SRM Institute of Science and Technology
College of Engineering and Technology
School of Computing

Mode of Exam
OFFLINE

DEPARTMENT OF COMPUTING TECHNOLOGIES
SRM Nagar, Kattankulathur – 603203, Chengalpattu District, Tamilnadu

Academic Year: 2023-2024 (Even) **SET A- Answer Key**

Test: CLAT-1 **Date: 14.02.24**
Course Code & Title: 18CSE419T & GPU Programming **Duration:** 50 minutes
Year & Sem: III & V **Max. Marks:** 25

Course Articulation Matrix:

S.No.	Course Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
1	CO1	3	2										

Part – A
(5 x1= 5 Marks)
Instructions: Answer all

Q. No	Question	Marks	BL	CO	PO	PI Code
1	---------- leads to concurrency. a)Serialization b)Parallelism c)Serial processing d)Distribution **Ans: B**	1	1	1	1	1.31*
2	To which class of systems does the von Neumann computer belong? a)SIMD (Single Instruction Multiple Data) b)MIMD (Multiple Instruction Multiple Data) c)MISD (Multiple Instruction Single Data) d)SISD (Single Instruction Single Data) **Ans: D**	1	1	1	1	1.3.1
3	What is a high performance multi-core processor that can be used to accelerate a wide variety of applications using parallel computing. a)CLU b)GPU c)CPU d)DSP **Ans: B**	1	1	1	1	1.3.2
4	Speedup tends to saturate and efficiency------- as a consequence of Amdahl's law. a)increase b) constant c) decreases d) none Ans: C	1	1	1	1	1.3.1

5.	A code, known as GRID, which runs on GPU consisting of a set of a)32 Thread b)32 Block c)Unit Block d)Thread Block **Ans: D**	**1**	**1**	**1**	**1**	**1.3.3**
	Part – B **(2 x4 = 8 Marks)** **Instructions: Answer any 2**					
6	Compare the differences between CPUs and GPUs.	**4**	**1**	**1**	**1**	**2.2.3**

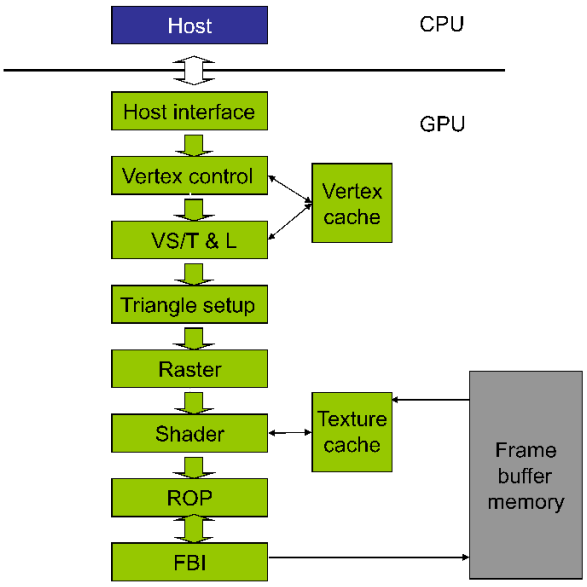
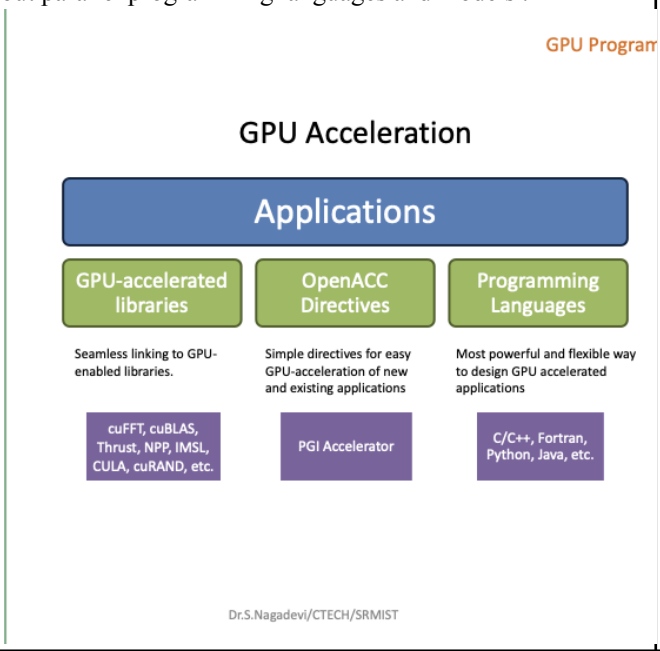
CPU	GPU
A smaller number of larger cores (up to 24)	A larger number (thousands) of smaller cores
Low latency	High throughput
Optimized for serial processing	Optimized for parallel processing
Designed for running complex programs	Designed for simple and repetitive calculations
Performs fewer instructions per clock	Performs more instructions per clock
Automatic cache management	Allows for manual memory management
Cost-efficient for smaller workloads	Cost-efficient for bigger workloads

7	Illustrate fixed function NVDIA GeForce graphics pipeline.	**4**	**1**	**1**	**1**	**2.2.3**

the leading performance graphics hardware was fixed-function pipelines that were configurable, but not pro- grammable. In that same era, major graphics Application Programming Interface (API) libraries became popular. An API is a standardized layer of software, that is, a collection of library functions that allows applica- tions (e.g., games) to use software or hardware services and functionality. For example, an API can allow a game to send commands to a graphics processing unit to draw objects on a display. One such API is DirectX, Microsoft's proprietary API for media functionality. The Direct3D compo- nent of DirectX provides interface functions to graphics processors. The other major API is OpenGL, an open-standard API supported by multiple vendors and popular in professional workstation applications. This era of fixed-function graphics pipeline roughly corresponds to the first seven generations of DirectX.

he host interface receives graphics commands and data from the CPU. The commands are typically given by application programs by calling an API function. The host interface typically contains a specialized DMA hardware to efficiently transfer bulk data to and from the host system memory to the graphics pipeline. The host interface also

	communicates back the status and result data of executing the commands. Before we describe the other stages of the pipeline, we should clarify that the term vertex usually means the “corners” of a polygon. The GeForce graphics pipeline is designed to render triangles, so vertex is typi- cally used to refer to the corners of a triangle. The surface of an object is drawn as a collection of triangles. The finer the sizes of the triangles are, the better the quality of the picture typically becomes. The vertex control stage in Figure 2.1 receives parameterized triangle data from the CPU. The vertex control stage converts the triangle data into a form that the hardware understands and places the prepared data into the vertex cache. The vertex shading, transform, and lighting (VS/T&L) stage in Figure 2.1 transforms vertices and assigns per-vertex values (colors, nor- mals, texture coordinates, tangents, etc.). The shading is done by the pixel shader hardware. The vertex shader can assign a color to each vertex but it is not applied to triangle pixels until later. The triangle setup stage further creates edge equations that are used to interpolate colors and other per- vertex data (e.g., texture coordinates) across the pixels touched by the tri- angle. The raster stage determines which pixels are contained in each tri- angle. For each of these pixels, the raster stage interpolates per-vertex values necessary for shading the pixel, which includes color, position, and texture position that will be shaded (painted) on the pixel. The shader stage in Figure 2.1 determines the final color of each pixel. This can be generated as a combined effect of many techniques: interpola- tion of vertex colors, texture mapping, per-pixel lighting mathematics, reflections, and more. Many effects that make the rendered images more realistic are incorporated in the shader stage					

	 JRE 2.1 xed-function NVIDIA GeForce graphics pipeline.					
8	Discuss about parallel programming languages and models . 	**4**	**1**	**1**	**1**	**2.2.3**
	Part – C(1x12 = 12 Marks) Instructions: Answer any 1					
9	Elaborate in detail about architecture of modern GPU with a neat sketch.	**12**	**1**	**1**	**1**	**2.1.2**

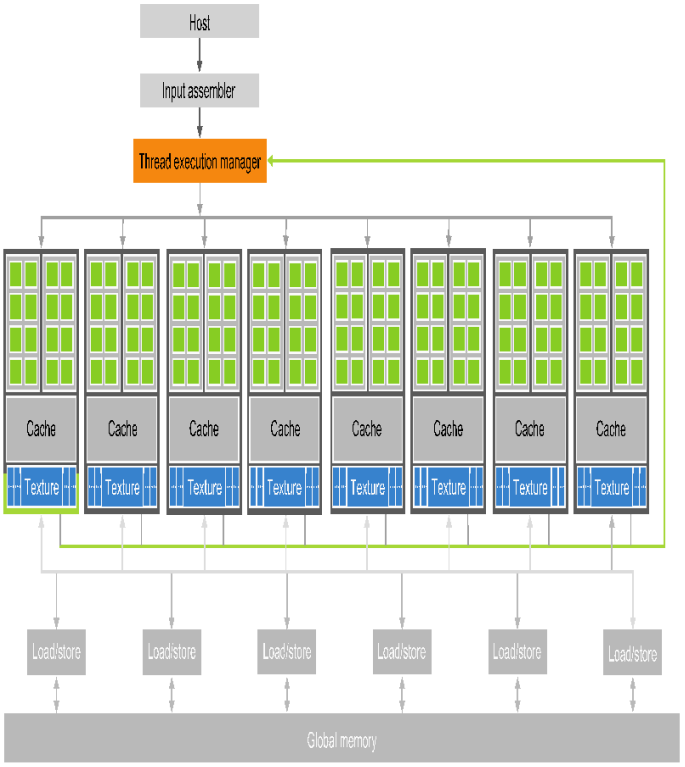
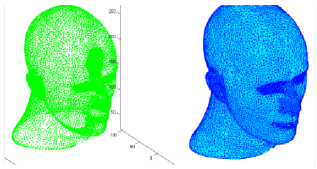


FIGURE 1.2

Architecture of a CUDA-capable GPU.

Figure 1.2 shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Figure 1.3, two SMs form a building block. However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, in Figure 1.3, each SM has a number of streaming processors (SPs) that share control logic and an instruction cache. Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory in Figure 1.3. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images and texture information for 3D rendering. But for computing, they function as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

The G80 introduced the CUDA architecture and had 86.4 GB/s of memory bandwidth, plus a communication link to the CPU

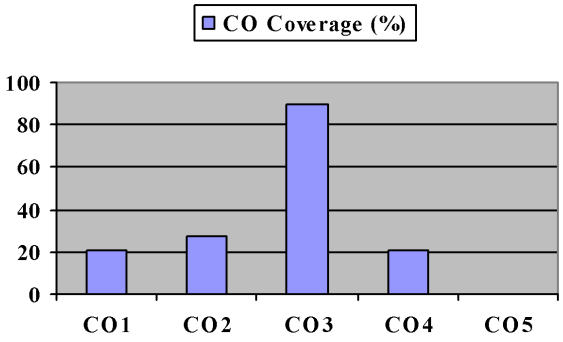
	core logic over a PCI-Express Generation 2 (Gen2) interface. Over PCI-E Gen2, a CUDA application can transfer data from the system memory to the global memory at 4 GB/s, and at the same time upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 GB/s. More recent GPUs use PCI-E Gen3, which supports 8 GB/s in each direc- tion. As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E to communicate with the CPU system memory if there is need for using a library that is only available on the CPUs. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future. With 16,384 threads, the GTX680 exceeds 1.5 teraflops in double precision. A good application typically runs 5,000_12,000 threads simul- taneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support two or four threads, depending on the machine model, per core. CPUs, however, are increasingly used with SIMD (single instruction, multiple data) instructions for high numerical performance. The level of parallelism supported by both GPU hardware and CPU hardware is increasing quickly. It is therefore very important to strive for high levels of parallelism when developing computing applications.					
10	Explain in detail about 3D Graphics pipeline and its stages with diagram.  Fig. 2: Triangulated representation of a head  The Input It will help to understand why the GPU architecture is what it is if you have a basic understanding of what must be done to render a 3D image to the screen, with all of the lighting e_ects and texturizing that you see in modern day applications. The most demanding application of a graphics processor is to perform real-time, high-resolution 3D image processing at a frame rate no less than 60 frames per second, as is necessary to create computer-based animations. To make this possible, the graphics processor must be able to exploit the high degree of data-parallelism in the problem. A 3D image is represented by a 3D model, which is a collection of 3D points. Usually, 3D models start out as triangulated surfaces, as shown in Figure 2. The vertices of the triangles	**12**	**1**	**1**	**1**	**2.1.2**

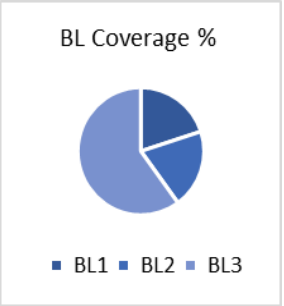
	de_ne the shape and are the starting point for display of the object. Thus, a 3D shape begins as a set of these vertices. Graphics processing proceeds as a sequence of pipelined stages. The basic graphics pipeline is shown in Figure 3. The individual stages are described below. 1.4.2 Z-Bu_ers One of the major tasks in three-dimensional (3D) graphics processing is determining where each 3D point should be placed on the 2D screen (which is essentially a projection problem), and what color it should have. A key concept underlying the projection problem is how to determine which 3D points are visible and which are obscured by other points, which is the visibility problem. In computer graphics, z-bu_ering is the management of image depth coordinates in three-dimensional (3D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, **The Pipeline Stages** Input Assembler The input assembler receives the 3D representation of the scene as a collection of geometric primitives such as points, lines, and vertices. It then distributes the vertices to the vertex shader. The input assembler typically assembles vertices into several di_erent primitive types such as line lists, triangle strips, or primitives with adjacency. Vertex Shading Shader programs in general determine how lighting and shadows interact with the surfaces to be rendered. A vertex shader is a graphics processing function that maps vertices onto the screen and adds special e_ects to objects in a 3D environment by performing mathematical operations on the objects' vertex data. One of its purposes is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen, as well as a depth value for the Z-bu_er, and then to apply color to it. Geometry Shading Geometry shading is the stage of the graphics pipeline after vertex shading. Its purpose is to enhance the details and accuracy of the 2D image by working at a larger degree of granularity than individual vertices. Its inputs consist of geometric primitives consisting of more than one vertex, such as lines and triangles. A geometry shader can take as its input, for example, the three points of a triangle and output intermediate points that can be used to re_ne the surface. It can only do this by operating with greater granularity. The geometry shader can modify the positions and orientation of the primitives. Rasterization The word "raster" was originally used in the raster scan of cathode ray tubes (CRT), which paint the image line by line; the term is now used to mean a grid of pixels. Rasterization is a process for converting vectorized input to a bitmap form, i.e. a 2D array of pixels. Given a triangle, for example, represented by three vertices, it determines the locations of all pixels and pixel fragments that lie inside, or on the edge of, this triangle. Pixel Shading A pixel shader is a function that computes the color and other attributes of each pixel or pixel fragment. Pixel shaders range from always outputting the same color, to applying a lighting value, to doing bump mapping, shadows, specular highlights, translucency and other phenomena. They					

	can alter the depth of the pixel (for Z-bu_ering), or output more than one color if multiple render targets are active. A pixel shader alone cannot produce very complex e_ects, because it operates only on a single pixel, without knowledge of a scene's geometry or of neighboring pixels.					

***Performance Indicators are available separately for Computer Science and Engineering in AICTE examination reforms policy.**

Course Outcome (CO) and Bloom's level (BL) Coverage in Questions





Approved by the Audit Professor/Course Coordinator