



UNIVERSITY OF SCIENCES AND TECHNOLOGIES
HOUARI BOUMEDIENNE

COMPUTER SCIENCE FACULTY

MASTER 2 HIGH PERFORMANCE COMPUTING (HPC)

**Tesla Architecture:
NVIDIA's entry into high-performance
computing and general-purpose GPU
(GPGPU) computing**

Students :

ZEMOUCHI NAFILA RAIHANA
KHENE SORAYA

Tutor :

Mr. SAADI HOCINE

30 novembre 2024

Table des matières

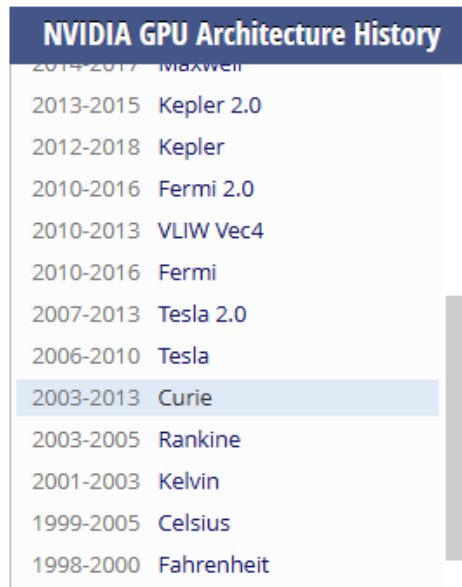
1	Introduction	3
1.1	OpenGL	3
1.2	Unified-Processor Design (Unified Shader Model)	3
1.3	Shader	4
2	The history of graphics and GPUs	4
2.1	Early 90 (Pre GPUs)	4
2.2	First era : Fixed function pipeline	5
2.2.1	Pipeline view	6
2.3	Second Era : Programmable Pipeline (Shader Programs) : 2000-2003	8
2.3.1	Pipeline's Overview of the Second Era	9
2.3.2	Architecture's Overview of the Second Era	9
2.4	Third era : Fully programmable GPU -The birth of Tesla-	11
3	Tesla Architecture	12
3.1	Overview of the Tesla architecture	12
3.2	Streaming Multiprocessors multithreaded	14
3.3	Streaming Processor	15
3.4	Interconnection Network	18
3.5	Memory Partitions and Caching	19
3.6	Atomic Operations	19
3.7	Compute models	19
3.8	Memory access instructions	19
4	The Scalability of Parallel Computing in Tesla Architecture	21
5	GPU examples	22
6	Conclusion	23
7	Bibliography	24

Table des figures

1	NVIDIA GPU Architecture History [5]	3
2	graphic rendering pipeline	4
3	graphics processing with CPU	5
4	gforce256 chip	5
5	A fixed-function NVIDIA GeForce graphics pipeline	6
6	Minimum Graphic Rendering Pipeline	7
7	Triangles Boat Example	8
8	GForce3 chip	8
9	Semi-Programmable Graphics Pipeline	9
10	GForce 6800 chip	9
11	GForce 6800 architecture	10
12	The problem with distinct shaders	10
13	Unified GPU architecture	11
14	GeForce 8800 architecture [2]	12
15	Texture/ processor Cluster architecture	13
16	Streaming multiprocessor withing TCP	14
17	Streaming multiprocessor execution pipeline	15
18	Execution units of a multiprocessor	15
19	Memory Hierachy	20

1 Introduction

Before the introduction of the Tesla architecture and CUDA, GPUs were primarily designed for **fixed-function graphics processing**. They were not programmable in the way we understand today. The CURIE architecture, the predecessor of Tesla, was mainly focused on gaming and graphics rendering..

A screenshot of a webpage titled "NVIDIA GPU Architecture History". The table lists various GPU architectures with their corresponding time periods. The "Curie" architecture (2003-2013) is highlighted with a blue background. The table has a dark blue header with white text.

NVIDIA GPU Architecture History	
2014-2017	Maxwell
2013-2015	Kepler 2.0
2012-2018	Kepler
2010-2016	Fermi 2.0
2010-2013	VLIW Vec4
2010-2016	Fermi
2007-2013	Tesla 2.0
2006-2010	Tesla
2003-2013	Curie
2003-2005	Rankine
2001-2003	Kelvin
1999-2005	Celsius
1998-2000	Fahrenheit

FIGURE 1 – NVIDIA GPU Architecture History [5]

Definitions

1.1 OpenGL

OpenGL (Open Graphics Library) is an API (Application Programming Interface) used to interact with the GPU for rendering 2D and 3D graphics. It provides a standardized way to access GPU functionality across different platforms and hardware, enabling developers to perform a variety of graphical tasks such as rendering 2D images and creating complex 3D models and effects.

1.2 Unified-Processor Design (Unified Shader Model)

The **Unified Processor Design**, also known as the **Unified Shader Model**, is an architecture where all computational cores in a GPU are general-purpose and can execute any type of task. Unlike older GPU architectures with dedicated hardware blocks for specific tasks (such as vertex processing or pixel shading), the unified model uses a pool of identical processing units that can dynamically switch between different tasks depending on workload demands. For example, the Tesla architecture employs this design, where the same set of cores can be allocated to vertex, geometry, and pixel shading tasks as needed.

1.3 Shader

A **shader** is a small program that runs on the GPU and performs graphics-related tasks. Shaders are executed in parallel by the GPU's many cores. There are various types of shaders, each serving specific roles in the graphics pipeline :

- **Vertex Shader** : This shader processes each vertex of a 3D model, transforming its 3D coordinates into 2D screen space. It handles tasks like transformations, lighting calculations, and assigning texture coordinates.
- **Pixel (Fragment) Shader** : This shader works on individual fragments (potential pixels) generated during rasterization. It calculates the final color of each pixel based on the data from the vertex shader, including texture information, lighting, and other attributes.

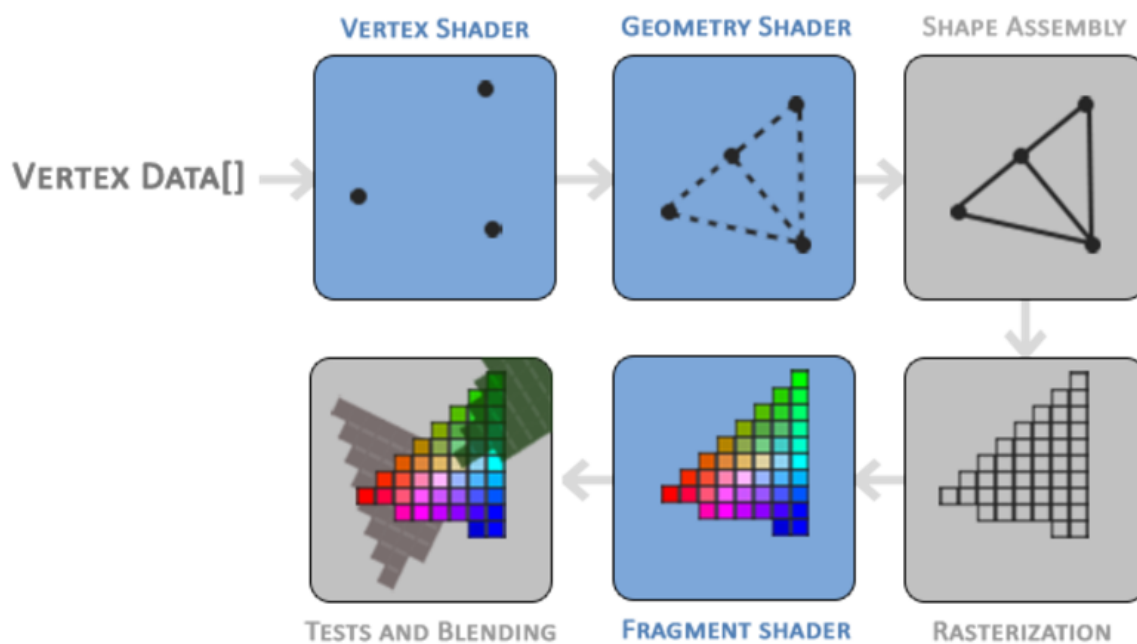


FIGURE 2 – graphic rendering pipeline

2 The history of graphics and GPUs

2.1 Early 90 (Pre GPUs)

In the first phase of computer graphics, from the early 1980s to the late 1990s, graphics were primarily handled by CPUs. During this time, graphical capabilities were minimal, with visuals limited to basic 2D representations and simple 3D models. Graphics were often non-realistic and stylized due to the limited processing power of CPUs and the lack of specialized hardware for graphics rendering.

This period saw the development of the first personal computers capable of displaying graphics, though they were primarily used for tasks like simple gaming or business applications. As CPU power was relatively low compared to modern standards, graphics

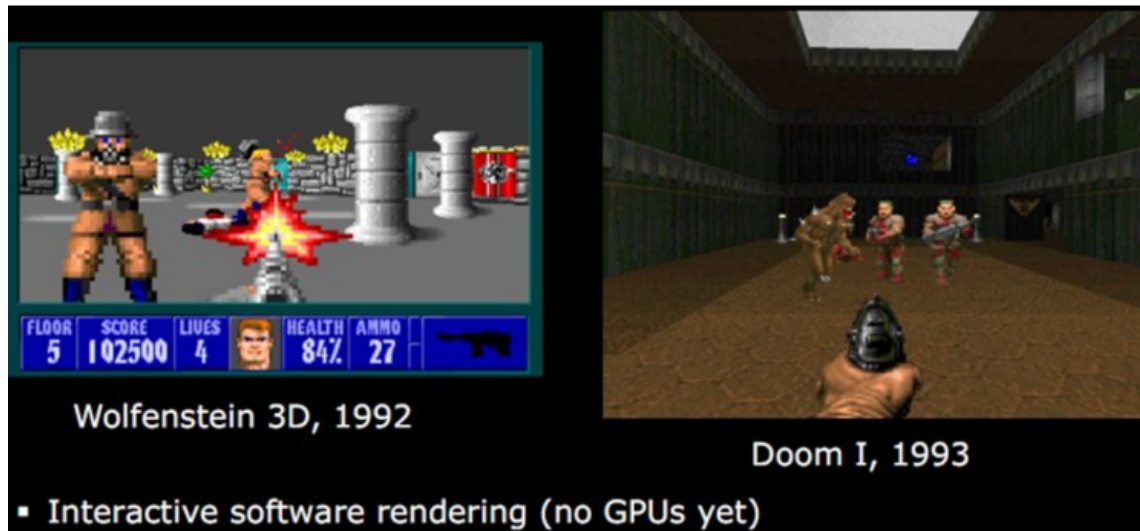


FIGURE 3 – graphics processing with CPU

were often blocky, with rudimentary shapes and colors. For instance, early 3D games like Wolfenstein 3D and Doom (1990s) employed very basic graphics due to the limitations of both hardware and software at the time.

Additionally, the use of software rendering allowed for some advancements, but it wasn't until the late 1990s that dedicated graphics processing units (GPUs) began to emerge, marking a significant shift towards more realistic and complex graphics. Graphics accelerated by GPUs, such as those from companies like NVIDIA, began to push the boundaries of what was possible, leading to more sophisticated visual effects in games and simulations.

2.2 First era : Fixed function pipeline

The term GPU (Graphics Processing Unit) was introduced by NVIDIA in 1999 for the GeForce 256 (Celsius architecture), to denote a single-chip processor with integrated transform, lighting, triangle manipulation, and rendering engines, able to process a minimum of 10 million polygons per second. It supports OpenGL and Direct3D APIs.



FIGURE 4 – gforce256 chip

2.2.1 Pipeline view

The first era of GPUs is called the fixed-function era because the graphics pipeline was hardwired to perform a predefined sequence of tasks. Developers had limited flexibility, as the hardware was designed to execute specific operations. These tasks were implemented as fixed hardware circuits, meaning developers couldn't customize or program them. They could only adjust parameters (e.g., light intensity or texture coordinates), not the underlying algorithms.

While developers could use OpenGL/Direct3D to configure graphics effects (e.g., set light positions or apply textures), they were constrained by what the fixed-function hardware could do. The surface of an object is drawn as a collection of triangles. The finer the size of the triangle, the better the image quality.

In the early NVIDIA GeForce GPUs, the fixed-function graphics pipeline performs a fixed task at each stage, with tasks parameterized by the user.

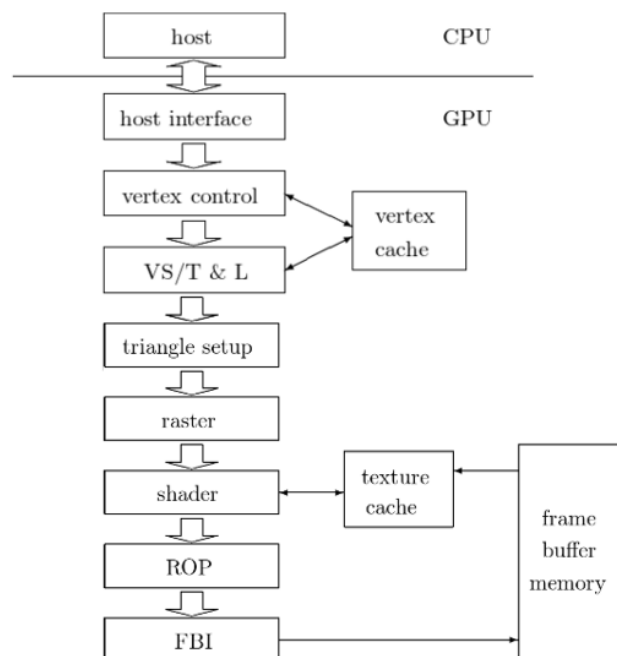


FIGURE 5 – A fixed-function NVIDIA GeForce graphics pipeline

1. The host interface receives graphics commands and data from the CPU. The commands are typically given by application programs by calling an API function. Examples of such APIs are DirectX and OpenGL.
2. The vertex control stage receives parameterized triangle data from the CPU. It converts the triangle data into a format that the GPU can process efficiently and places the prepared data into the vertex cache (data to be stored and modified for the next stages).
3. The vertex shading, transform, and lighting (VS/T and L) stage transforms vertices and assigns per-vertex values (e.g., colors, normals, texture coordinates, tangents, etc.). These operations are highly compute-intensive, with a set of very specific mathematical instructions performed billions of times per second to render a scene.

4. The triangle setup stage is responsible for preparing mathematical equations that allow the GPU to efficiently handle rendering tasks, outline triangles, and set up interpolation for attributes like colors, textures, and lighting across the triangle's surface.
5. The raster stage determines which pixels are contained within each triangle.
6. The shader stage determines the final color of each pixel. This can be generated as a combined effect of many techniques : interpolation of vertex colors, texture mapping, per-pixel lighting calculations, reflections, and more. Many effects that make the rendered images more realistic are incorporated in the shader stage.
7. The ROP (raster operation) stage performs the final raster operations on the pixels. It handles color raster operations that blend the color of overlapping/adjacent objects for transparency and anti-aliasing effects. It also determines the visible objects from a given viewpoint and discards the occluded pixels. A pixel becomes occluded when it is blocked by pixels from other objects according to the given viewpoint.
8. The frame buffer interface (FBI) stage manages memory reads from and writes to the display frame buffer memory.

The graphics card pipeline includes the three most important stages : the vertex shader, the geometry shader, and the rasterizer/pixel shader (the minimum required with a GPU to render a scene).

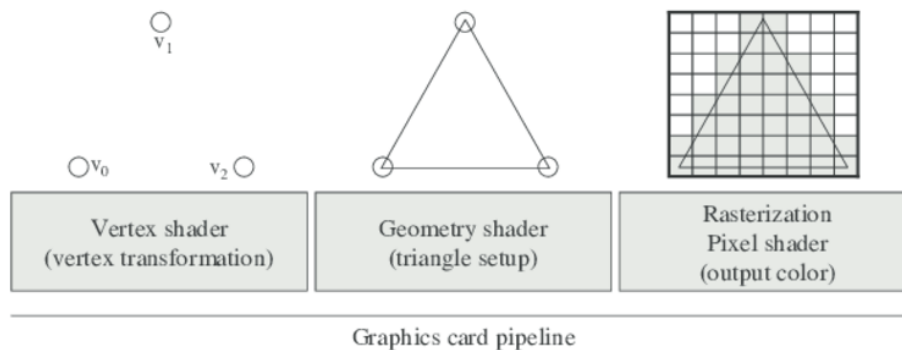


FIGURE 6 – Minimum Graphic Rendering Pipeline

We process the pipeline for millions of triangles (each triangle passes through that pipeline to generate the final scene).

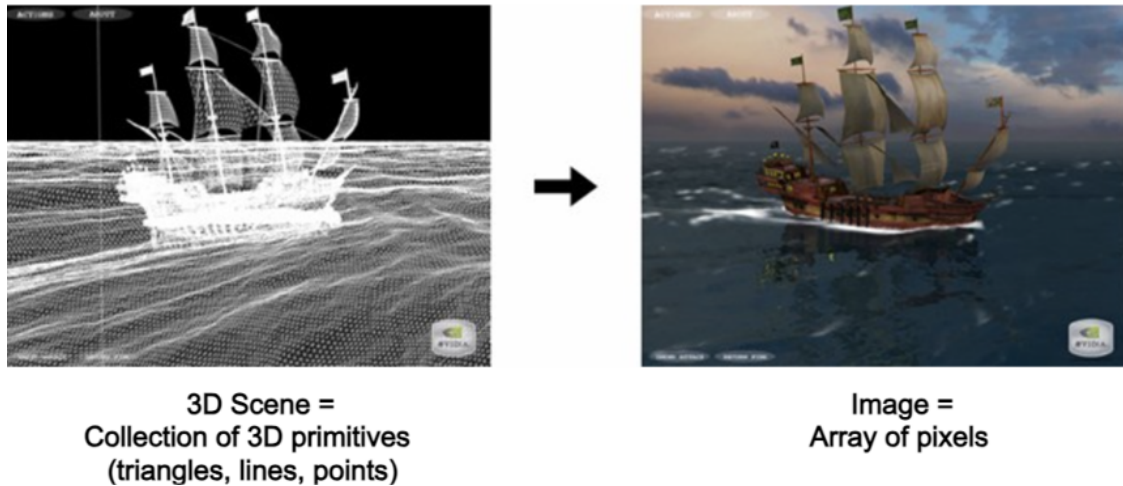


FIGURE 7 – Triangles Boat Example

2.3 Second Era : Programmable Pipeline (Shader Programs) : 2000-2003

In 2001, NVIDIA released the GeForce 3, built on the Kelvin architecture, which gave programmers the ability to program some parts of the previously non-programmable pipeline. Instead of sending all the graphics description data to the GPU and having it simply flow through the fixed pipeline, the programmer can now send this data along with vertex "programs" (called shaders) that operate on the data while in the pipeline.



FIGURE 8 – GeForce3 chip

It features 4 pixel shaders, 1 vertex shader, 8 texture mapping units, and 4 ROPs.

2.3.1 Pipeline's Overview of the Second Era

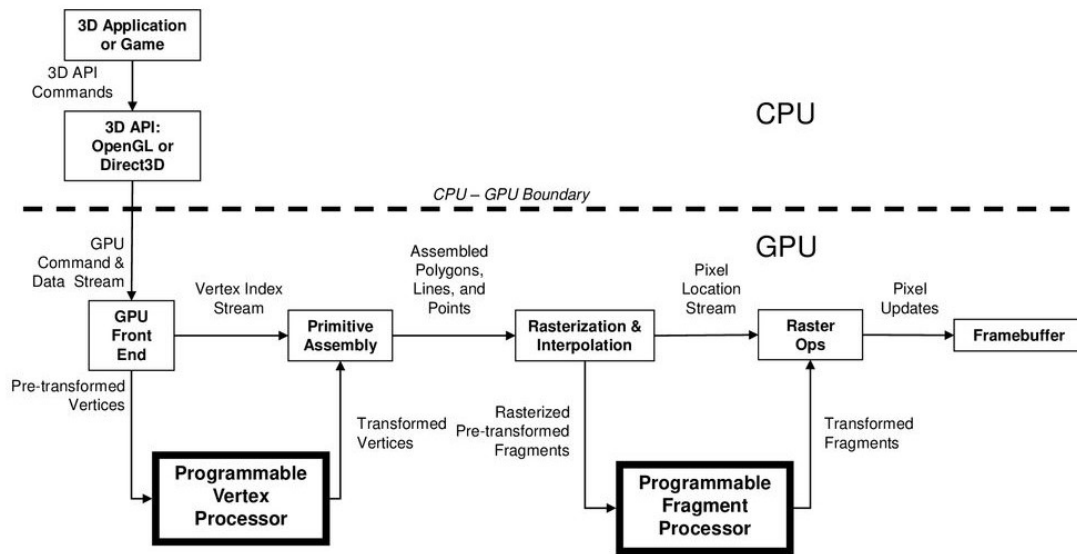


FIGURE 9 – Semi-Programmable Graphics Pipeline

Between the programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks. For example, the rasterizer performs rasterization and interpolation. It is a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries.

2.3.2 Architecture's Overview of the Second Era

A tour of the GeForce 6800, a specific model within the GeForce 6 series, built on the NV40, Curie architecture.



FIGURE 10 – GForce 6800 chip

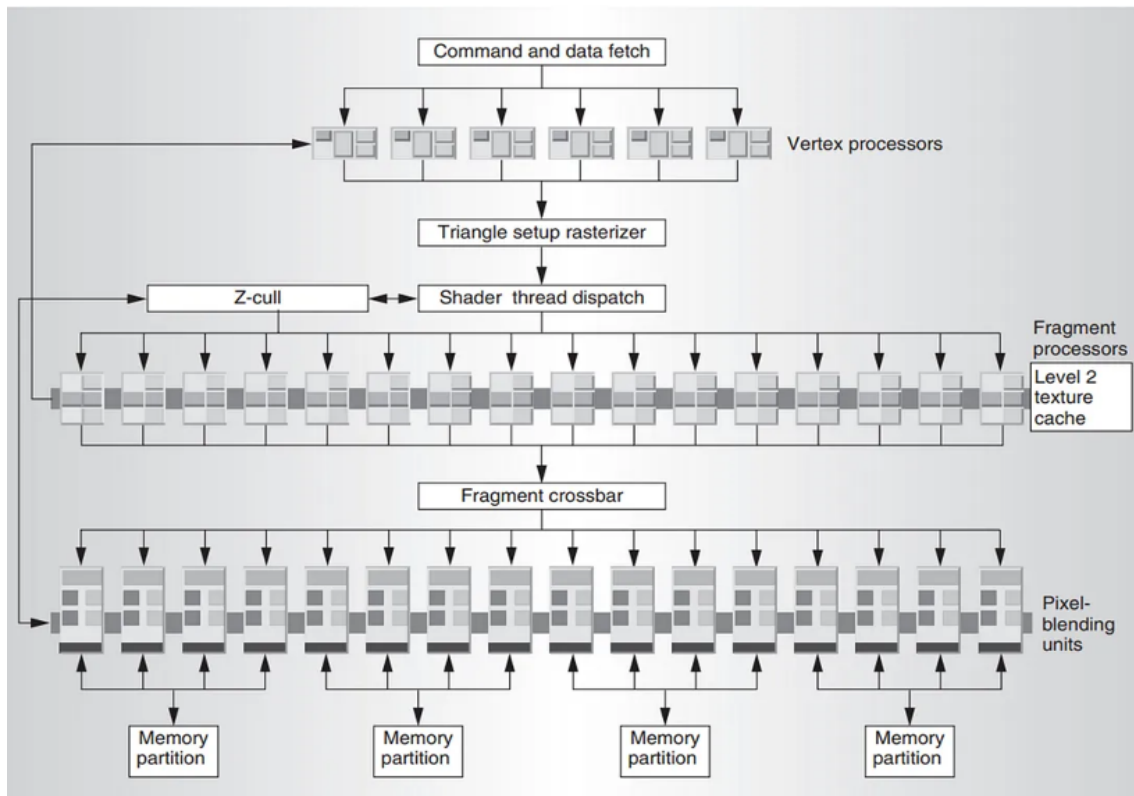


FIGURE 11 – GeForce 6800 architecture

It features 16 pixel shaders, 6 vertex shaders, 16 texture mapping units, and 16 ROPs.

Problem :

- Frames with many edges (vertices) require more vertex shaders.
- Frames with many primitives require more pixel shaders.
- Unequal task distribution leads to inefficient hardware usage.

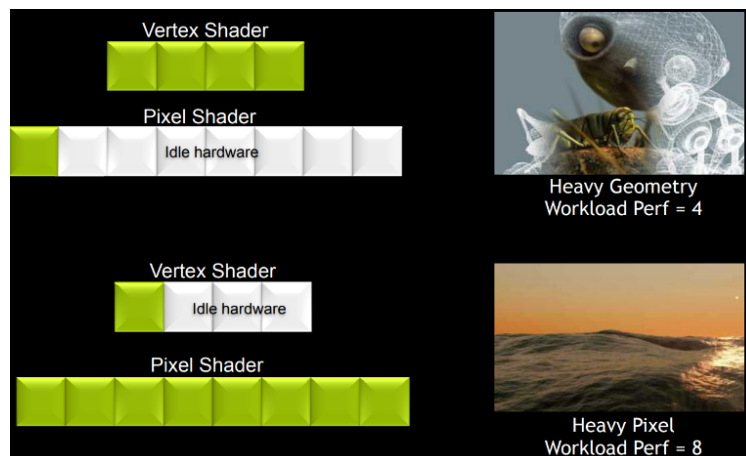


FIGURE 12 – The problem with distinct shaders

Solution : A unified shader : geometry shaders, pixel shaders, and vertex shaders run on the same core. This limits idle shader cores. The programmer determines the type of shader : Vertex, Pixel, and Geometry.

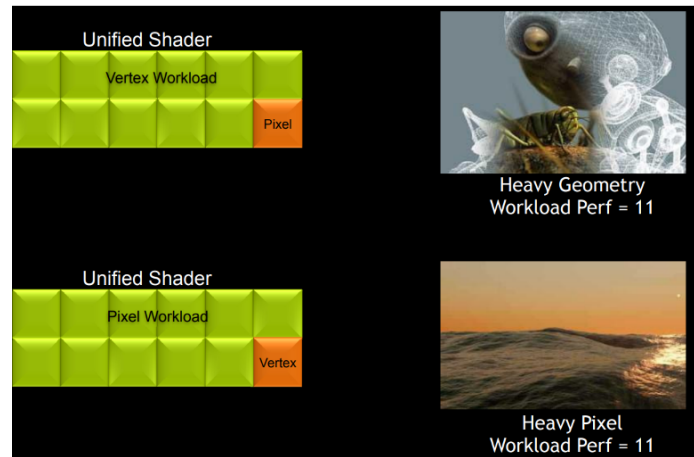


FIGURE 13 – Unified GPU architecture

Thanks to the Unified GPU architecture, all of these problems are eliminated because each SP is capable of processing Vertex, Pixel, and Geometry. This means that the graphics processor is more utilized, regardless of whether the scenes are as heavy in Vertex or Pixel. Vertex, Pixel (and later Geometry) operations, GPU propulsion, and control logic are dynamically assignable to any available SP.

2.4 Third era : Fully programmable GPU -The birth of Tesla-

The third era of GPUs, marked by the introduction of NVIDIA's Tesla architecture in November 2006 with the GeForce 8800, revolutionized graphics and computational processing. Tesla unified and extended the capabilities of vertex and pixel processors, enabling a shift toward high-performance parallel computing. This was facilitated by the Compute Unified Device Architecture (CUDA), which allowed developers to write parallel programs using the C programming language. The emergence of CUDA enabled GPUs to handle not only graphics tasks but also general-purpose computing, giving rise to General-Purpose GPU (GPGPU) applications. With Tesla, GPUs became powerful tools for a wide range of computationally demanding tasks beyond traditional graphics rendering.

3 Tesla Architecture

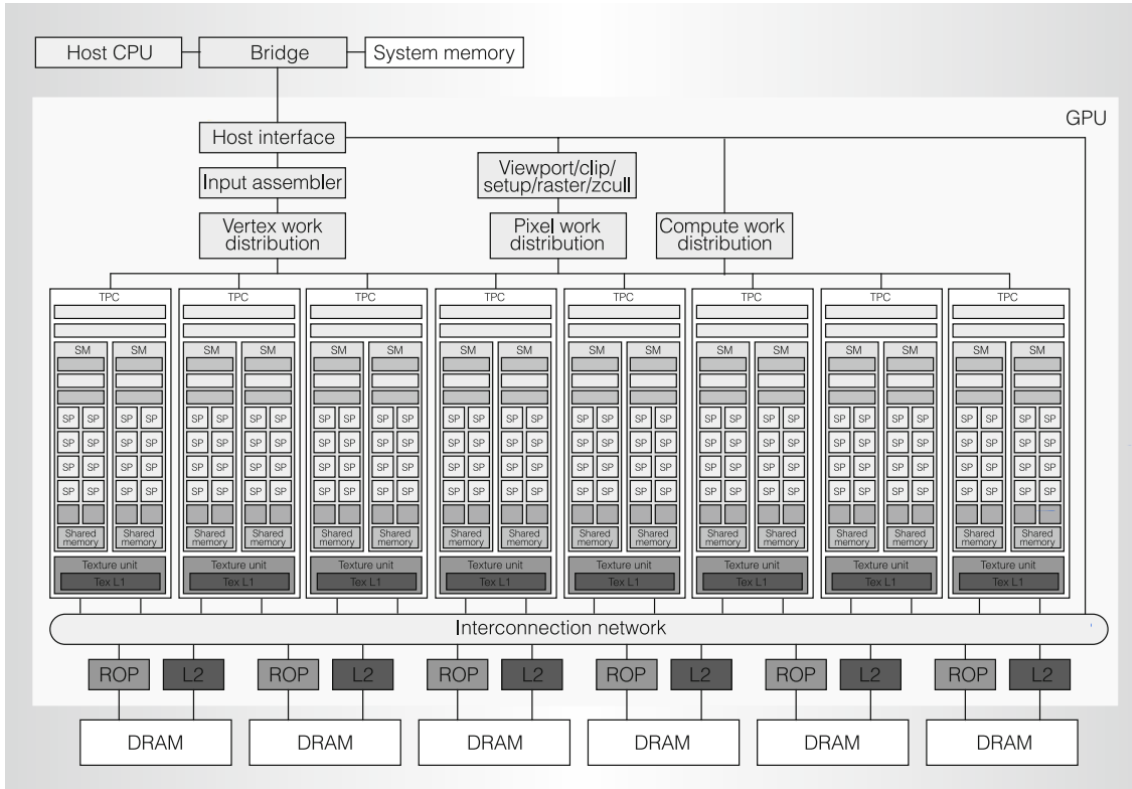


FIGURE 14 – GeForce 8800 architecture [2]

3.1 Overview of the Tesla architecture

The Tesla architecture departs from the traditional graphics pipeline, adopting a scalable processor array that enables parallel computations for both graphics and general-purpose computing. At its heart, the scalable streaming processor array (SPA) performs all programmable GPU calculations, offering flexibility beyond graphics rendering. The architecture also features a scalable memory system with external DRAM control and fixed-function raster operation processors (ROPs) for efficient color and depth frame buffer operations. This design facilitates high performance in both graphical and non-graphical applications, marking the evolution towards General-Purpose GPU (GPGPU) computing.

1. **Host Interface** : Communicates with the host CPU, responds to commands from the CPU, fetches data from system memory, checks command consistency, and performs context switching.
2. **Input Assembler** : Collects geometric primitives (points, lines, triangles, line strips, and triangle strips) and fetches associated vertex input attribute data.
3. **Work Distribution Unit** : Forwards the input assembler's output stream to the array of processors, which execute vertex, geometry, and pixel shader programs, as well as computing programs. The vertex and compute work distribution units deliver work to processors in a round-robin scheme. Pixel work distribution is based on the pixel location.

4. **Streaming Processor Array** : Executes graphics shader thread programs and GPU computing programs and provides thread control and management. The number of TPCs determines a GPU's programmable processing performance and scales from one TPC in a small GPU to eight or more TPCs in high-performance GPUs.
5. **Texture/Processor Cluster (TPC)** : TPCs receive work from different work distribution

Work Distribution	Input	TPC Execution Type
Vertex WD	vertex work packets	Vertex shader programs, and (if enabled) geometry shader programs.
Pixel WD	pixel fragments	Pixel-fragment processing.
Compute WD	compute thread arrays (CTA)	Processes work for multiple logical streams simultaneously.

TABLE 1 – Work Distribution and TPC Execution Types

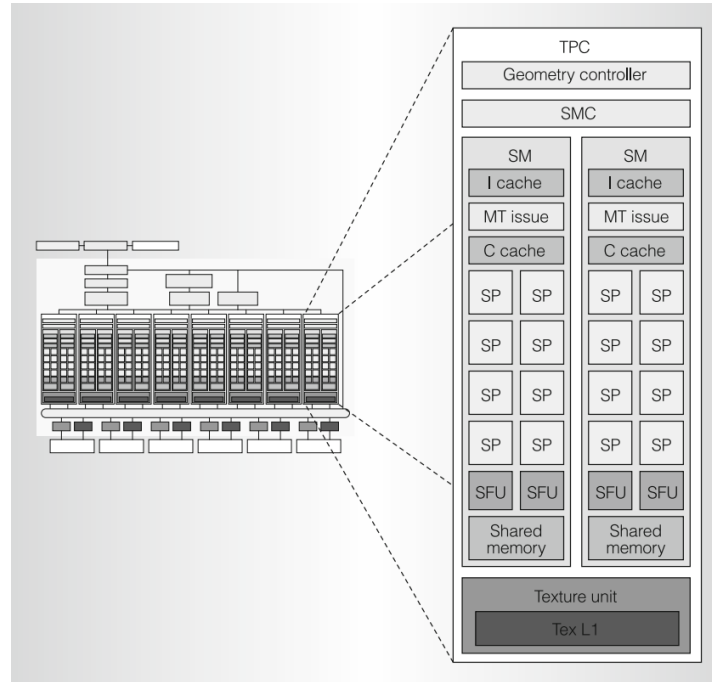


FIGURE 15 – Texture/ processor Cluster architecture

[4] Each TCP contains :

- **Geometry Controller** : This unit manages the flow of primitive data and vertex attributes to the physical Streaming Multiprocessors (SMs). It ensures that data is handled in the right sequence and is sent to the correct units for further processing.
- **SM Controller (SMC)** : The SMC controls multiple SMs by arbitrating shared resources such as the texture unit, load/store paths, and I/O paths. It

supports three graphics workloads—vertex, geometry, and pixel—and is responsible for balancing the load among these workloads. Each input type has independent I/O paths.

- **Texture Unit** : The texture unit processes groups of four threads (vertex, geometry, pixel, or compute threads) per cycle. It generates filtered samples (typically RGBA) based on texture coordinates. Each texture unit contains four texture address generators and eight filter units. Texture memory is typically used to read from a 2D location and apply filtering if necessary.
- **Streaming Multiprocessor (SM)** : The SM is a unified processor capable of handling both graphics and computing tasks. It executes vertex, geometry, and pixel shader programs, as well as parallel computing programs.

[3]

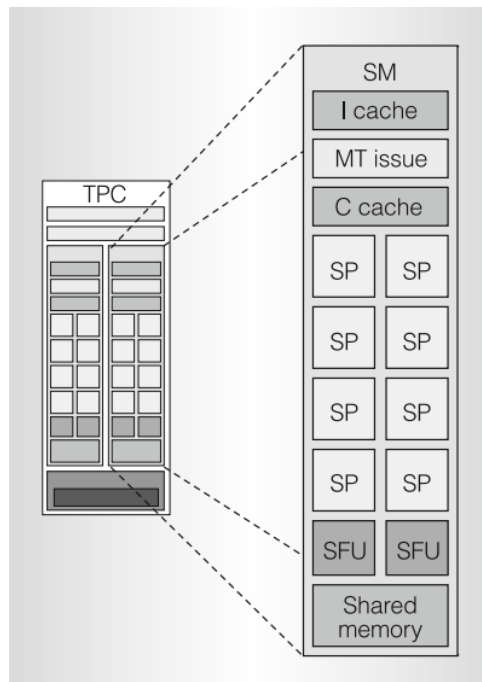


FIGURE 16 – Streaming multiprocessor withing TCP

3.2 Streaming Multiprocessors multithreaded

[1]

- **Parallel Execution** : The SM can handle up to 768 threads concurrently, executing different programs like vertex shaders, pixel shaders, or compute tasks with zero scheduling overhead, resulting in faster thread management.
- **Independent Threads** : Each thread has its own execution state and can follow an independent code path. This allows the SM to handle various workloads in parallel without interdependencies.
- **Synchronization** : Threads can synchronize at barriers using a single SM instruction. This ensures that all threads reach a specific point before any of them proceed, maintaining correct execution order, particularly in fine-grained parallelism tasks.

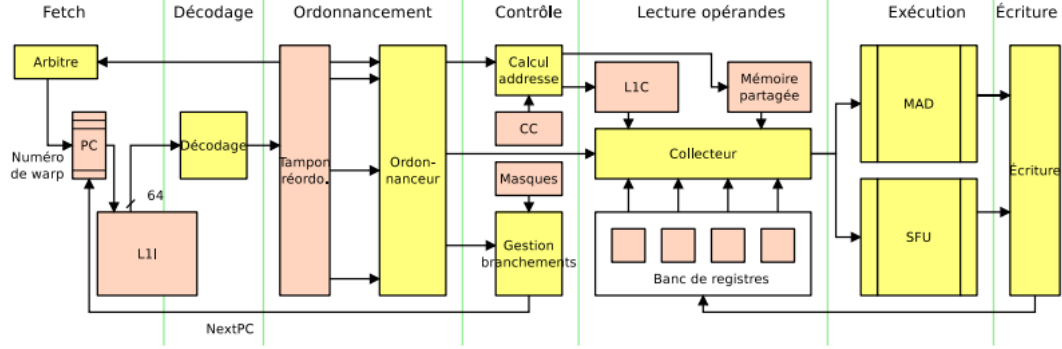


FIGURE 17 – Streaming multiprocessor execution pipeline

- **Fast Barrier Synchronization** : Barriers ensure that all threads reach a synchronization point before continuing. This prevents out-of-order execution, which could lead to incorrect results in parallel computing.

3.3 Streaming Processor

Each SM contains multiple **Streaming Processors (SPs)** responsible for executing the threads of a given workload. SMs are designed to efficiently manage large-scale parallel computations, making them ideal for applications in machine learning, simulations, and rendering.

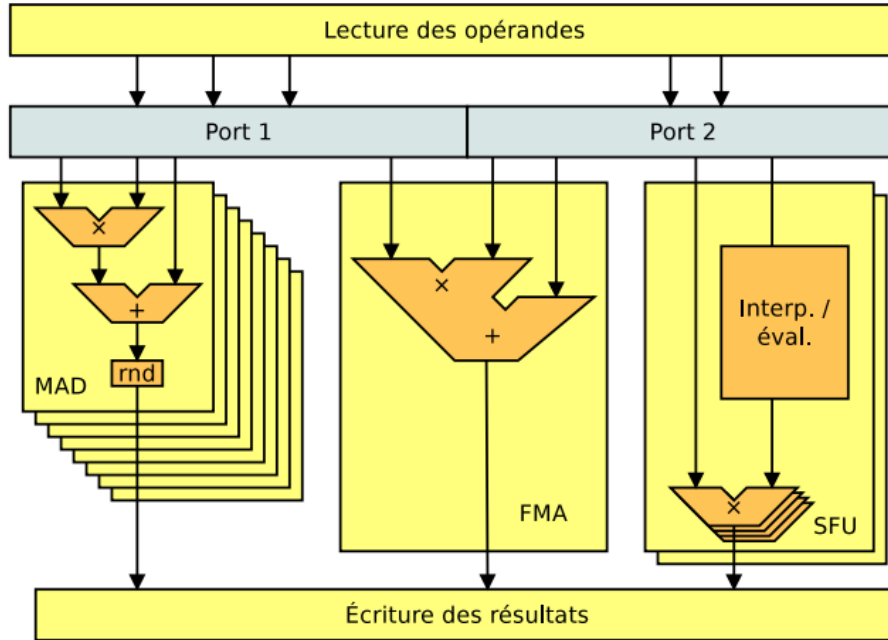


FIGURE 18 – Execution units of a multiprocessor

MAD (Multiply-Add)

Definition : MAD stands for *Multiply-Add*, a critical operation in GPU streaming processors. It combines multiplication and addition into a single instruction. For example, it computes :

$$\text{Result} = (A \times B) + C$$

Efficiency : By fusing multiplication and addition, MAD minimizes latency and increases throughput compared to performing the operations separately.

Usage : Common applications include :

- Vector computations.
- Linear algebra operations.
- Graphics transformations.
- Physics-based simulations.

FMA (Fused Multiply-Add)

Definition : FMA stands for *Fused Multiply-Add*, an enhanced version of MAD. It performs the multiply and add operations in a single step, with no intermediate rounding :

$$\text{Result} = (A \times B) + C$$

Differences Between MAD and FMA :

- **Precision :** FMA offers higher precision by eliminating rounding errors between the multiplication and addition steps.
- **Performance :** While both are similar in throughput, FMA is more accurate for applications requiring precise numerical computations, such as scientific simulations.
- **Implementation :** FMA is widely supported in modern GPUs due to its advantages for computational tasks.

SFU (Special Function Unit)

Definition : SFU stands for *Special Function Unit*, specialized hardware in streaming processors designed to execute complex mathematical functions that are computationally expensive to calculate using standard cores.

Functions Performed :

- Trigonometric functions : \sin , \cos , \tan .
- Exponentials : e^x .
- Logarithms : $\log(x)$.
- Square roots : \sqrt{x} .

Role : SFUs accelerate these computations using hardware approximations or lookup tables for high speed.

Usage in Graphics : SFUs are critical for tasks such as :

- Lighting calculations.
- Texture mapping.
- Geometric transformations.

GPU Scheduler Overview

A GPU **scheduler** (*ordonnanceur*) is responsible for distributing thread blocks (groups of threads guaranteed to execute on the same multiprocessor) across the streaming multiprocessors (SMs). Before signaling the start of execution to an SM, the scheduler performs initialization tasks, including :

- Setting kernel arguments.
- Initializing block coordinates and dimensions (block size and grid size).
- Defining thread coordinates within the block.

Shared memory is utilized for common data such as block coordinates and kernel arguments since these are shared by all threads within a block. For example :

- The first 16 bytes of shared memory store eight 16-bit integers representing dimensional information.
- Kernel arguments are stored in the subsequent memory locations.

Although some of this data (e.g., constants) could be placed in constant memory, shared memory is often chosen to avoid the overhead of initializing constant memory. Each thread's unique identifier is written in the architectural register R_0 as a bitfield, as illustrated in the relevant format diagram.

Scheduling Policy

Tests reveal the following characteristics of the scheduling mechanism :

- Block scheduling follows a **round-robin policy**.
- The global scheduler waits for all currently executing blocks to finish before assigning the next group of blocks.
- There is no pipelined execution of thread blocks.

Power Efficiency

When a Thread Processing Cluster (TPC) has no tasks to execute, its clock counter stops incrementing. This behavior suggests the implementation of **coarse-grained clock gating**, aimed at minimizing power consumption.

Single-Instruction, Multiple-Thread (SIMT) Architecture :

SIMT is a specialized processing model used in Tesla SMs to efficiently manage multiple threads executing different programs. The architecture is designed for high throughput across many threads while simultaneously executing various types of programs.

- **Warp and Thread Management** : Threads are organized into groups known as "warps," with each warp consisting of 32 threads. The threads in a warp execute the same instruction in parallel, but they can diverge if a branch occurs. Divergent threads are executed serially until they reconverge, allowing for flexible thread execution despite branching.
- **Efficiency** : SIMT performs best when all threads in a warp follow the same execution path. If threads diverge due to conditional branches, the system handles the divergence by executing each possible branch path serially, then reconverging them once all paths have been processed.

- **Warp Scheduling** : The SM schedules and executes multiple warps concurrently, prioritizing them based on factors such as warp type (vertex, pixel, compute) and instruction type. This ensures fair execution and optimal performance across different tasks.

SIMT in Practice :

- **Shader Programs** : The SM executes vertex, geometry, pixel, and compute programs in parallel. Warps may execute different types of programs concurrently, making the architecture versatile for complex workloads.
- **Instruction Fetch and Execution** : The SM fetches and issues instructions across all threads in a warp, ensuring performance is maximized when all threads are active and working together.

SM Instructions and Architectures :

- **Instruction Set** : Tesla SM uses a vector-based instruction set that supports a range of operations including floating-point, integer, memory load/store, and texture operations.
- **Floating-Point and Integer Operations** : Common instructions include basic arithmetic (addition, multiplication), as well as transcendental functions like sine, cosine, and logarithms.
- **Efficient Attribute Interpolation** : Special instructions in the instruction set allow for efficient generation of pixel attributes, which are crucial for rendering and graphics processing tasks.

This instruction set shares many similarities with intermediate languages generated from the compilation of shaders in Direct3D (HLSL5 assembler), OpenGL (ARB Fragment/Vertex programs), or CUDA (PTX). In particular, it abstracts almost entirely the architectural SIMD (Single Instruction, Multiple Data) width. All instructions operate on vectors, including memory read/write instructions, which are transformed into gather/scatter operations, as well as comparison and branching instructions. These are executed in a manner resembling a MIMD (Multiple Instruction, Multiple Data) model, where computations in different SIMD lanes are independent and can be treated as scalar-like, even though the architecture is vector-based.

In essence, while the architecture is vectorized, it enables execution that feels like scalar or MIMD by managing independent computations across SIMD lanes. Despite this, the architecture doesn't offer scalar instructions, scalar registers, or even scalar memory operations; it is entirely vectorized. Ironically, this vectorization is what allows the architecture to be referred to as "scalar" by its designer.

3.4 Interconnection Network

The interconnection network linking the TPCs (Streaming Multiprocessors) to memory partitions in a GPU is a custom *crossbar circuit*, specifically designed for each GPU rather than a generic, scalable *ring bus* system. This architecture allows for efficient data transfer between the processing cores and memory, tailored to the GPU's specific needs, optimizing both computing and graphics performance.

3.5 Memory Partitions and Caching

Memory partitions are responsible for managing both *ascending* and *descending* memory traffic to and from the crossbar. They include, in particular, *second-level texture caches (L2)*. The role of these caches is similar to that of *first-level texture caches (L1)*, but with a broader scope. These caches serve to *merge concurrent memory access requests*, reducing the load on memory bandwidth and increasing memory access granularity. Specifically :

- The **L1 caches** handle concurrent accesses from the Streaming Multiprocessors (SMs) within the same TPC.
- The **L2 caches** merge accesses from L1 caches of each TPC, operating as a distributed cache system.

This setup minimizes memory access bottlenecks, increases throughput, and boosts memory bandwidth for both computing and graphics tasks.

3.6 Atomic Operations

The system also supports *atomic operations*, which are essential for operations that require synchronization, such as *parallel reductions* or fragment merging in graphics rendering. These atomic operations can be initiated either through *CUDA instructions* or graphic rendering processes. They are processed within the same memory unit, ensuring data consistency during parallel operations.

This combination of specialized memory management and atomic operations significantly improves the GPU's performance in handling complex computational tasks, including both general-purpose computing and high-performance graphics rendering.

3.7 Compute models

At the architectural level, the compute units of all Tesla GPUs recognize the same basic instruction set, but extensions are introduced or removed depending on the architecture revisions or the product line. These architecture revisions are referred to as the Compute Model by NVIDIA and are represented in the form of a version number. The first digit indicates the architecture generation, while the second digit specifies the revision number of the architecture. Each revision includes the features introduced by all previous revisions.

Compute Model	Added Features
1.0	The basic Tesla architecture
1.1	Atomic instructions in Global memory
1.2	Atomic instructions in shared memory, vote instructions
1.3	Double precision

TABLE 2 – Compute Models and Features in Tesla Architecture

3.8 Memory access instructions

In Tesla GPU architectures, memory access instructions support both graphical and computational tasks. The **texture** unit fetches and filters texture samples from memory,

while the ROP unit writes pixel-fragment outputs to memory. For computational tasks, Tesla SMs implement memory **load/store** instructions, allowing access to three primary memory spaces :

- **Local Memory** : Stores per-thread, private, temporary data, typically in external DRAM.
- **Shared Memory** : Provides low-latency access to data shared by threads within the same Streaming Multiprocessor (SM).
- **Global Memory** : Shared by all threads of a computing application and implemented in external DRAM.

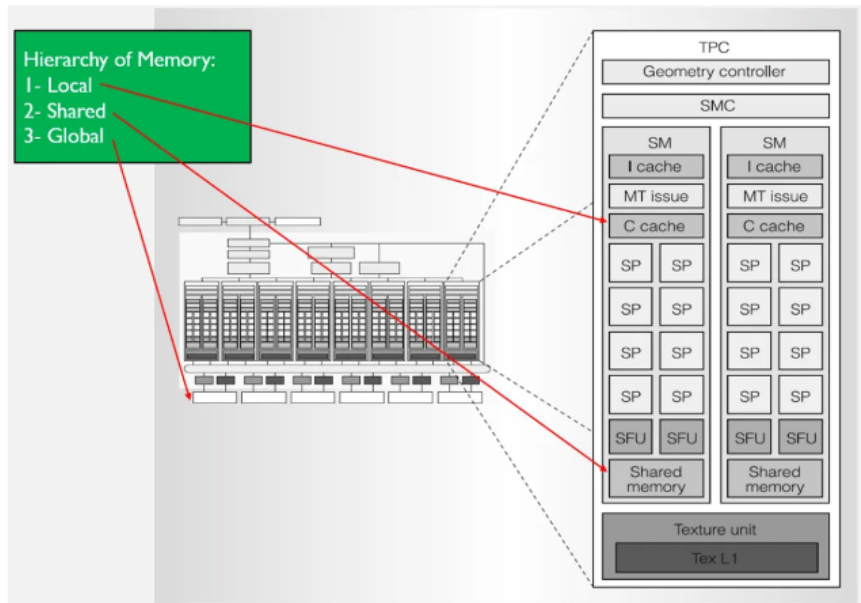


FIGURE 19 – Memory Hierarchy

To synchronize threads within an SM, particularly those communicating via shared and global memory, a **fast barrier synchronization instruction** is used, enabling efficient coordination of parallel thread execution.

An optimization technique called **memory coalescing** is employed to improve performance. This technique merges multiple parallel thread memory accesses into fewer, more efficient block-level accesses. For coalescing to occur, the addresses must fall within the same memory block and meet alignment criteria. Coalescing reduces overhead and significantly enhances performance compared to separate memory requests.

Additionally, Tesla GPUs support **atomic memory operations**, such as integer additions, minimum/maximum operations, logical operators, and compare-and-swap. These atomic operations are essential for parallel reductions and efficient management of parallel data structures, improving concurrency and the overall performance of computational tasks.

Overall, these memory access features, combined with coalescing and atomic operations, improve memory bandwidth, reduce latency, and boost performance in computational tasks.

4 The Scalability of Parallel Computing in Tesla Architecture

The Tesla scalable parallel computing architecture is designed to enable GPUs to excel in throughput computing, which is essential for executing both high-performance computing (HPC) and graphics applications. Throughput computing has several characteristics that set it apart from traditional CPU-based, serial applications :

Key Characteristics of Throughput Applications

- **Extensive Data Parallelism** : Thousands of independent computations are performed in parallel on different data elements, allowing for high throughput. For example, in graphics rendering, each pixel may be processed independently.
- **Modest Task Parallelism** : Groups of threads (smaller program units) execute the same program, but different groups can run different programs. This parallel structure enhances efficiency, particularly for complex tasks.
- **Intensive Floating-Point Arithmetic** : GPUs excel at performing floating-point calculations, which are crucial for scientific computing, simulations, and 3D rendering.
- **Latency Tolerance** : Performance is measured by the amount of work completed within a given time. Tesla GPUs can tolerate some delays (e.g., waiting for memory access), continuing to process other tasks simultaneously.
- **Streaming Data Flow** : Tesla GPUs handle streaming data that requires high memory bandwidth with minimal data reuse. This is typical in real-time processing, such as video decoding or 3D rendering.
- **Modest Inter-Thread Synchronization** : Threads within graphics tasks usually do not need to communicate. Similarly, in parallel computing applications, synchronization and communication between threads are limited, which boosts performance.

Performance Evolution

Tesla GPUs have seen their parallel performance double every 12 to 18 months, primarily driven by the needs of the 3D gaming market. These GPUs are now available for a variety of systems, including laptops, desktops, and workstations, and they are programmable using CUDA (Compute Unified Device Architecture) tools, allowing developers to write parallel programs in C.

Decomposing Data-Parallel Problems

To leverage Tesla's parallel architecture, developers must decompose a large computing problem into smaller, parallelizable sub-problems. This involves dividing large datasets into blocks and further subdividing each block into elements. These blocks can be processed independently, and elements within each block are computed cooperatively by parallel threads.

Cooperative Thread Array (CTA)

In contrast to graphics programming, which often has independent parallel threads, parallel computing applications require threads to synchronize and communicate efficiently. The Cooperative Thread Array (CTA) is introduced in Tesla GPUs to manage concurrent threads that cooperate on solving a problem. A CTA consists of 1 to 512 threads, each with a unique Thread ID (TID). Threads in a CTA can share data and synchronize to ensure correct computation. CTAs execute as SIMT warps (32 threads) in parallel, depending on the available resources.

Parallel Granularity Levels

Tesla's parallel computing model operates at three granularities :

- **Thread Level** : Each thread computes individual elements.
- **CTA Level** : A group of threads computes blocks of data.
- **Grid Level** : Several CTAs together compute the final result. Dependent grids are executed sequentially.

Parallel Memory Sharing

Memory sharing in Tesla architecture occurs at three levels :

- **Local Memory** : Each thread has its own private memory for temporary variables.
- **Shared Memory** : Threads within a CTA share memory, making communication faster.
- **Global Memory** : Data is shared across multiple CTAs in a grid.

Transparent Scaling

The Tesla architecture is scalable, meaning that programs can run on GPUs with different numbers of Streaming Multiprocessors (SMs) and SP cores without recompiling. The workload is divided into independent blocks, which can be processed on any number of cores. The scaling is transparent to the program, meaning the program operates the same way whether it runs on a small or large GPU.

5 GPU examples

Main Domain of Usage	GPU Model	Compute Capability	Notes
2*Professional Graphics	Quadro FX Series	1.0 – 1.3	Designed for professional applications ; specific models include Quadro FX 4600, FX 5600.
<i>Continued on next page</i>			

Main Domain of Usage	GPU Model	Compute Capability	Notes
	Quadro NVS Series	1.0 – 1.3	Limited CUDA support; primarily for business and multi-display setups.
5*Consumer Graphics	GeForce 8 Series	1.0	Includes models like GeForce 8800 GTX, primarily for gaming and general graphics tasks.
	GeForce 9 Series	1.1	Successor to the 8 Series, offering improved performance for gaming.
	GeForce 100 Series	1.1	Rebranded versions of the 9 Series with minor enhancements.
	GeForce 200 Series	1.2 – 1.3	High-performance gaming GPUs; includes models like GeForce GTX 280.
	GeForce 300 Series	1.2 – 1.3	OEM-only series, based on previous architectures.
3*High-Performance Computing (HPC)	Tesla C870	1.0	One of the early GPUs designed for computational tasks.
	Tesla C1060	1.3	Improved double-precision performance for scientific computing.
	Tesla C2050/C2070	2.0	Features ECC memory and support for larger datasets.

6 Conclusion

The Tesla architecture marked a groundbreaking advancement in GPU technology, setting the foundation for modern graphics and general-purpose parallel computing. By introducing the Unified Shader Model, Tesla eliminated the division between vertex and pixel pipelines, optimizing hardware utilization and enhancing the efficiency of graphics rendering.

Equally transformative was Tesla’s pioneering role in GPGPU (General-Purpose GPU) computing, which opened the door to leveraging GPUs for non-graphics tasks such as

scientific simulations, machine learning, and big data processing. With the integration of flexible programming frameworks like CUDA, Tesla empowered developers to craft customized solutions, addressing computational challenges across a wide spectrum of fields.

The architecture's innovations in memory handling, task parallelism, and high computational throughput resolved traditional bottlenecks, enabling the efficient processing of vast datasets. This not only improved performance in graphics-heavy applications but also revolutionized industries ranging from healthcare and finance to artificial intelligence and engineering.

In essence, Tesla architecture redefined the role of GPUs in computing, bridging the gap between graphics specialization and general-purpose computation. It continues to influence GPU design and serves as a cornerstone for advancements in high-performance parallel computing.

7 Bibliography

Références

- [1] Caroline COLLANGE. *Analyse de l'architecture GPU Tesla*. 2010. URL : <https://hal.science/hal-00443875>.
- [2] IEEE. *Geforce8800 Architecture*. 2024. URL : https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08_tesla.pdf.
- [3] IEEE. *Streaming Multiprocessor*. 2024. URL : https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08_tesla.pdf.
- [4] IEEE. *Texture/Processor Cluster*. 2024. URL : https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08_tesla.pdf.
- [5] NVIDIA. *NVIDIA GPU Architecture History*. 2024. URL : <https://www.techpowerup.com/gpu-specs/nvidia-c51.g592>.