not synched with main memory)
  - Texture caches (read-only)



GPU memory hierarchy

- L1-L2 caches have been added (later in Fermi architecture…)
  - GPU applications present high spatial locality but very low temporal locality
    - Caches are smaller w.r.t. CPU ones to exploit spatial locality
  - No cache coherency (synchronization explicitly forced by the programmer)



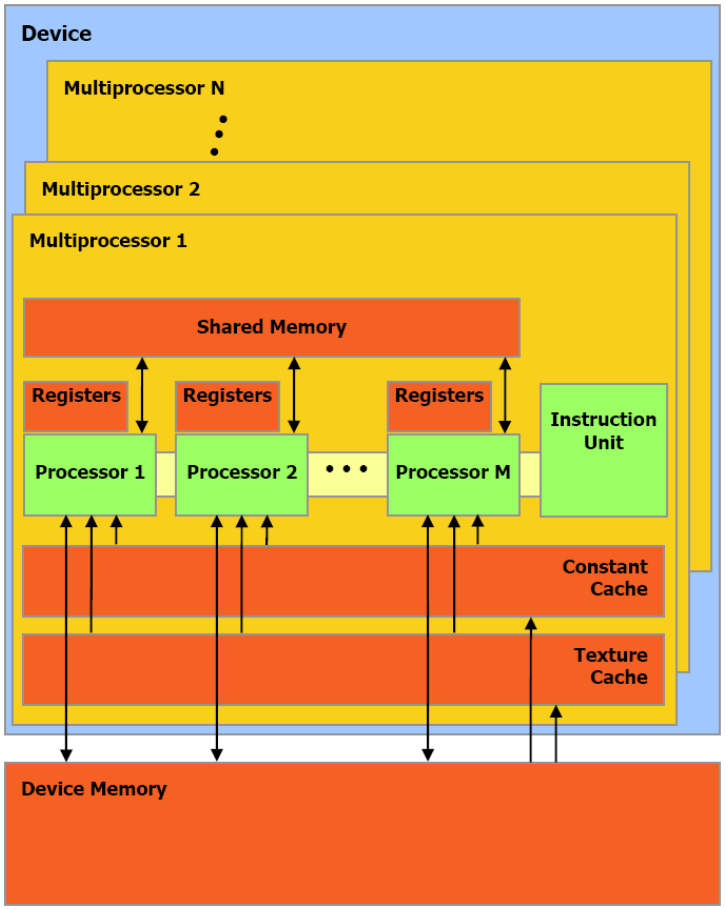
GPU memory hierarchy

- Memory needs to be properly used to avoid performance bottlenecks
- The **programmer** needs to write code that
  - Is **arithmetic intensive** (high math/memory instructions ratio)
  - Re-uses data fetched from global memory by means of shared memories

NVIDIA GeForce 8 (2006)

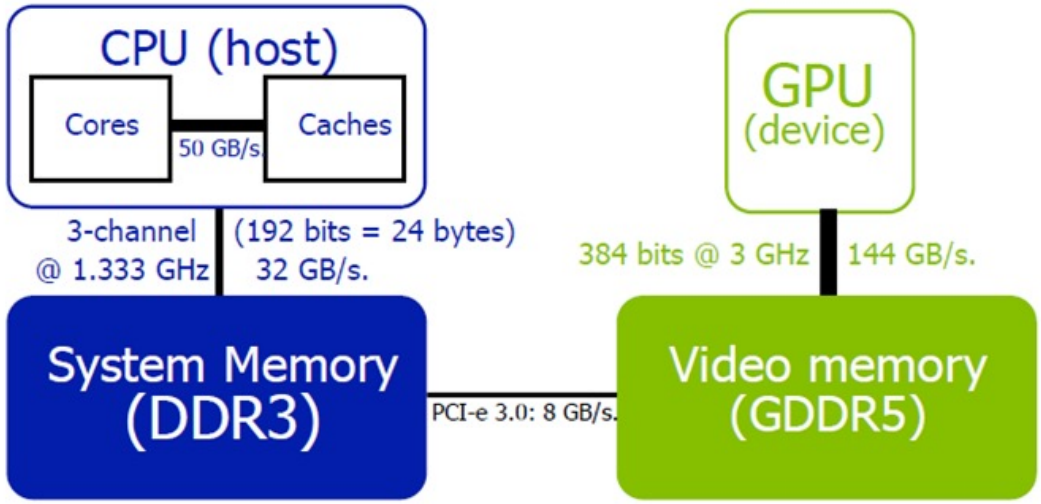
- Each multiprocessor is provided with
  - 16KB shared memory
  - 64KB constant cache
  - 8KB texture cache

- Each thread can access all device memory locations
- Different latencies:
  - Shared memory: 2 cycles
  - Device memory: 300 cycles
- Different address spaces for different memories



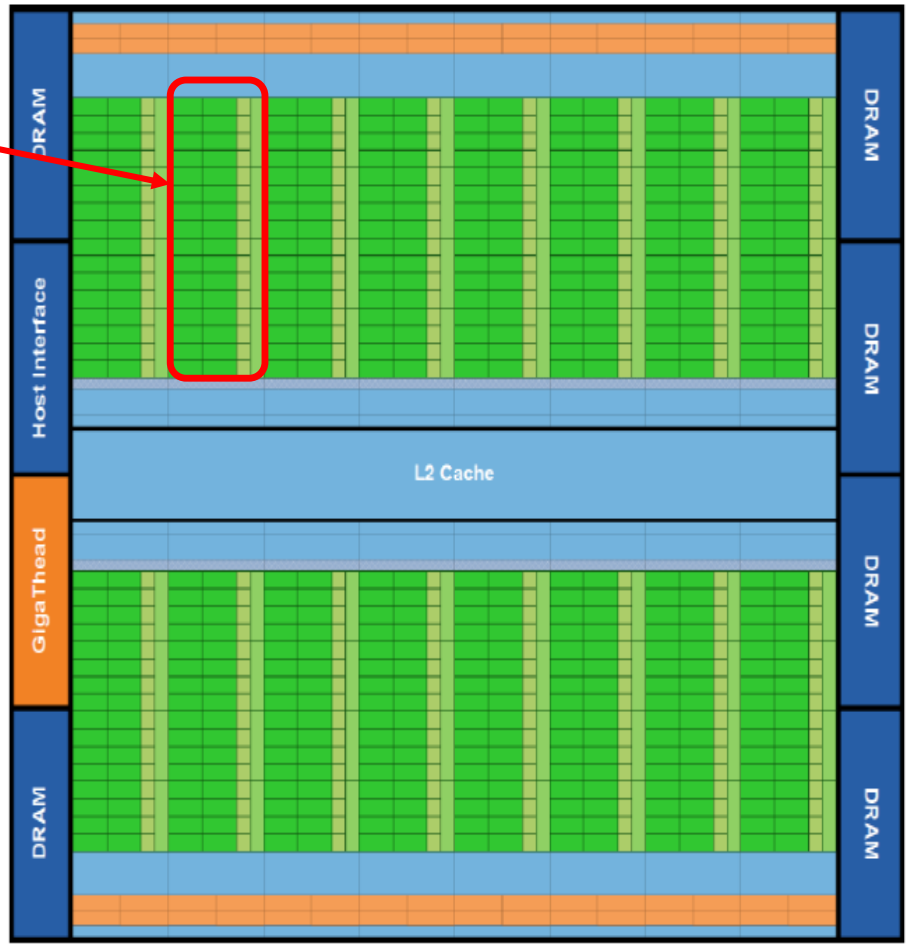
CPU-GPU connection

- Another relevant aspect is the CPU/GPU transmission bandwidth
- PCI Express 3.0 bandwidth: 8GB/s on each direction
  - Subsequent PCI Express versions have larger bandwidth but does not solve the problem



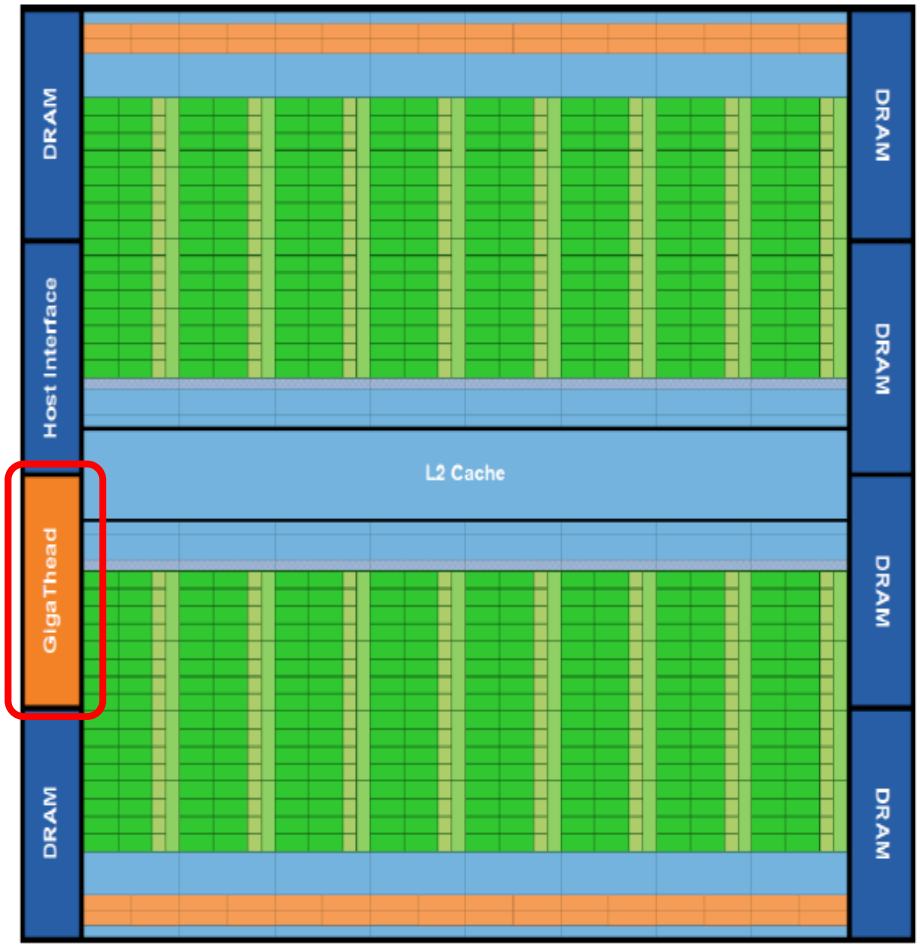
NVIDIA Fermi (2010)

- 16 streaming multiprocessors
- Each multiprocessor has 32 streaming cores
- 6 64-bit memory partitions
- 1 global scheduler



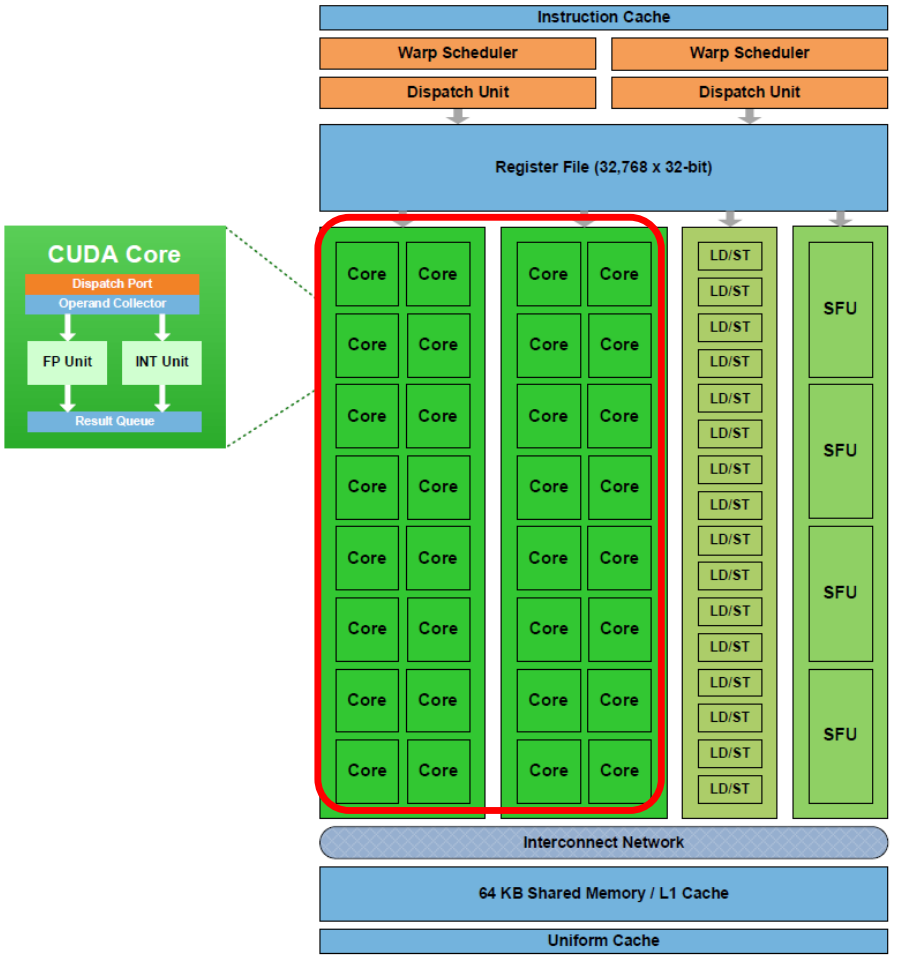
NVIDIA Fermi (2010)

- The programmer divides threads to be spawn for a kernel function in **blocks**
- The global scheduler dispatches blocks among streaming multiprocessors



Streaming Multiprocessor (SM)

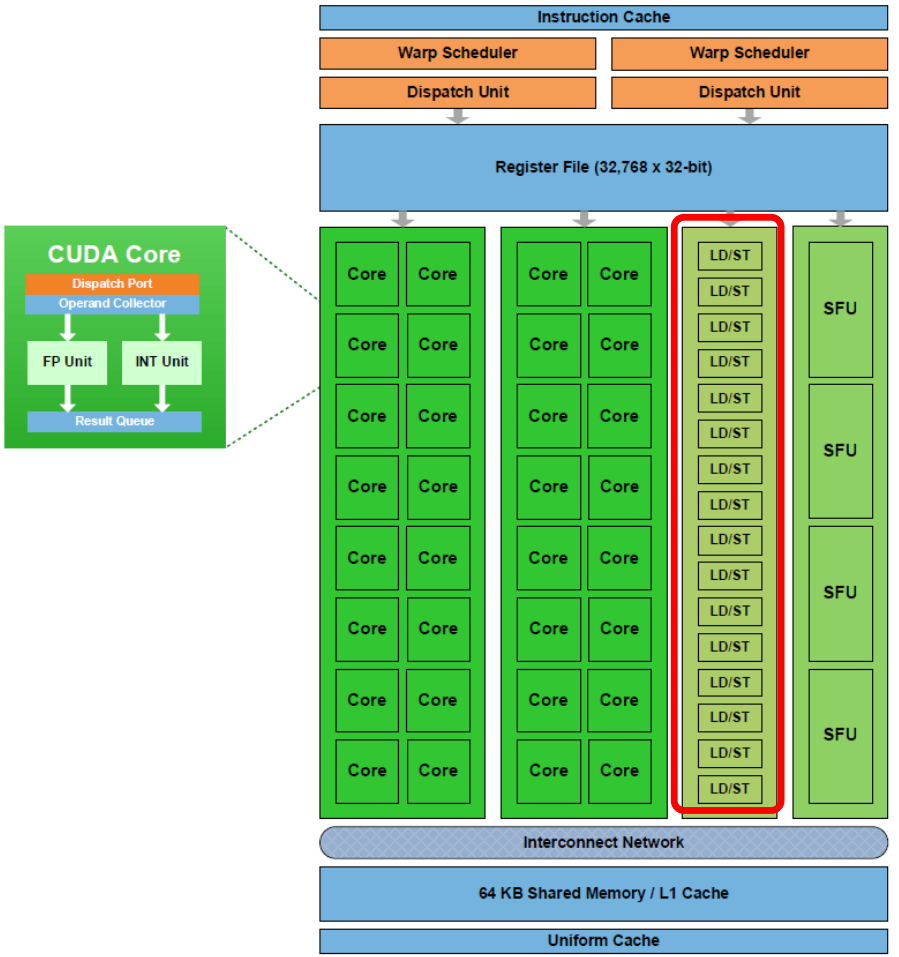
- 32 streaming cores
  - 32 bit pipelined integer arithmetic unit – with support for 64 bit operations (1 cycle)
  - IEEE 754-2008 single/double-precision floating point unit providing fused multiply-add instructions (1/2 cycles)



- Larger number of processing cores
- Higher datatype precision

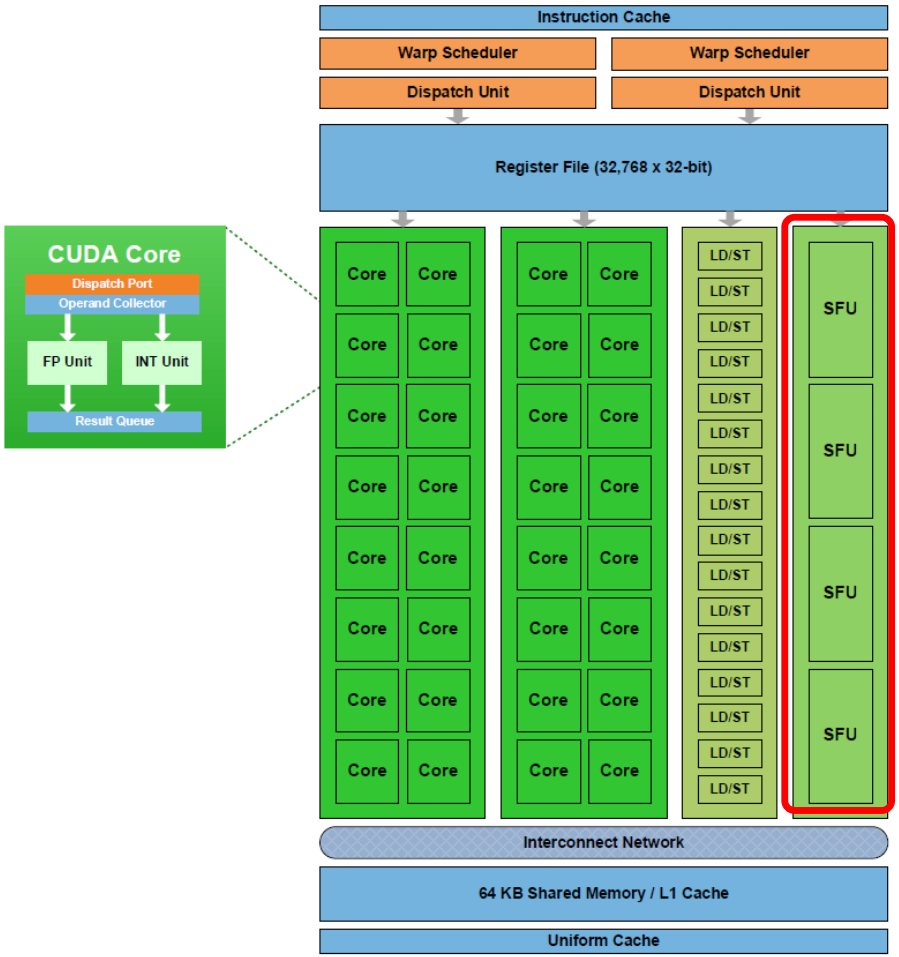
Streaming Multiprocessor (SM)

- 16 load/store units
  - Concurrent access to data in each address of the cache or DRAM (1 cycle with cache hit)
  - Int-float (and viceversa) casts performed while copying data from memory to register file
  - Support for atomic math instructions on memory



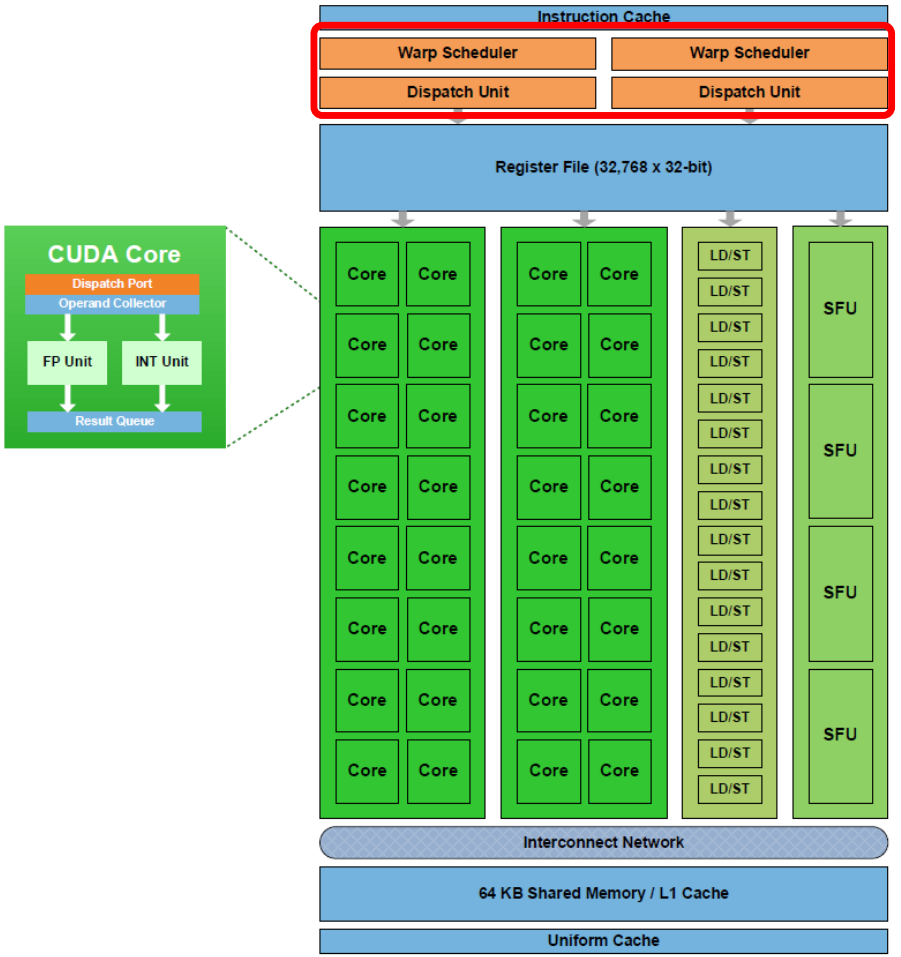
Streaming Multiprocessor (SM)

- 4 special function units (SFUs)
  - For transcendent functions (sine, cosine, square root, …) (8 cycles)
  - Decoupled from the dispatching units to improve performance
- Shader clock = 2x GPU clock to optimize area/performance



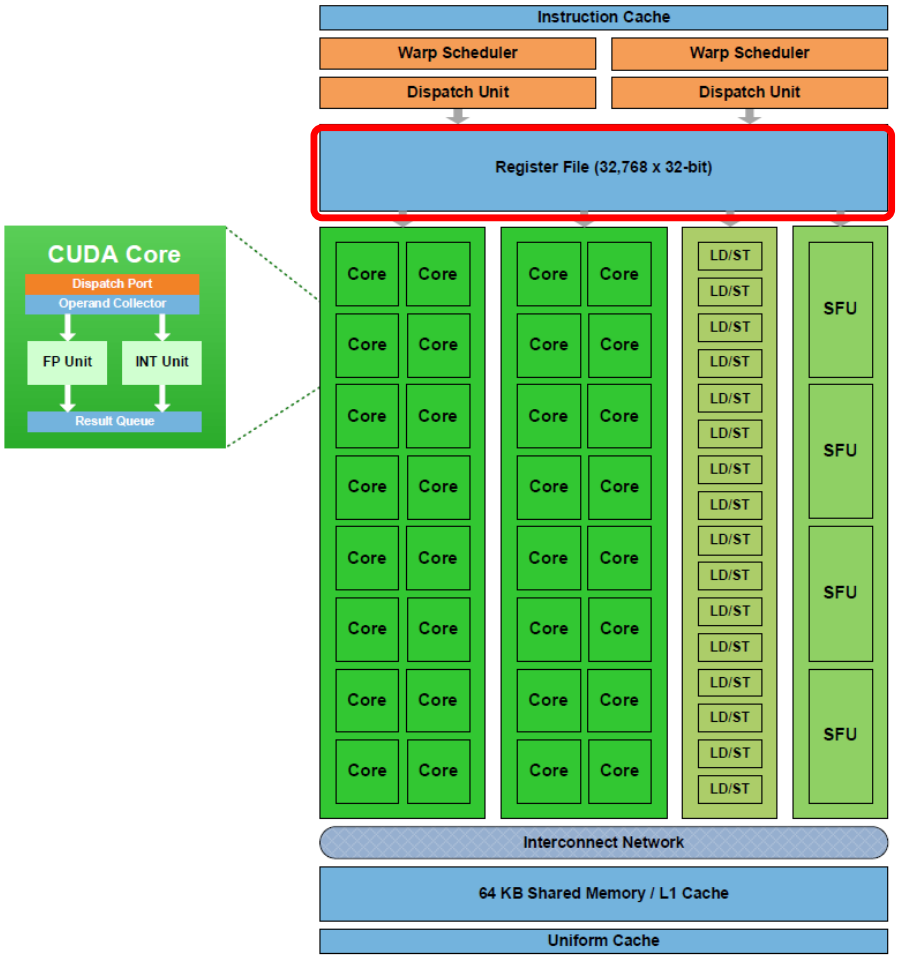
Streaming Multiprocessor (SM)

- Block threads are split in groups of 32 elements, called **warp,** sharing an instruction stream
- The SM has 2 scheduling and dispatching units
  - Two warps are selected each clock cycle (fetch, decode and execute two warps in parallel)



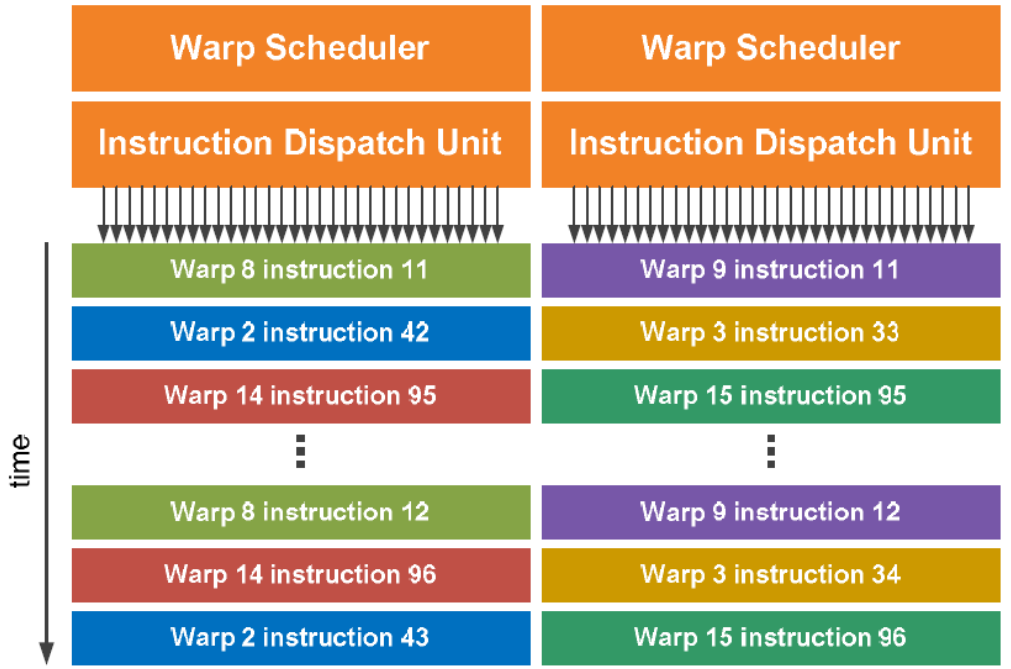
Streaming Multiprocessor (SM)

- The register file may host up to 48 interleaved warps
  - 1536 threads per SM!
  - Globally 24576 threads!



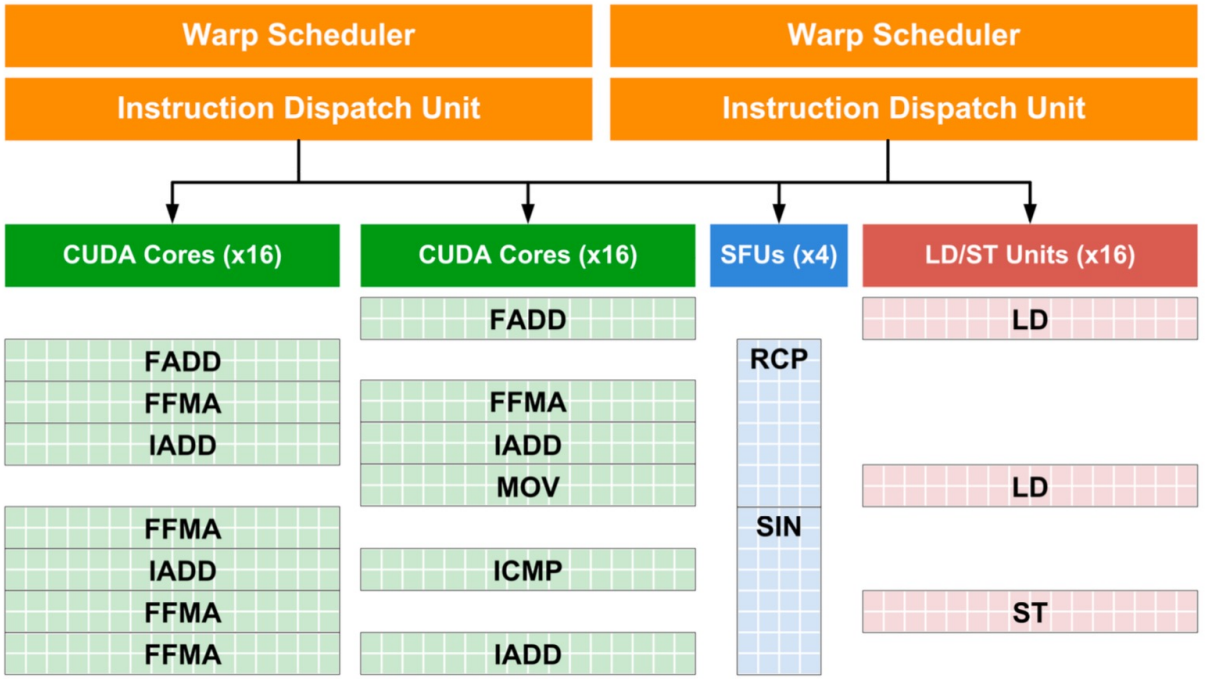
Streaming Multiprocessor (SM)

- Each clock cycle the scheduler selects a warp that is ready to be executed
- Warps are independent -> no dependency check is required
- A scoreboard is used to keep track of the status of the warps



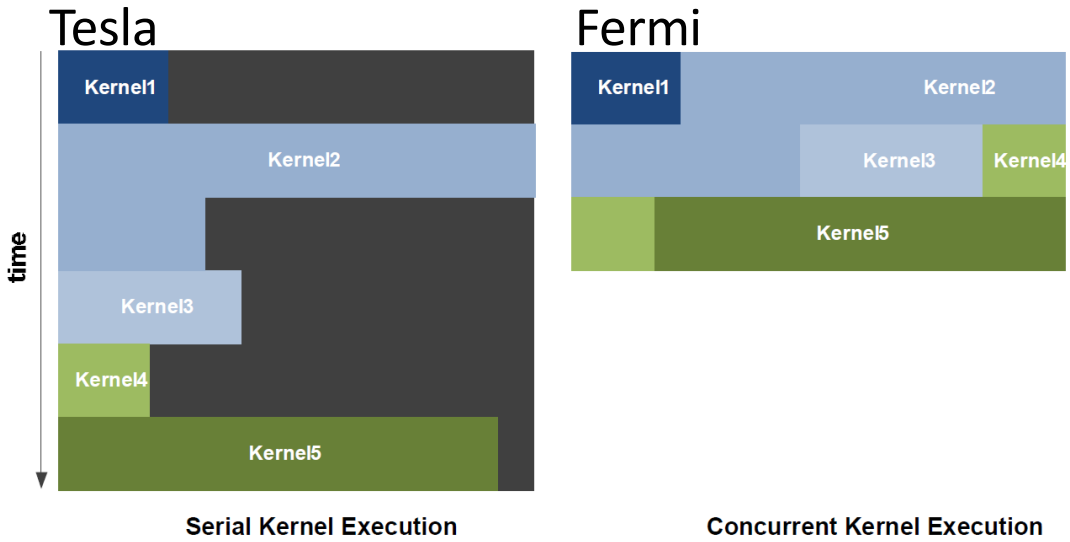
Streaming Multiprocessor (SM)

- Each scheduler may execute an instruction on 16 ALU cores, 16 load/store units, or 4 SFUs
- Double precision instructions do not support dual dispatching since they requires 2 ALU core for each operation



Concurrent executions

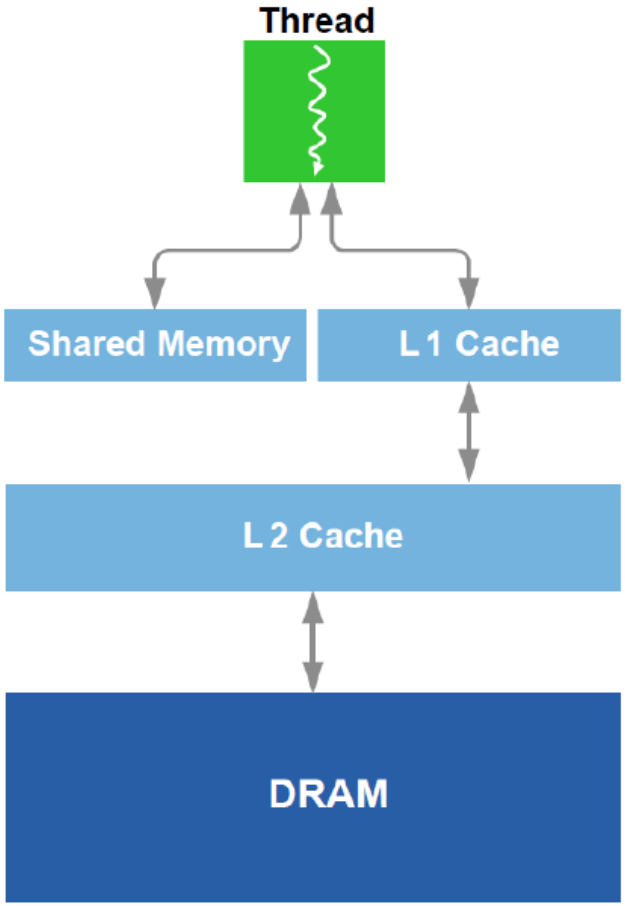
- Support concurrent execution of multiple compute kernels from the same application



- A single application can use the GPU per time
  - But there is the support for fast application context switch (25us)

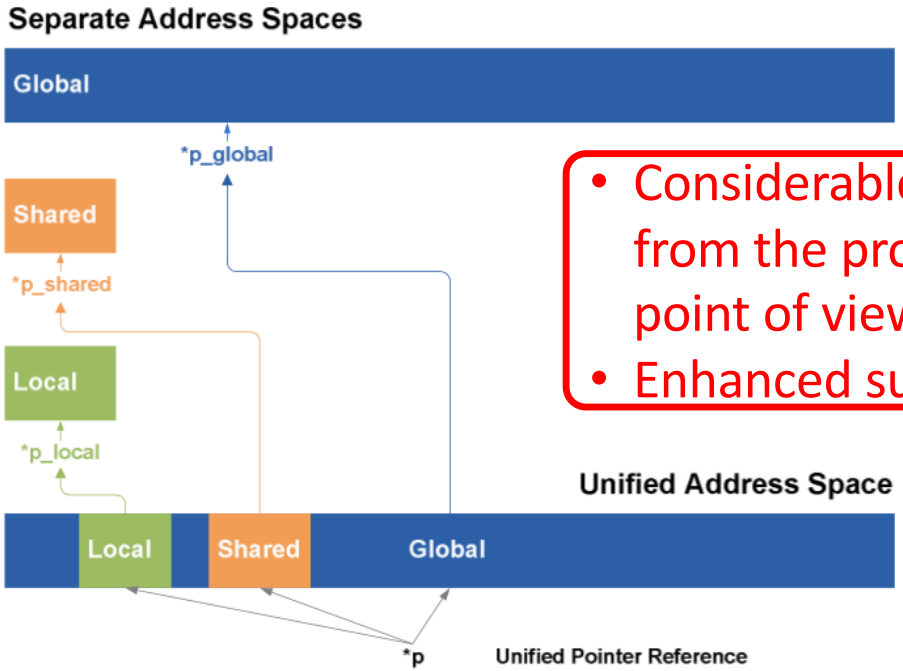
Memory hierarchy

- Restructured cache hierarchy
  - Introduction of private L1 cache and chip-level L2 cache
  - Texture cache has been removed from L1 since not efficient for general purpose computing

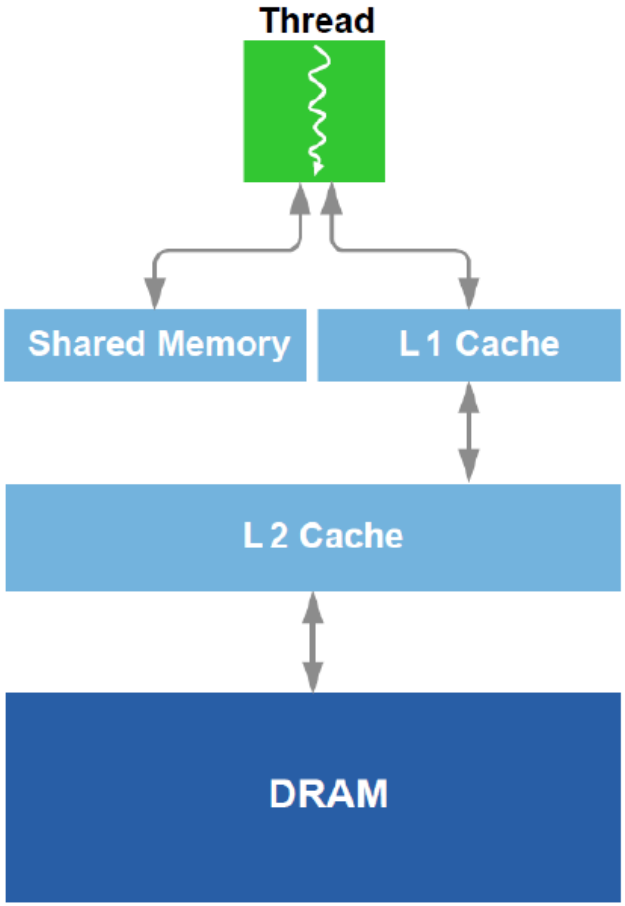


Memory hierarchy

- Unified address space on local, shared and global memory

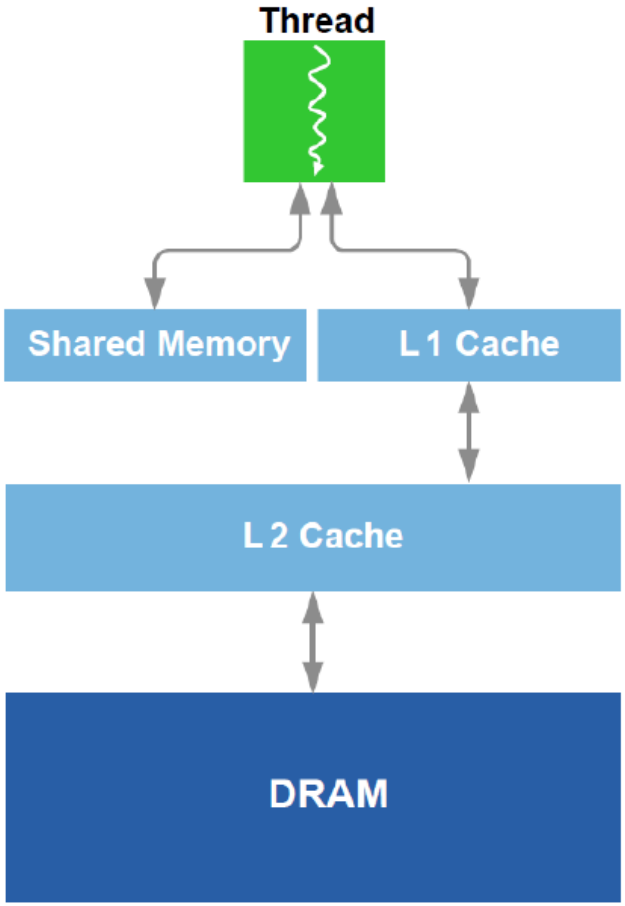


- Considerable advantages from the programming point of view
- Enhanced support to C++



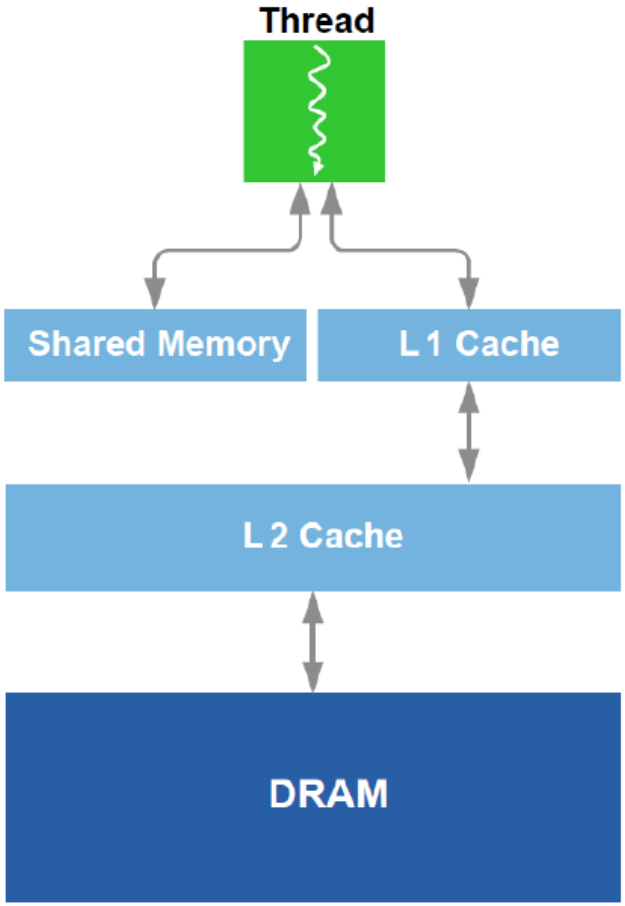
Memory hierarchy

- Possibility for the programmer to configure the local memory
  - 16KB L1 cache and 48KB shared memory or viceversa
  - Shared memory allow threads of the same “block” to cooperate by means of the scratchpad memory



Memory hierarchy

- Register spilling on L1
- Fast atomic memory operations on L2 memory
  - E.g., read-modify-write, compare-and-swap
  - Avoid the necessity of mutex and semaphores
  - Efficient implementation of several parallel algorithms (sorting, reduction, …)
- L1 and L2 avoid the necessity to access the global memory



NVIDIA Kepler (2012)

- Architecture similar to the Fermi one with performance and power efficiency improvements

Streaming Multiprocessor (SMX)

- New SM architecture (now called SMX)
  - 192 single-precision int/fp streaming core
  - 64 double-precision int/fp streaming cores
  - 32 special function units
  - 32 load/store units
  - 8 texture filtering units
  - Shader clock = GPU clock

Streaming Multiprocessor (SMX)

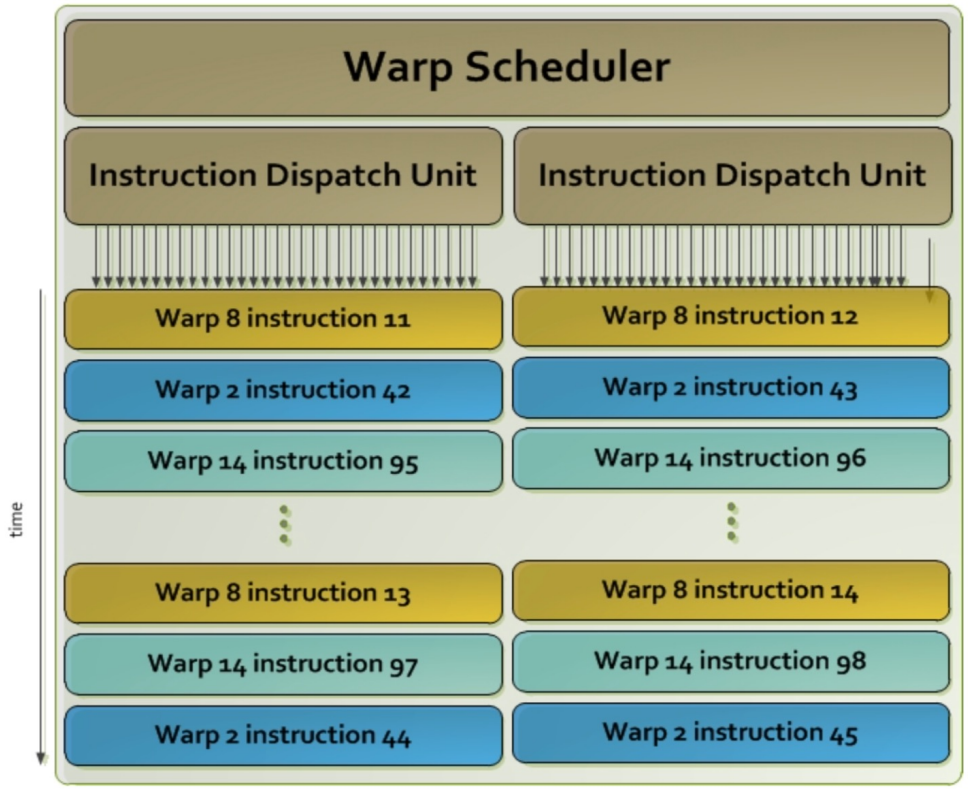
- New scheduler
  - 4 warp schedulers
  - 2 dispatchers per warp
  - Each cycle 2 independent instructions per warp are executed
  - Double precision instructions can be paired with other ones

Exploit ILP

Streaming Multiprocessor (SMX)

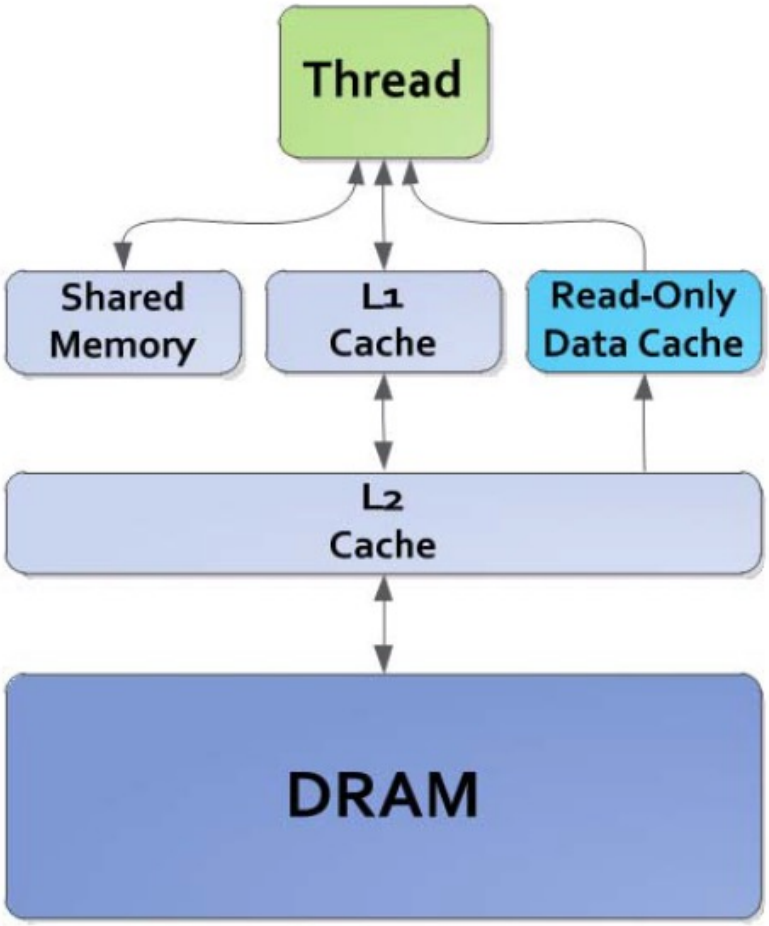
Exploit ILP

- Similarly, to the Fermi architecture includes
  - Register scoreboarding for long latency operations
  - Inter-warp scheduling decisions (e.g., pick the best warp to go)
- ..but the scheduler is simpler since it exploits instruction latencies determined by the compiler



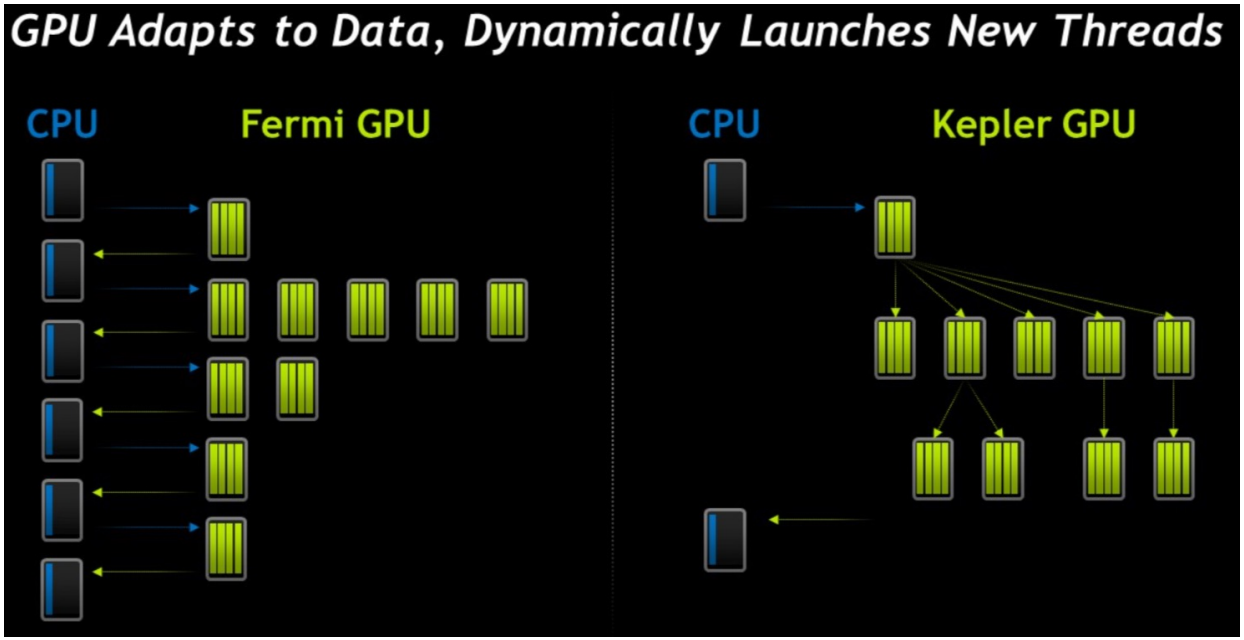
Memory hierarchy

- Doubled Fermi cache size: 128KB L1, 1536KB L2
  - Larger configurability of L1/shared memory
- Introduced a Read-only cache (similar to a texture cache)
  - Explicitly used by the programmer
- Added shuffle instructions to exchange data among warp threads without using shared memory
  - Reduces synchronization time



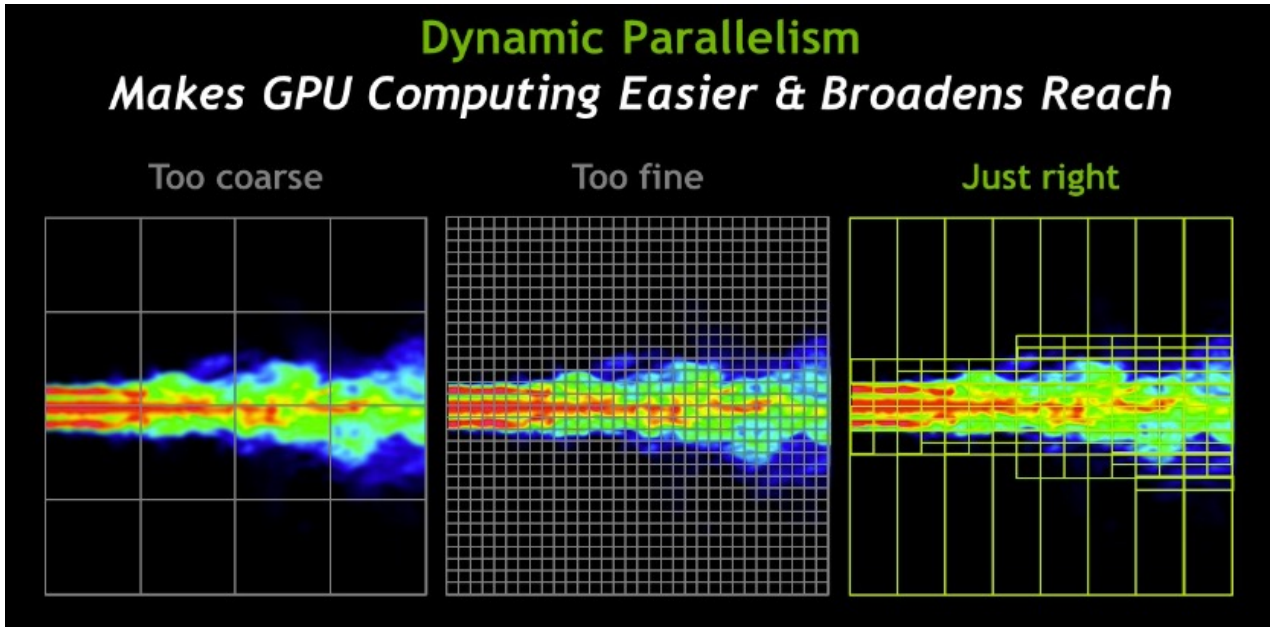
Dynamic parallelism

- A kernel can launch another kernel
  - Kernel launch, synchronization on results and work scheduling are managed directly by the GPU device



Dynamic parallelism

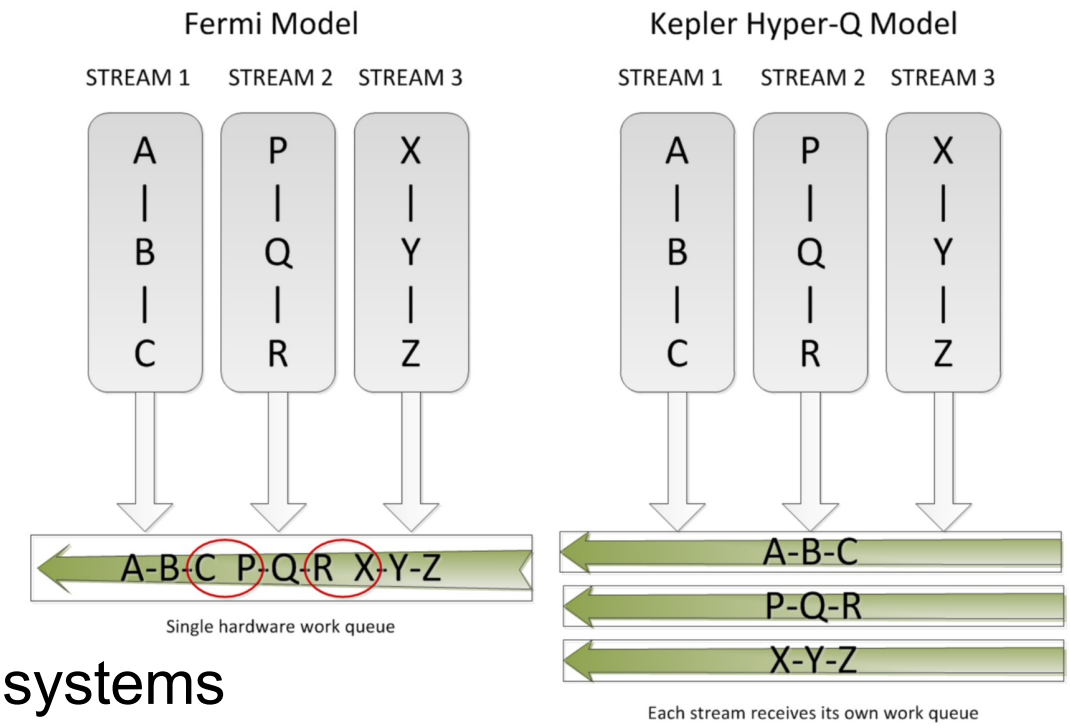
- A kernel can launch another kernel
  - This allows the optimization of recursive and data-dependent executions patters



Several work queues

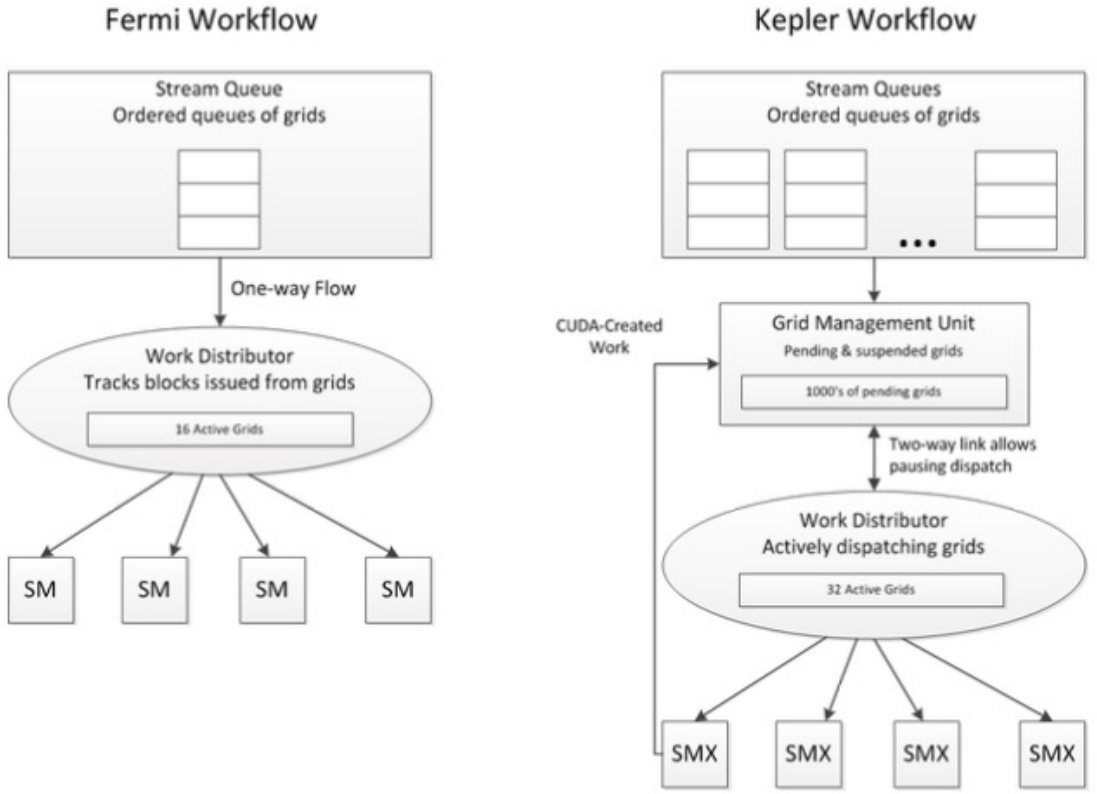
- Hyper-Q mechanism offering 32 HW-managed work queues
- It enables multiple CPU cores to launch work on a single GPU simultaneously

- Advantages:
  - Avoidance of false intra-stream dependencies
  - Dramatically increase of GPU utilization and reduction of CPU idle times
  - Particularly useful in MPI-based parallel computer systems



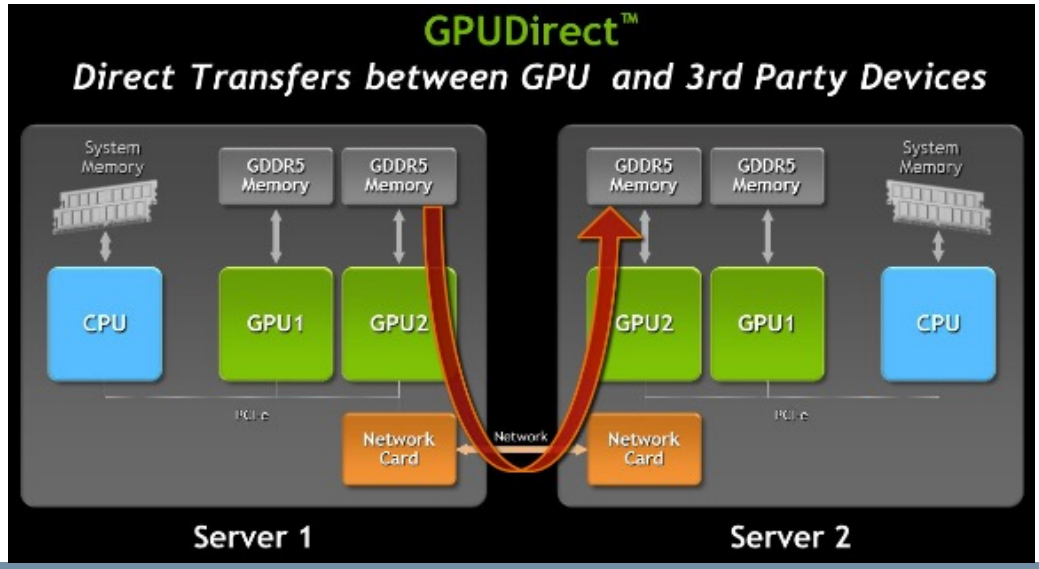
Grid management unit

- The grid management unit has been redesigned



Multi-machine systems

- GPUDirect technology allows direct access to GPU memory from third-party devices
  - DMA-based data transmission without CPU involvement
  - Optimized for MPI send/receive

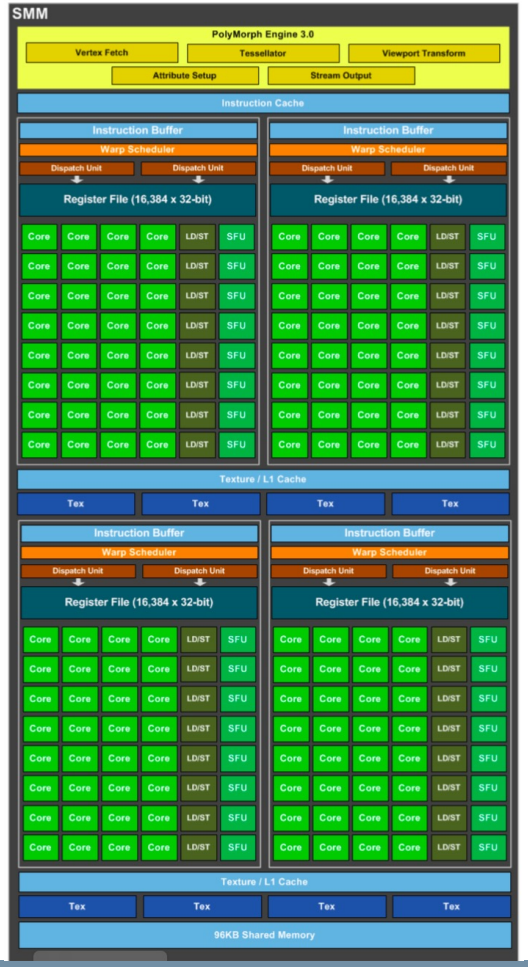


NVIDIA Maxwell (2014)

- Similar architecture to Kepler

NVIDIA Maxwell (2014)

- New streaming multicore (SMM)
  - SCs are partitioned in 4 groups each one assigned to a single wrap scheduler
- Memory hierarchy
  - Separate shared memory
  - L1 cache shared with texture cache
- Performance/power improvements

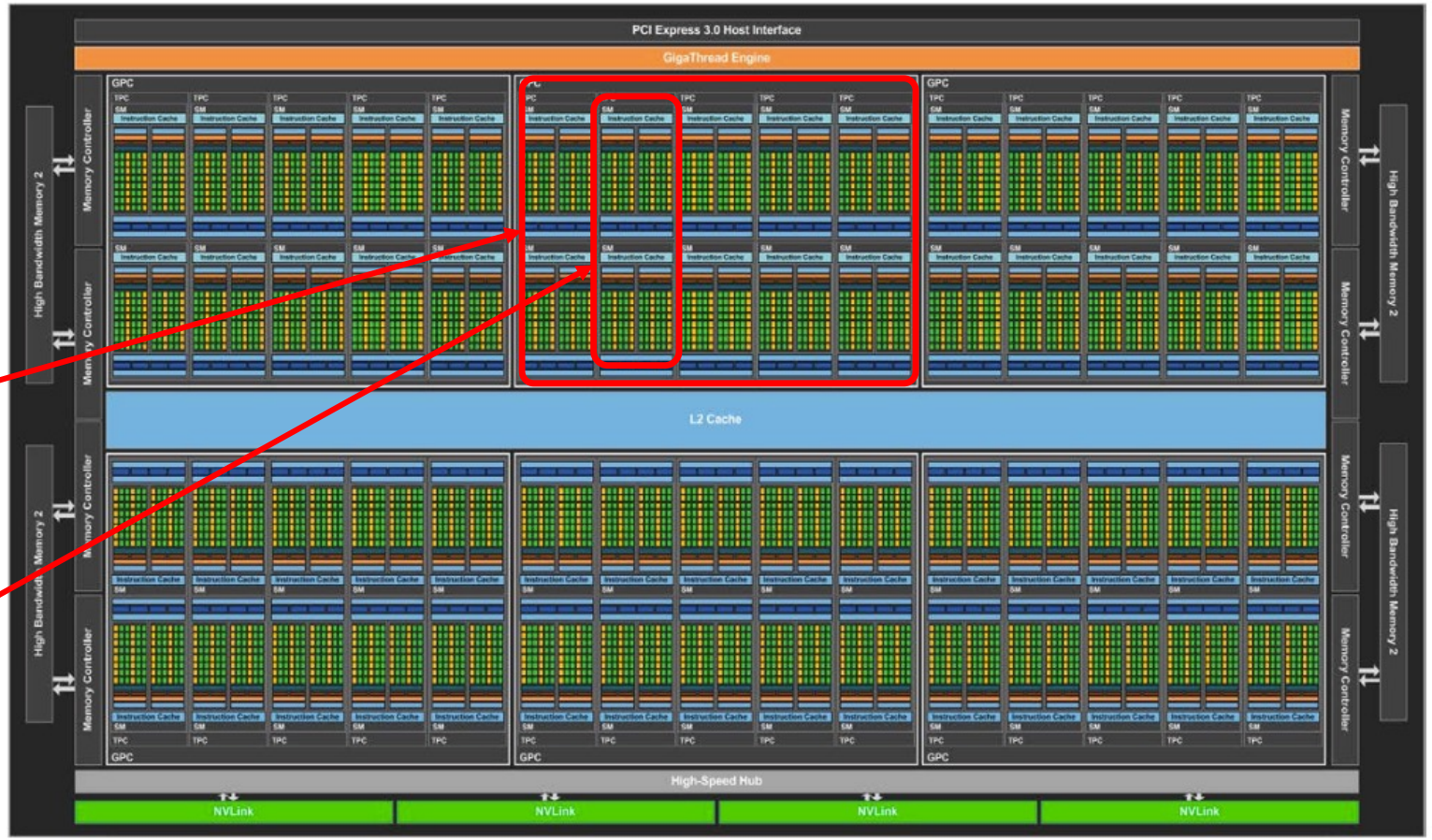


NVIDIA Pascal (2016)

- More hierarchical organization

Graphics Processing Clusters (GPCs)

Texture Processing Clusters (TPCs)



Streaming multicore

- Application preemption at single instruction granularity
  - It allows execution of multiple applications in time multiplexing
- Further performance/power improvements

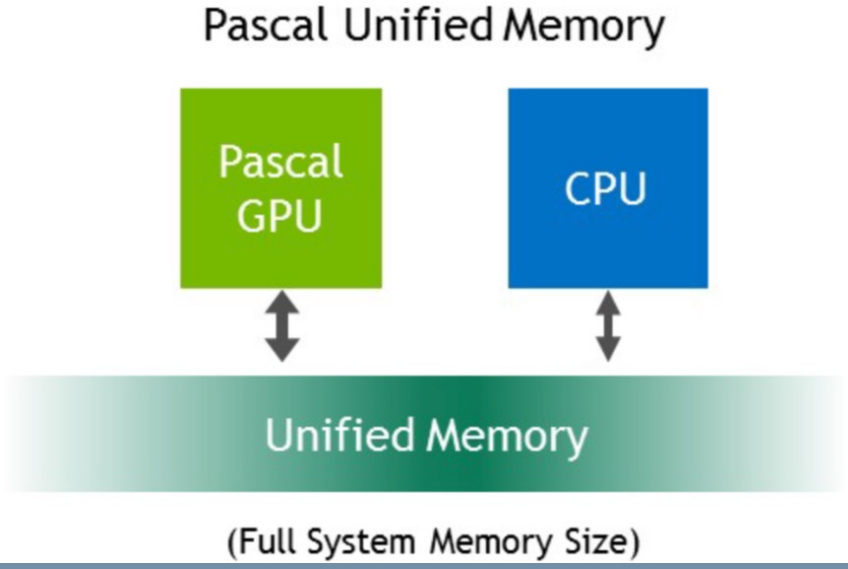
Streaming multicore

- Separate cores for single precision and double precision operations
- Introduction of half precision floating point instructions (x2 throughput w.r.t. single-precision one)

- Double precision arithmetic for linear algebra, numerical simulation and quantum chemistry, …
- Half precision for deep learning

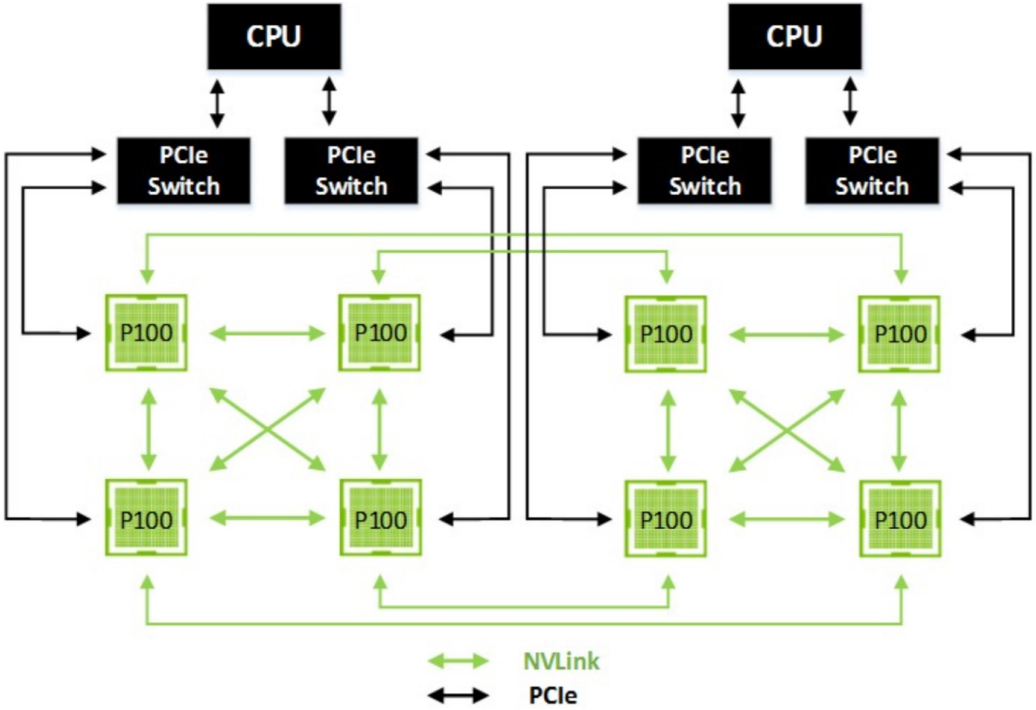
Unified memory

- Unified virtual addressing between CPU and GPU
  - Memory pages are transparently transmitted between CPU and GPU memories
  - Automatic handling of page fault and global data coherence



NVLink

- NVLink: new high-speed interface (160GB/s bidirectional) to enable multi-GPU architectures for high performance computing



NVIDIA Volta (2017)

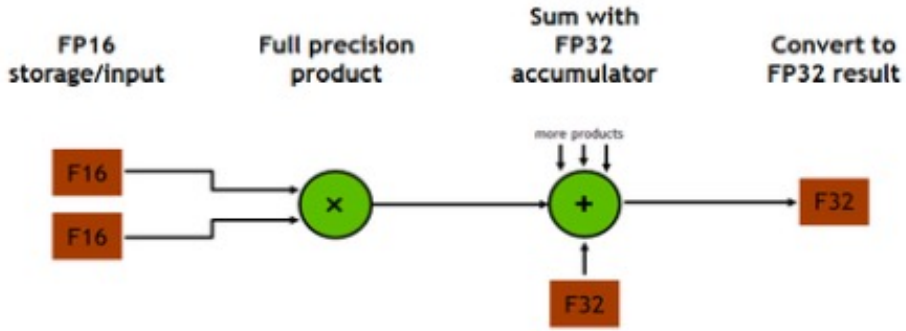
Streaming multicore

- Streaming multicore partitioned in processing blocks
- Separate floating point and integer cores
- New Tensor cores
  - For matrix multiplication and accumulation
  - Specific for DNN applications
- Independent thread scheduling

Tensor cores

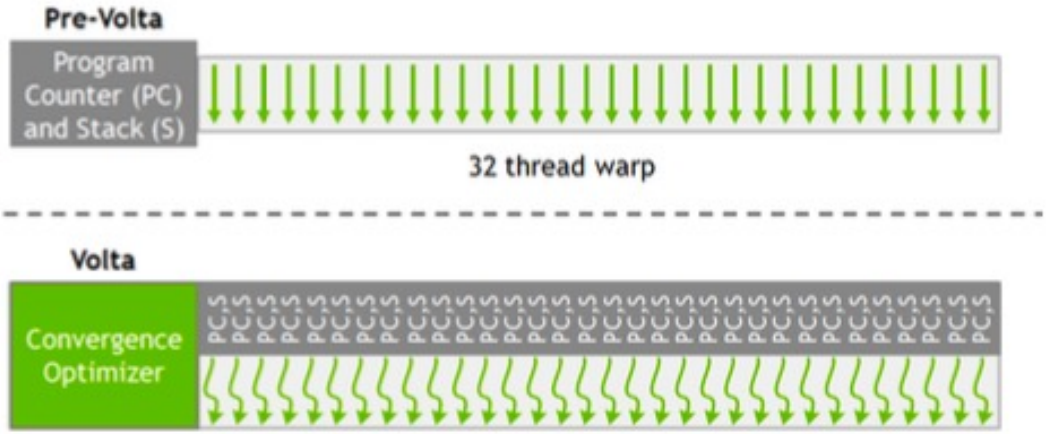
- Tensor cores perform multiplication and accumulation of 4x4 matrices

- Mixed precision operations



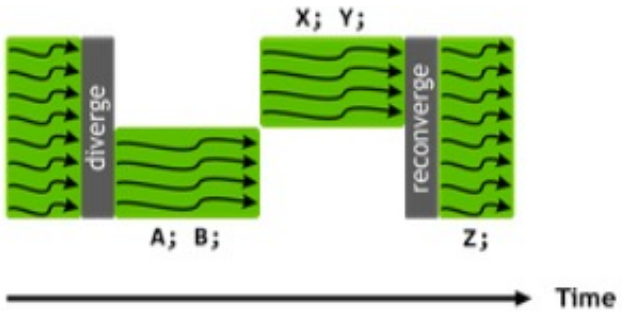
Independent thread scheduling

- Pre-Volta architecture
  - Per-warp execution state
- Volta architecture
  - Per-thread execution state

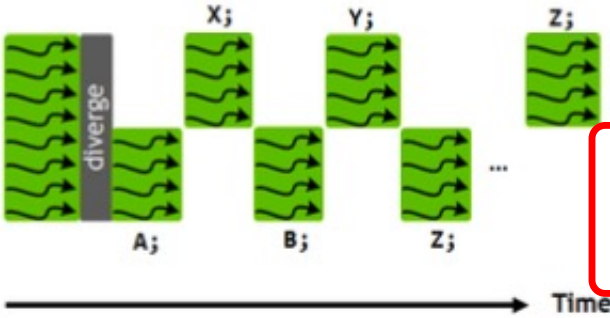



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

pre-Volta architecture



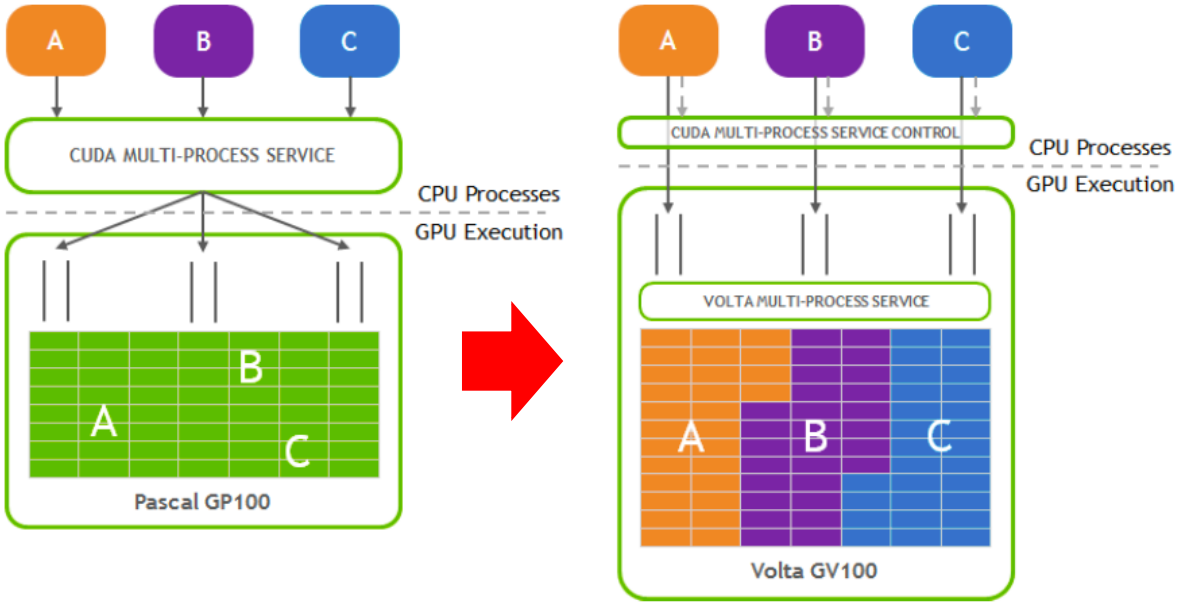
Volta architecture



If b; has a data dependency on a; then x; is scheduled

Multi process server (MPS) support

- MPS allows multiple applications to run concurrently on separate resources
  - Limitation: Memory system shared among processing resources



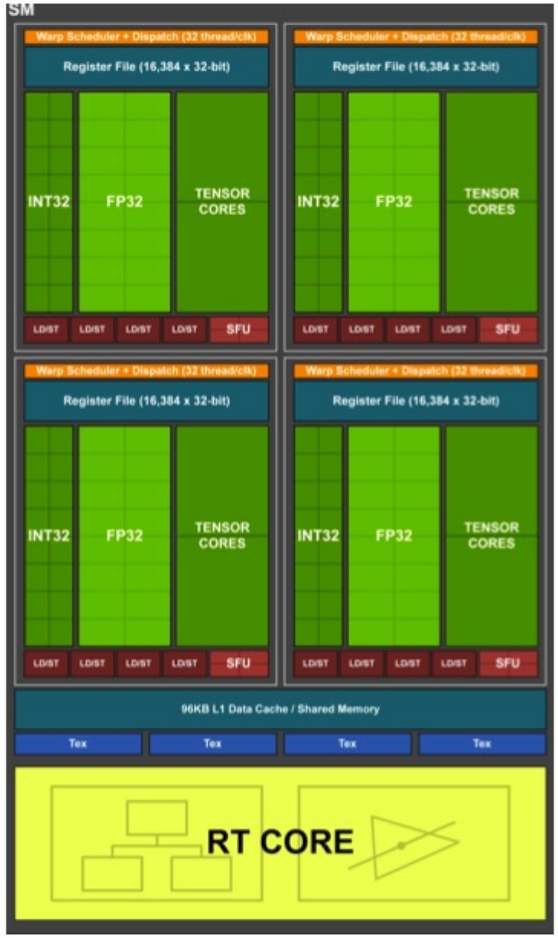
Pascal architecture

Volta architecture

NVIDIA Turing (2018)

Streaming multiprocessor

- Main enhancements on graphics pipeline acceleration
  - E.g., a new RT unit for ray tracing
- Relevant performance improvements on deep learning acceleration



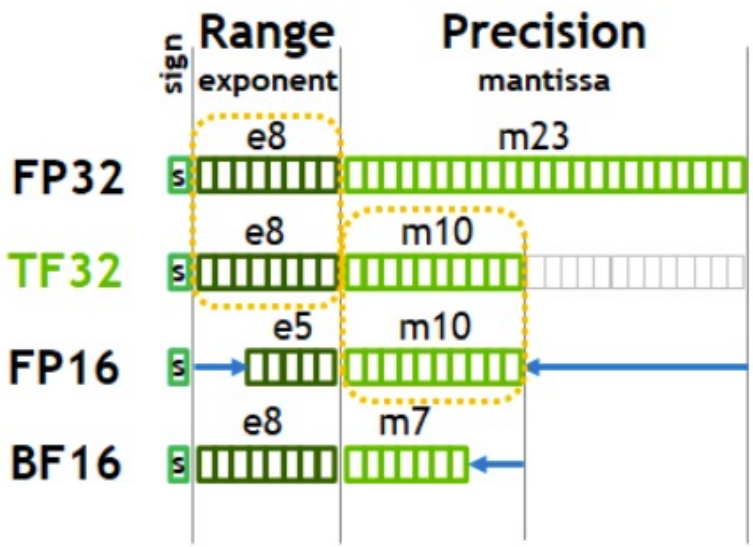
NVIDIA Ampere (2020)

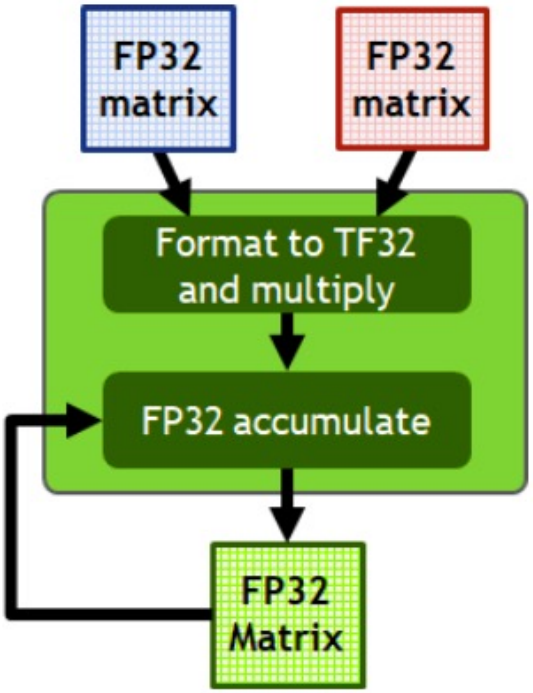
Streaming multiprocessor

- New tensor cores
  - Accelerating different data types
  - Exploiting matrix sparsity
- Virtualization

New tensor cores

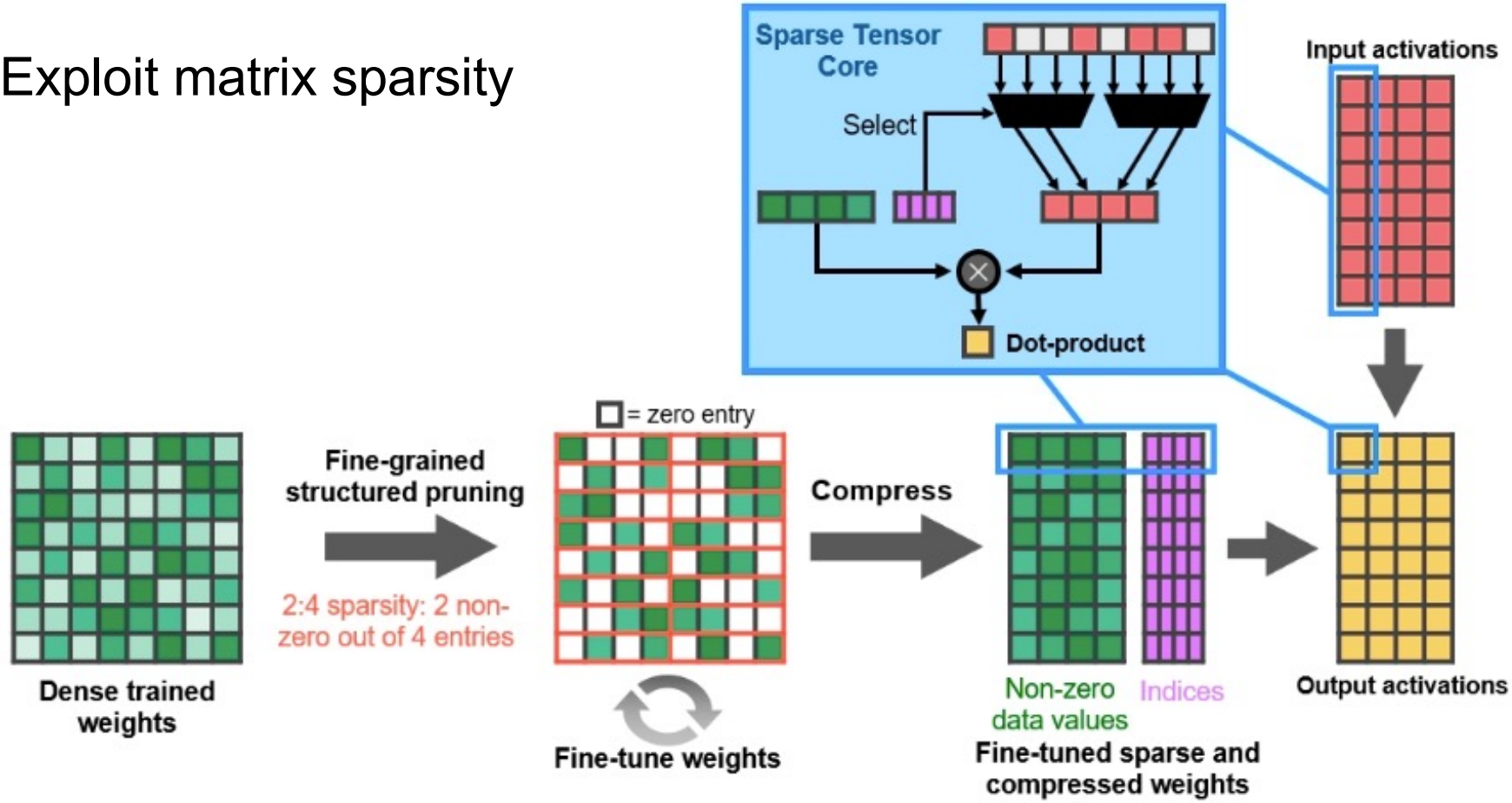
- Acceleration for all data types: FP16, FP64, INT8, INT4, binary, …





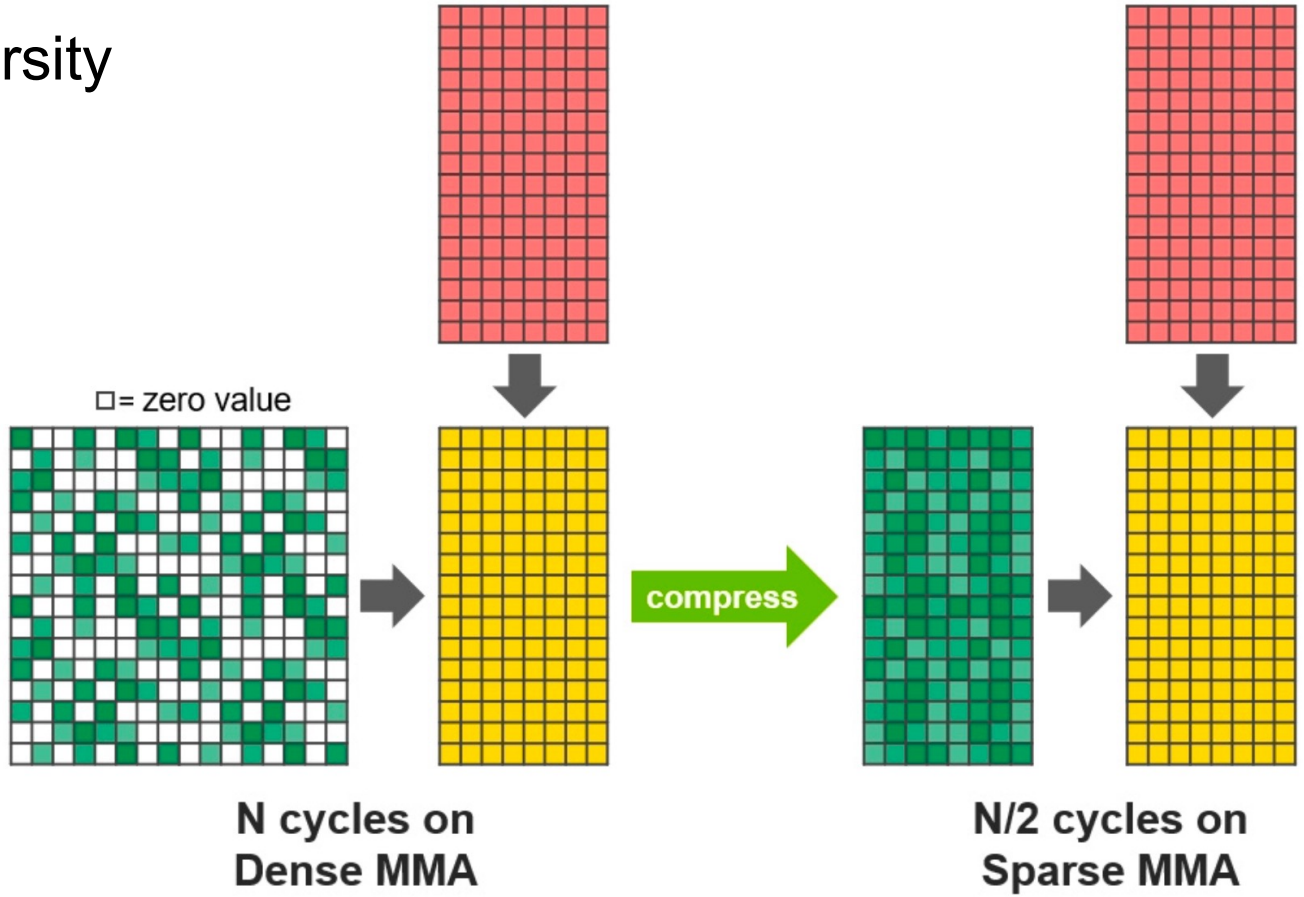
New tensor cores

- Exploit matrix sparsity



New tensor cores

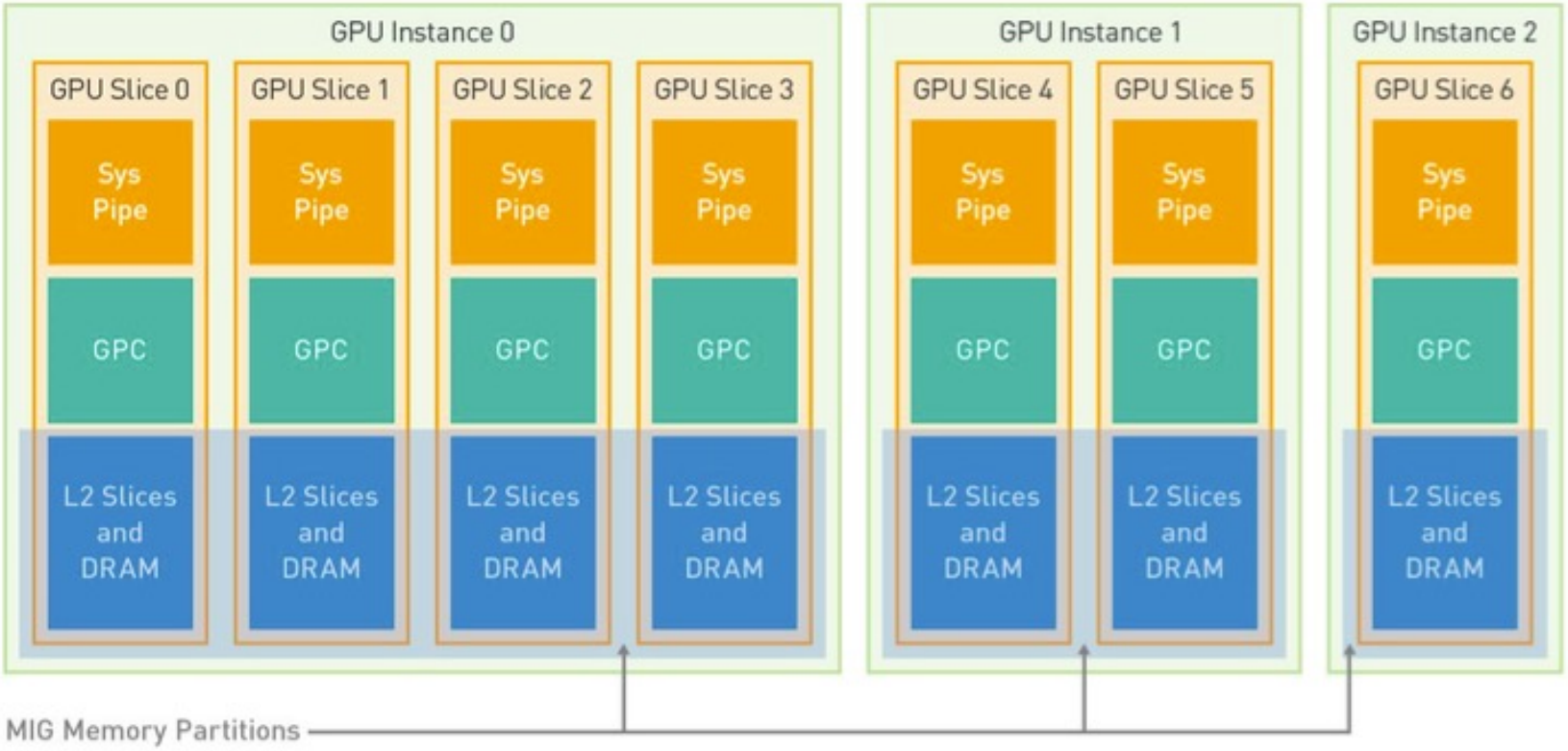
- Exploit matrix sparsity



Avoid multiplications by zero

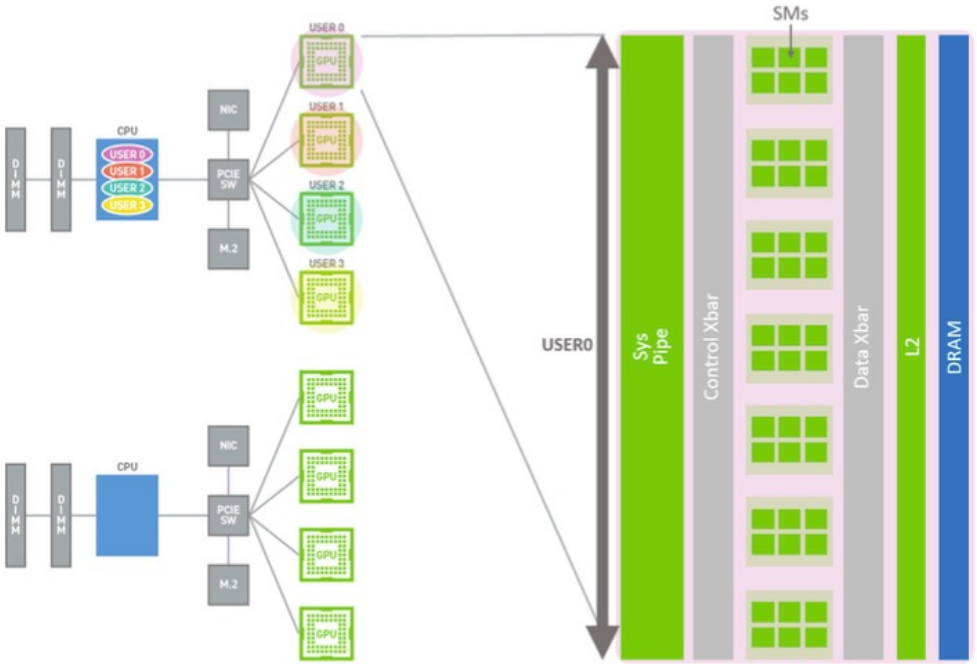
Multi Instance GPU (MIG) virtualization

- The GPU can be partitioned in 7 separate virtual GPUs

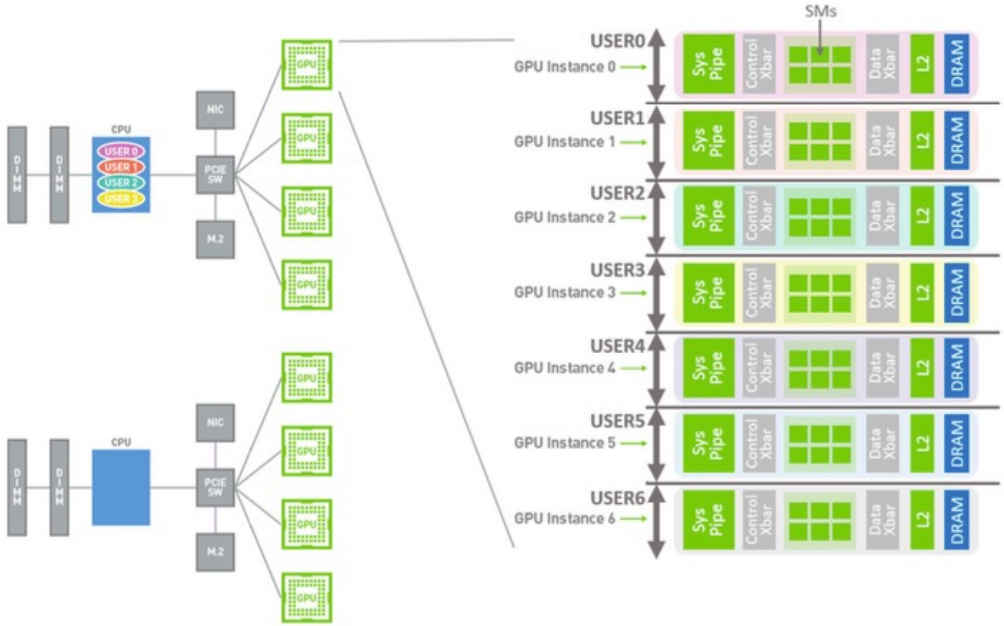


Multi Instance GPU (MIG) virtualization

- Cloud service provider's multi-user node



previous GPUs



Ampere GPU

Summary on NVIDIA GPUs

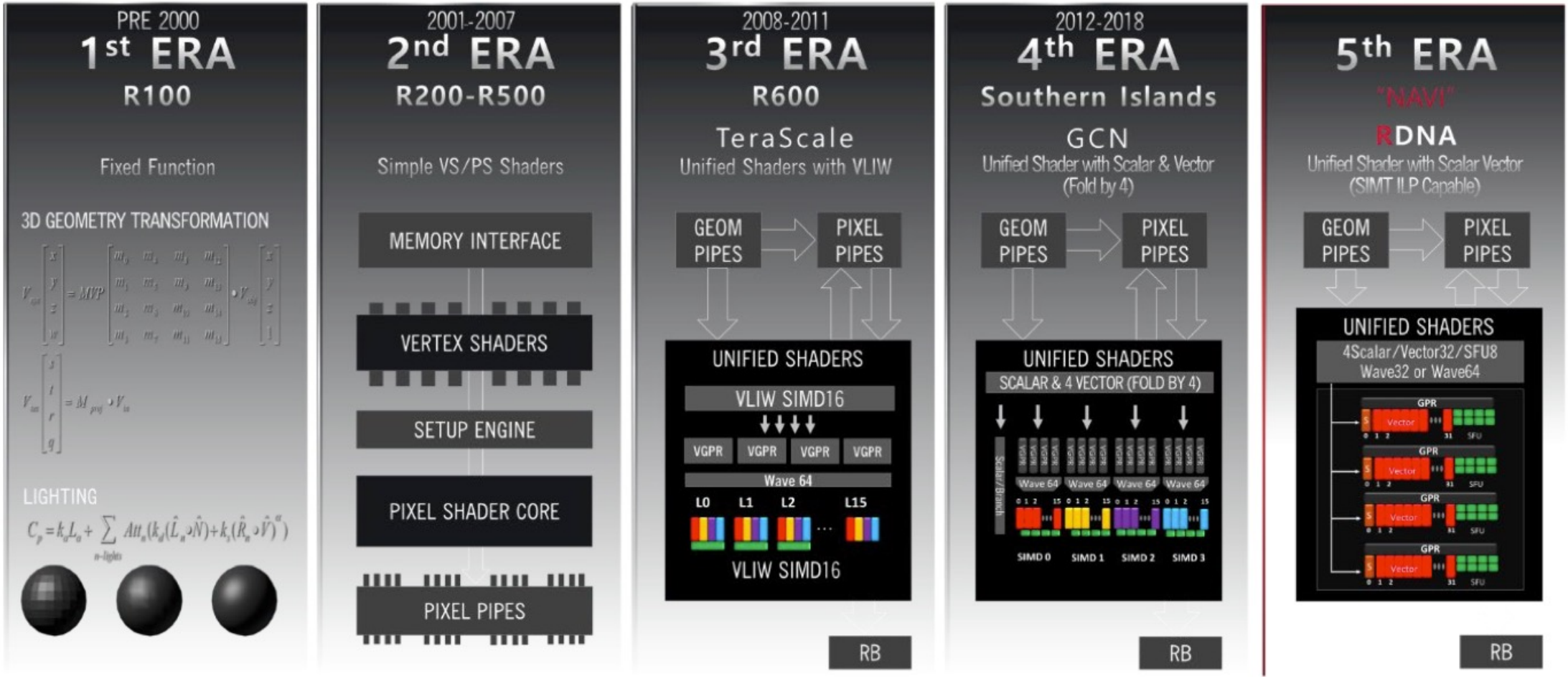
	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"	"Turing"	"Ampere"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**	**GV100**	**TU104**	**GA100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0	7.0	8.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56	84	72	128
FP32 CUDA Cores / SM	32	32	192	192	128	64	64	64	64
FP32 CUDA Cores	512	512	1,536	2,880	3,072	3,584	5,376	4,608	8,192
FP64 Units	-	-	512	960	96	1,792	2,688	-	4,096
Tensor Core Units							672	576	512
Threads / Warp	32	32	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64	64	64
Max Threads / SM	1,536	1,536	2,048	2,048	2,048	2,048	2,048	2,048	2,048
Max Thread Blocks / SM	8	8	16	16	32	32	32	32	32
32-bit Registers / SM	32,768	32,768	65,536	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Thread	63	63	63	255	255	255	255	255	255
Max Threads / Thread Block	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config	Config	Config
	48 KB	48 KB	32 KB	32 KB			Up To	Up To	Up To
			48 KB	48 KB			96 KB	96 KB	164 KB
Hyper-Q	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes	Yes	Yes	Yes
Pre-Emption	No	No	No	No	No	Yes	Yes	Yes	Yes
Sparse Matrix	No	No	No	No	No	No	No	No	Yes

Other vendors…

- We have analyzed NVIDIA GPUs so far…
- There are many other GPU vendors
  - E.g.: AMD, ARM, …
- The overall GPU architecture is quite similar to the NVIDIA one

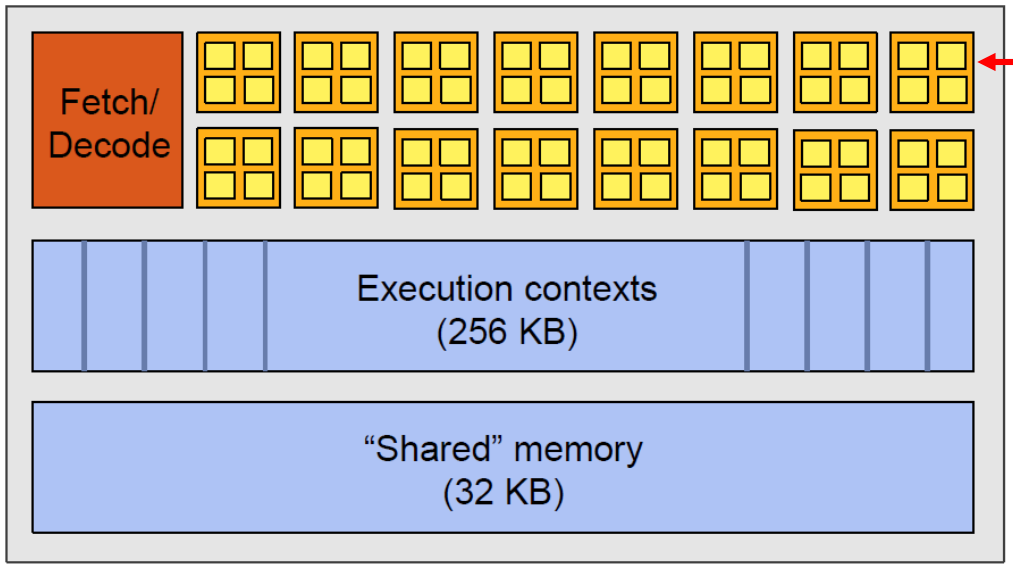
Just few examples!

AMD timeline



AMD Radeon HD 6970 Cayman (2010)

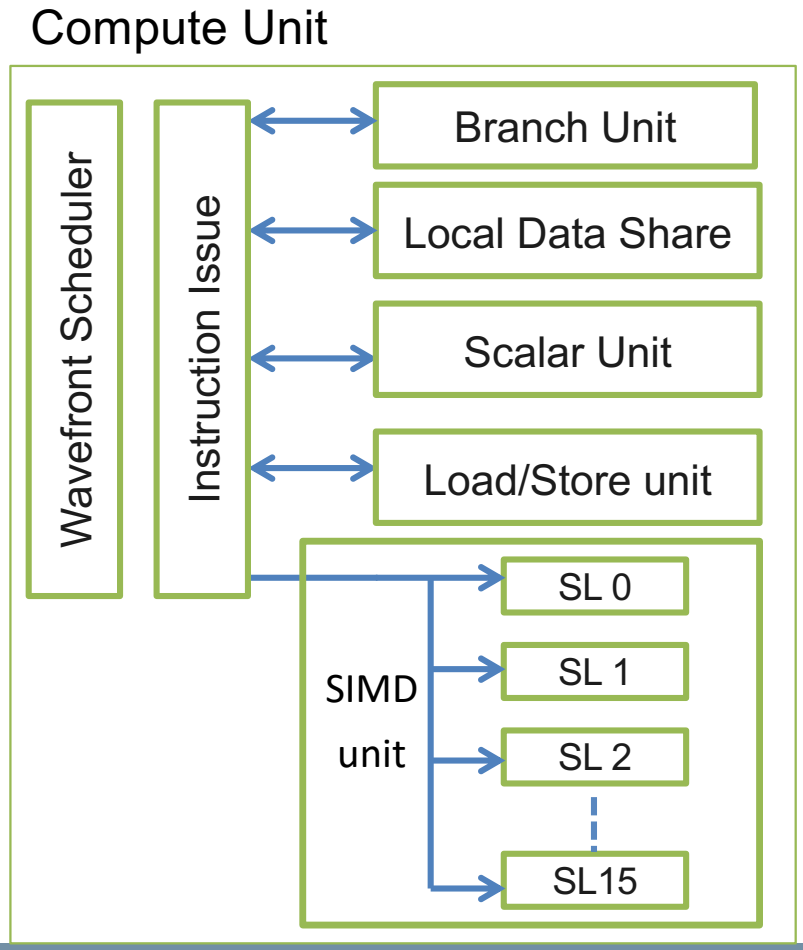
- The streaming multiprocessor is here called **compute unit**
- Threads are grouped in 64 elements (**wavefront**)
- Four clocks to execute an instruction for all fragments in a wavefront



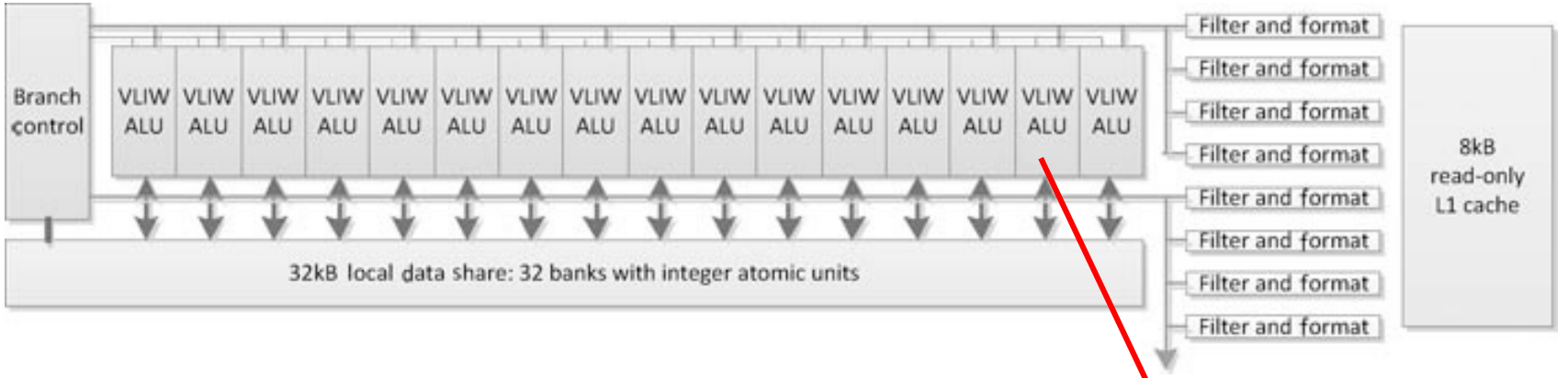
SIMD function unit, control shared across 16 functional units (up to 4 MUL-ADDs per clock cycle)
VLIW processing!

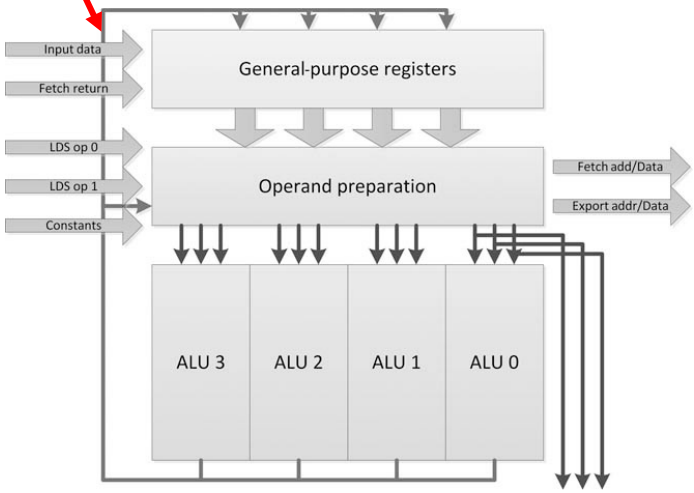
AMD Radeon HD 6970 Cayman (2010)

- The compute units contains also
  - A scalar unit
  - A branch unit
  - A shared memory (Local data share)



AMD Radeon HD 6970 Cayman (2010)

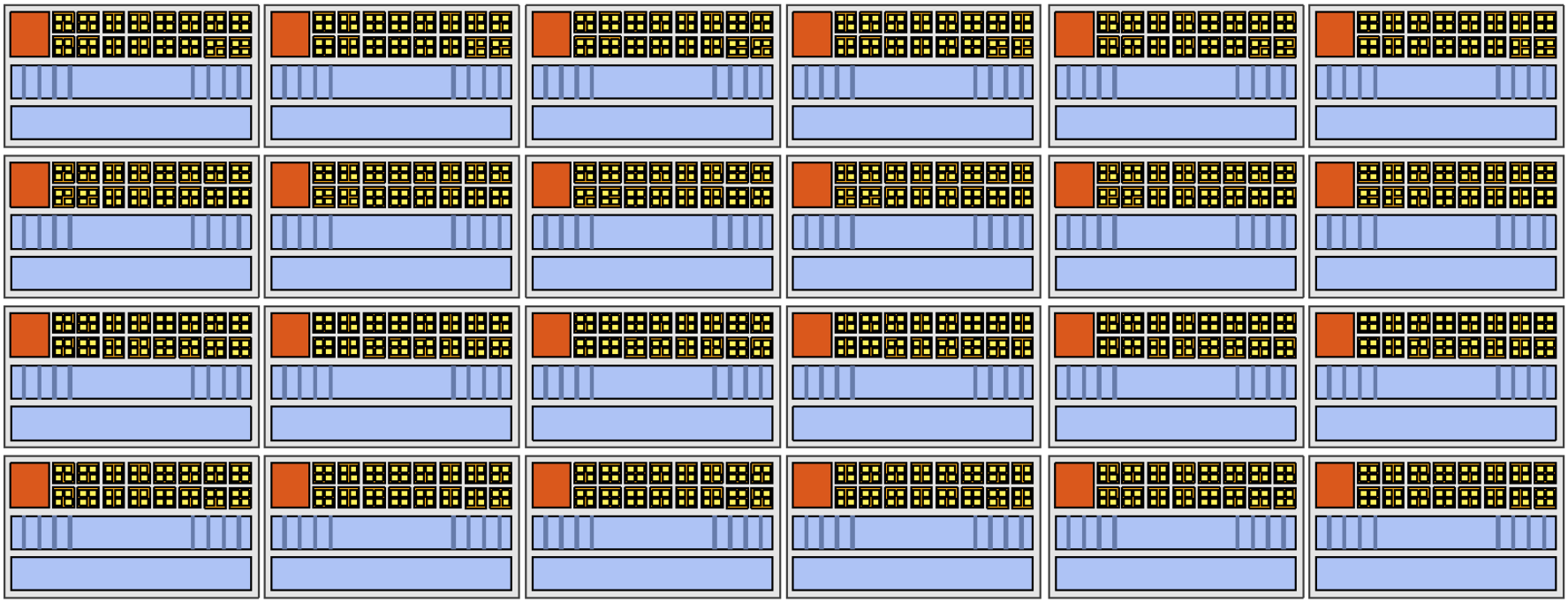




- The single SIMD lane has a VLIW architecture
- VLIW parallelism extracted by the compiler

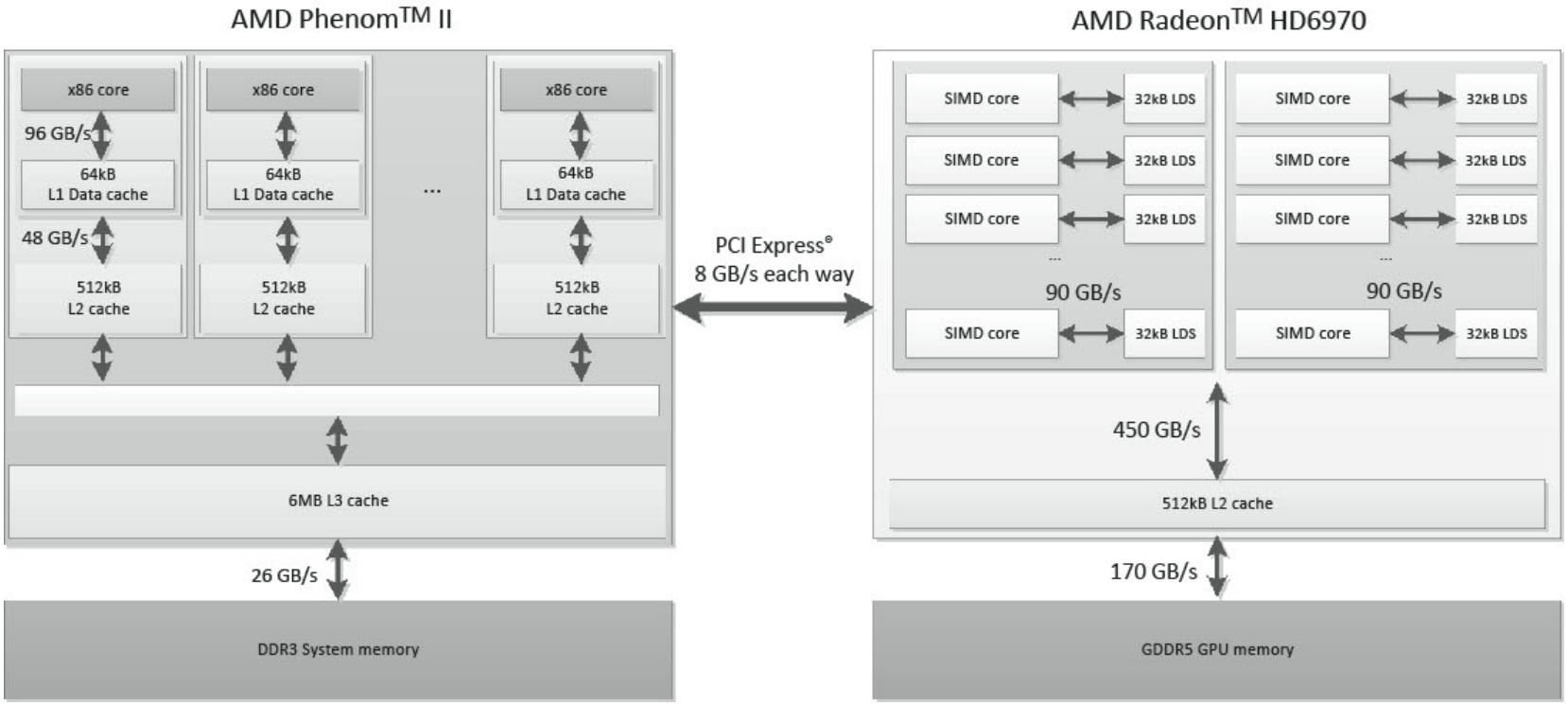
AMD Radeon HD 6970 Cayman (2010)

- There are 24 of these “cores” (compute units) on the 6970 able to handle up to ~32,000 threads!



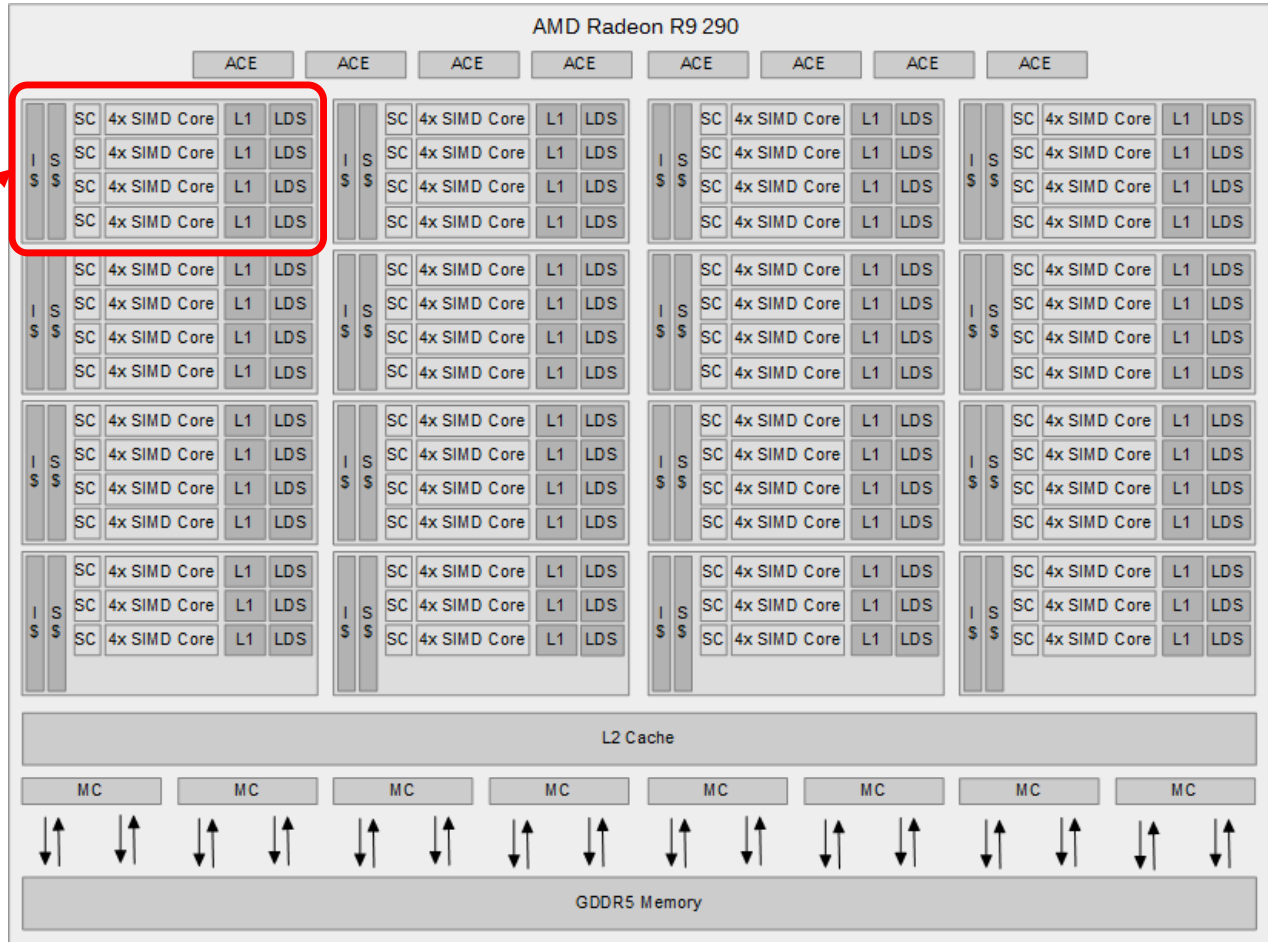
AMD Radeon HD 6970 Cayman (2010)

- Memory hierarchy and communication infrastructure



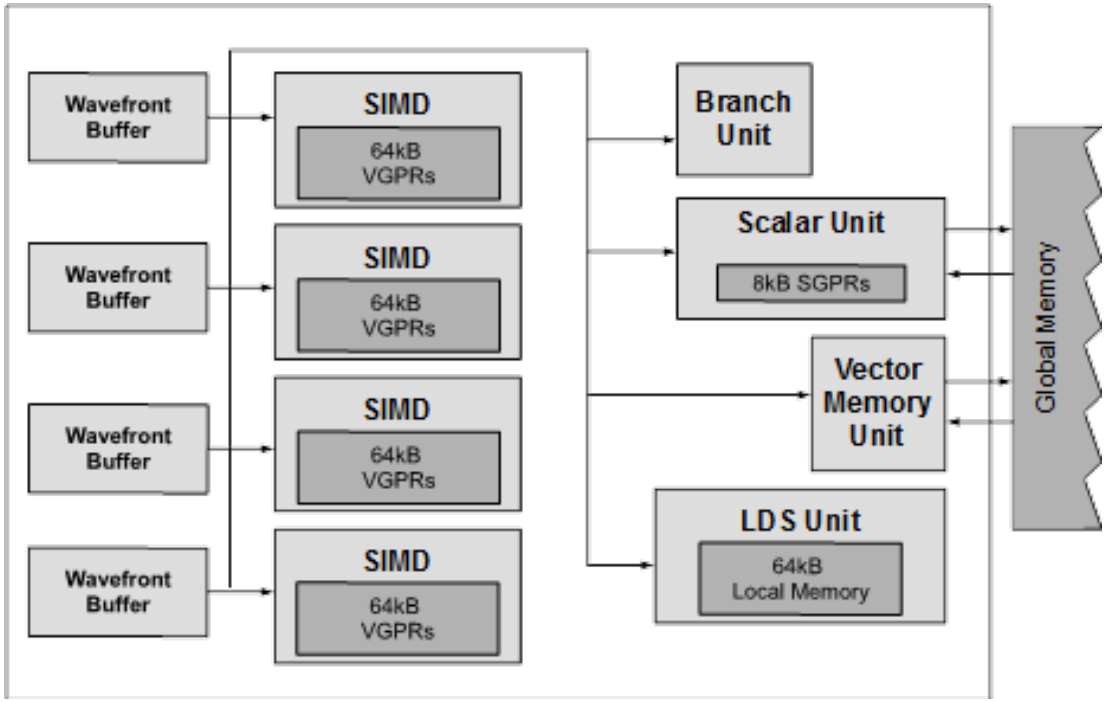
AMD Radeon R9 290X (2013)

- The overall multicore architecture
  - Up to 44 compute units



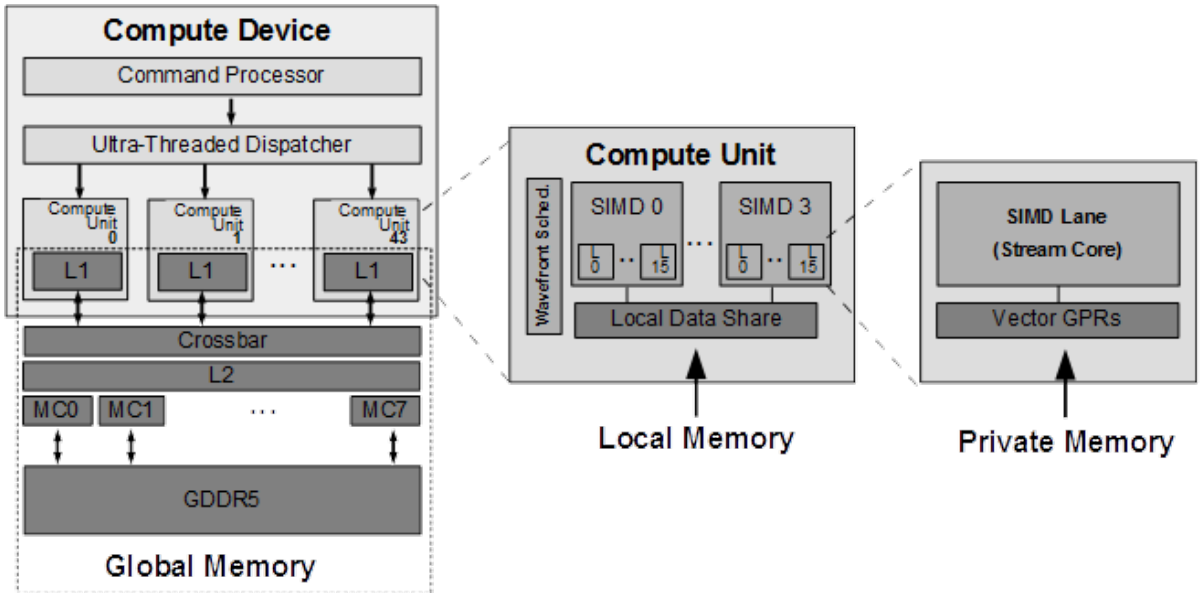
AMD Radeon R9 290X (2013)

- The compute unit
  - VLIW architecture replaced with a SIMD 16x vector architecture
- Similar workflow of the NVIDIA counterpart
  - SIMD execution
  - Wavefront interleaving
  - …



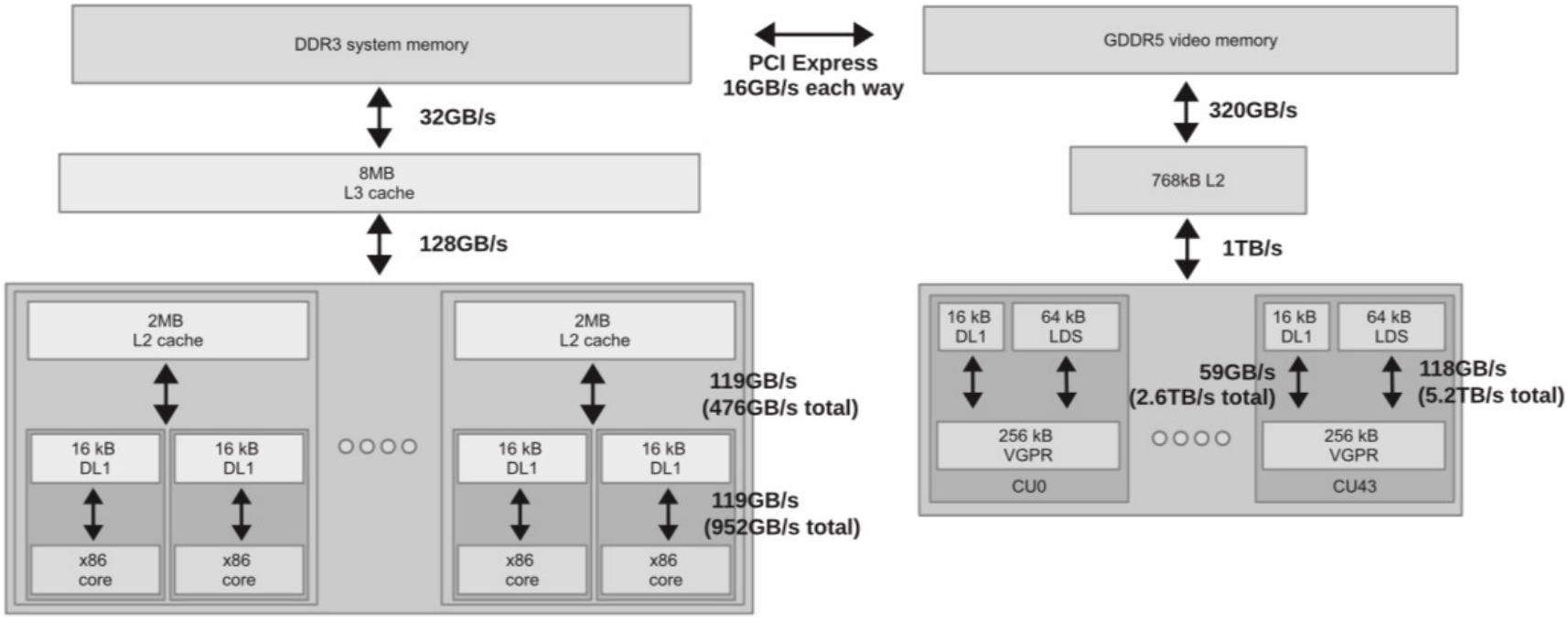
AMD Radeon R9 290X (2013)

- R/W L2 cache
  - Write back policy
- R/W L1 cache
  - Write through policy
- Local Data Share (LDS)
  - Contains integer atomic units



AMD Radeon R9 290X (2013)

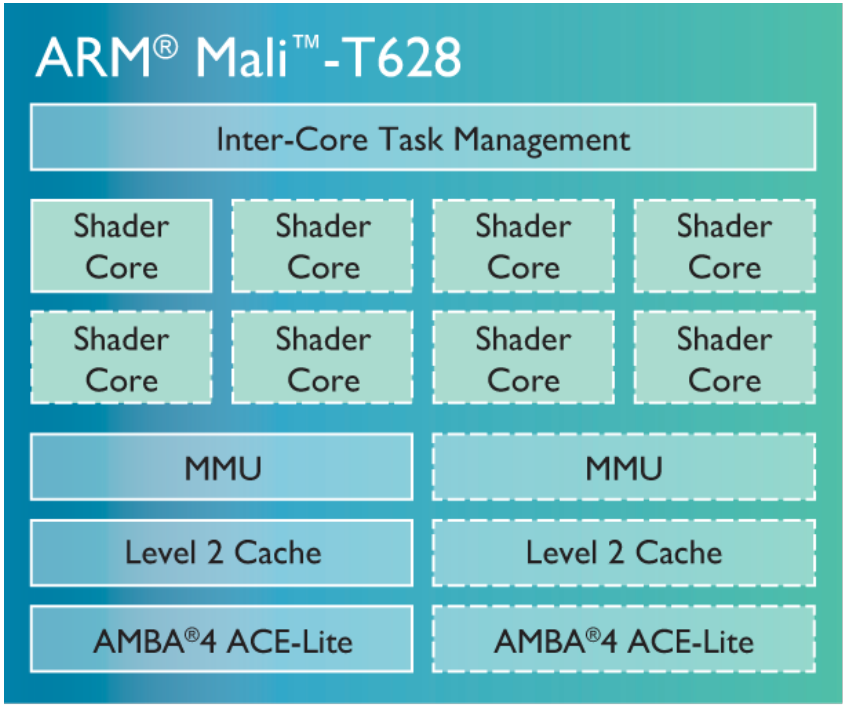
- Memories and interconnections



ARM Mali 628 (2014)

- Targeted for embedded computing

- Considerably smaller architecture



ARM Mali G720 (2023)

- Targeted for embedded computing

- Considerably smaller architecture