Register No

SRM Institute of Science and Technology

College of Engineering and Technology

School of Computing

SRM Nagar, Kattankulathur – 603203, Chengalpattu District, Tamilnadu

Academic Year: 2023-24 **SET-D-Answer Key**

Test: CLA-T1 **Date: 14.02.2024**
Course Code & Title: 18CSE419T-GPU Programming **Duration:** 1 Hour
Year & Sem: III Year / VI Sem **Max. Marks:** 25

Course Articulation Matrix:

S.No.	Course Outcome	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12
1	CO1	3	2										

Part – A (5 x 1 = 5 Marks) Instructions: Answer all Questions						
Q. No	**Question**	**Marks**	**BL**	**CO**	**PO**	**PI Code**
1	How many CIUDA cores in NVIDIA's Fermi Architecture? a) 16 b) 32 c) 64 d) 128 Ans:B	1	L1	1	1	1.3.1
2	Which one is not an operation 3D graphics pipeline? a) Z-scoring b) Rasterization c) Vectorizing d) Texturing Ans:A	1	L1	1	1	1.1.2
3	CUDA is a) Compute Unified Device Architecture b) Compute Unique Device Architecture c) Compute Use Data Architecture d) Compute Unified Data Architecture Ans:A	1	L2	1	1	1.1.2

4	Pipeline bubbles are a) stall cycles b) multitrack pipeline c) cache mapping d) data dependencies Ans:A	1	L1	1	1	1.3.1
5	How many double precision floating-point values could be stored on a 128-bit register? a) 1 b) 2 c) 3 d) 4 Ans:B	1	L2	1	2	2.1.1

Part – B(2 x 4 = 8 Marks)
Answer Any Two Questions

1. **State the advantages and disadvantages of multi-core trajectory and many-core trajectory.**

The multicore trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began with two core processors with the number of cores increasing with each semiconductor process generation. A current exemplar is the recent Intel Core i7t microprocessor with four processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full X86 instruction set, supporting hyper- threading with two hardware threads, designed to maximize the execution speed of sequential programs. In contrast, the many-thread trajectory focuses more on the execution throughput of parallel applications. The many-threads began with a large number of threads, and once again, the number of threads increases with each generation. A current exem- plar is the NVIDIA GTX680 graphics processing unit (GPU) with 16,384 threads, executing in a large number of simple, in-order pipelines.
Many-threads processors, especially the GPUs, have led the race of floating-point performance since 2003. As of 2012, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips: 1.5 teraflops versus 150 gigaflops double precision in 2012.

2. **List the design factors of GPU over CPU.**

The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution.

• More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications.

•Neither control logic nor cache memories contribute to the peak calculation speed. As of 2012, the high-end general-purpose multicore microprocessors typically have six to eight large processor cores and multiple megabytes of on-chip .

•Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the memory system into the processors. Graphics chips have been operating at approximately six times the memory bandwidth of contemporaneously available CPU chips.

•The design philosophy of GPUs is shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a **massive number of floating-point calculations** per video frame in advanced games.

•This demand motivates GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations.

•The prevailing solution is to optimize for the execution throughput of massive numbers of threads. The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency.

• The reduced area and power of the memory access hardware and arithmetic units allows the designers to have more of them on a chip and thus increase the total execution throughput.

•**Small cache memories** are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM.

•This design style is commonly referred to **as throughput-oriented design** since it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute.

•The CPUs, on the other hand, are designed to minimize the execution latency of a single thread.

•Large last-level on-chip caches are designed to capture frequently accessed data and convert some of the long-latency memory accesses into short-latency cache accesses.

•The arithmetic units and operand data delivery logic are also designed to minimize the effective latency of operation at the cost of increased use of chip area and power.

•By reducing the latency of operations within the same thread, the CPU hardware reduces the execution latency of each individual thread. However, the large cache memory, low-latency arithmetic units, and sophisticated operand delivery logic consume chip area and power that could be otherwise used to provide more arithmetic execution units and memory access channels.
This design style is commonly referred to as **latency-oriented design**.

•GPUs are designed as parallel, throughput oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well.

•For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.

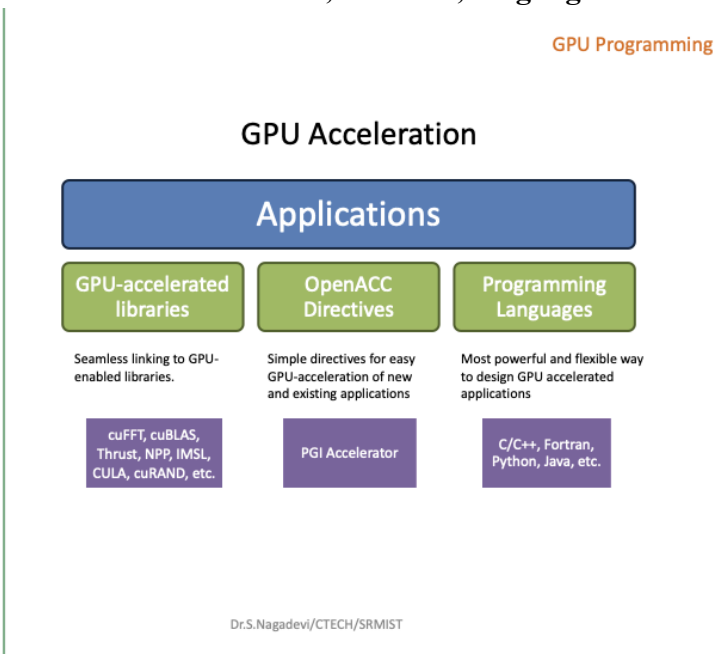
•When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs.

•Therefore, one should expect that many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.

•This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU-GPU execution of an application.

• The demand for supporting joint CPUGPU execution is further reflected in more recent programming models such as OpenCL , OpenACC , and C++ AMP.

3. **Summarize the directives, libraries, languages used in GPU**

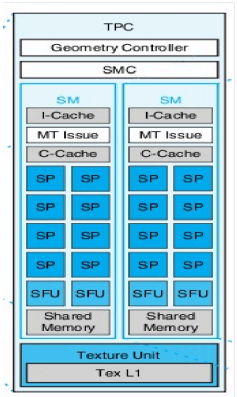


Part – C(1 x12 = 12 Marks)

4. Brief on the NVIDIA GPU architecture and its relation to graphics pipeline.

GPU terminology diers from conventional CPU language, and to make matters worse, NVIDIA changed its terminology over time, so we begin by dening the basic terms. We will use the GEForce 8800 GTX as an example architecture.
A discrete GPU is one that sits on a card that is plugged into the PCI-Express interconnect bus. In contrast, a motherboard-GPU is integrated into the chipset on the motherboard. Tesla-based GPUs can have from 1 to 16 nodes, each called a streaming multiprocessor (SM). NVIDIA uses the terms "node" and streaming multiprocessor interchangeably. The largest version available in 2008 was the GeForce 8800 GTX, with 16 SMs, each containing 8 multithreaded single-precision oating point and integer processing units, which used to be called streaming processors (SP) but are now called cores. The clock rate was 1.35 GHz. The later Fermi-based GPUs support 32 cores per streaming multiprocessor, implying that they have up to 512 cores. The newest architecture, at the time of this writing, is the Kepler, which has up to 15 Streaming Multiprocessor (SMX) units, each of which has up to 192 single-precision CUDA cores, with each core having fully pipelined oating-point and integer arithmetic logic units. Figure 6 shows the architecture of a single Kepler SMX processor.
Figure4 depicts the architecture of an NVidia GEForce 8800 that has 14 streaming multiprocessors, dis- tributed into 7 pairs of SMs. Each SM pair is integrated into a unit called a texture/processor cluster (TPC) that contains a shared geometry controller, a streaming multiprocessor controller (SMC), and a shared texture unit and cache and shared lines for load/store and I/O, as shown in Figure 5. The SMC controls access to the shared texture unit, load/store path, and I/O path. Each TPC is connected to an interconnec- tion network that connects it to the device memory,

L2 cache (containing textures), raster processors, and the interface to



The texture/processing cluster.

SMX architecture of the Kepler GK110 (*from the NVIDIA Kepler GK110 whitepa*

the actual display device, as well as to the bridge to the CPU and system memory.

Each SM in the Tesla GPUs contains 8 cores (32 in the Fermi GPUs), as well as its own local shared memory, an instruction cache, a constant cache, a multithreaded instruction unit, and two special function units (SFUs). The special function units compute special functions such as the transcendental functions (e.g., trigonometric functions), reciprocals, and square roots.
A core (SP) is the primary thread processor. In the GEForce 8800, each core is a multithreaded processor supporting 96 threads, with a register _le containing 1024 scalar 32-bit registers for the use of these threads.
The processor is fully pipelined and implements all 32-bit and 64-bit integer arithmetic, comparison, conver-1sion, and logical PTX1
instructions, as well as IEEE 754 standard single precision _oating point operations and a compatible add/multiply operation. Earlier GPUs executed vector instructions, but the later GPUs were designed so that core processors executed ordinary scalar instructions.

5. Draw the process of programmable function NVIDIA graphics pipeline operations in detail.

In 2001, the NVIDIA GeForce 3 took the first step toward true general shader programmability. It exposed the application developer to what had been the private internal instruction set of the floating-point vertex engine (VS/T&L stage). This coincided with the release of Microsoft DirectX 8 and OpenGL vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating-point capability to the pixel shader stage, and made texture accessible from the vertex shader stage. The ATI Radeon 9700, introduced in 2002, featured a programma- ble 24-bit floating-point pixel shader processor programmed with DirectX 9 and OpenGL. The GeForce

FX added 32-bit floating-point pixel proces- sors. These programmable pixel shader processors were part of a general trend toward unifying the functionality of the different stages as seen by the application programmer. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs dedicated to vertex and pixel processing. The XBox 360 introduced an early unified-processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

In graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the posi- tions of triangle vertices or generating pixel colors. This data indepen- dence as the dominating application characteristic is a key difference between the design assumptions for GPUs and CPUs. A single frame, ren- dered in 1/60 of a second, might have 1 million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms. Such variation has motivated the hard- ware designers to make those pipeline stages programmable. Two particular programmable stages stand out: the vertex shader and the pixel shader. Vertex shader programs map the positions of triangle ver- tices onto the screen, altering their position, color, or orientation. Typically a vertex shader thread reads a floating-point (x, y, z, w) ver- tex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Vertex shader programs and geometry shader programs execute on the VS/ T&L stage of the graphics pipeline.

Pixel shader programs each "shade" one pixel, computing a floating- point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. These programs execute on the shader stage of the graphics pipeline. For all three types of graphics shader programs, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects. This property has motivated the design of the programmable pipe- line stages into massively parallel processors.

Figure 2.4 shows an example of a programmable pipeline that employs a vertex processor and a fragment (pixel) processor. The programmable vertex processor executes the programs designated to the VS/T&L stage

and the programmable fragment processor executes the programs desig- nated to the (pixel) shader stage. Between these programmable graphics pipeline stages are dozens of fixed-function stages that perform well- defined tasks far more efficiently than a programmable processor could, and that would benefit far less from programmability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. Together, the mix of programmable and fixed-function stages is engi- neered to balance extreme performance with user control over the render- ing algorithms.

Common rendering algorithms perform a single pass over input primi- tives and access other memory resources in a highly coherent manner. That is, these algorithms tend to simultaneously access contiguous mem- ory locations, such as all triangles or all pixels in a neighborhood. As a result, these

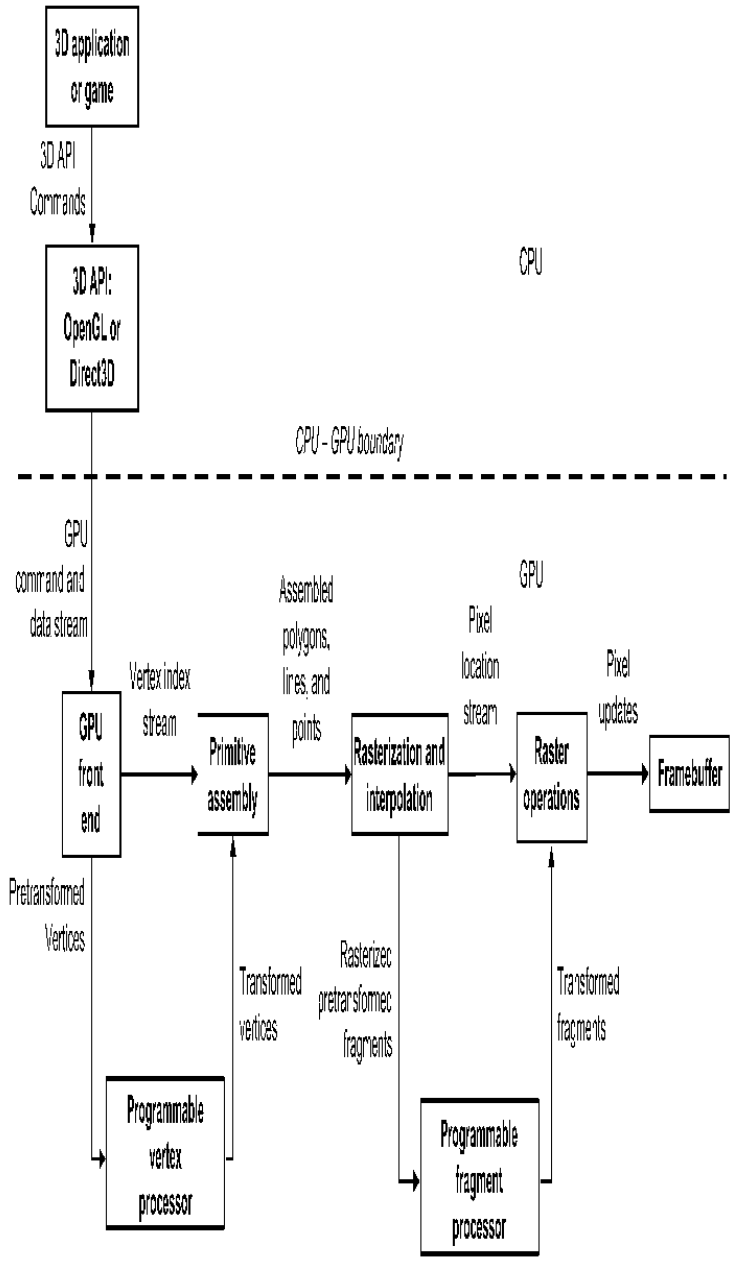


FIGURE 2.4

An example of a separate vertex processor and fragment processor in a programmable graphics pipeline.

algorithms exhibit excellent efficiency in memory bandwidth utilization and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute-limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. In par- ticular, whereas the CPU die area is dominated by cache memories, GPUs are dominated by floating-point data path and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (since latency can be readily hidden by massively parallel execution); indeed, bandwidth is typically many times higher than a CPU, exceeding 190 GB/s in more recent designs.