Register number ______________________________

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 (EVEN)
B.Tech-Computer Science & Engineering **SET - B**

Test: CLA-T2 **Date: 28.03.2024**
Course Code & Title: 18CSE419T & GPU Programming **Duration: 2 periods**
Year & Sem: III Year /VI Sem **Max. Marks: 50**

Course articulation matrix:

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO-1	3														3
CO-2		3	2												3
CO-3		3	3												3
CO-4		3	3												3
CO-5			3	1									2		3

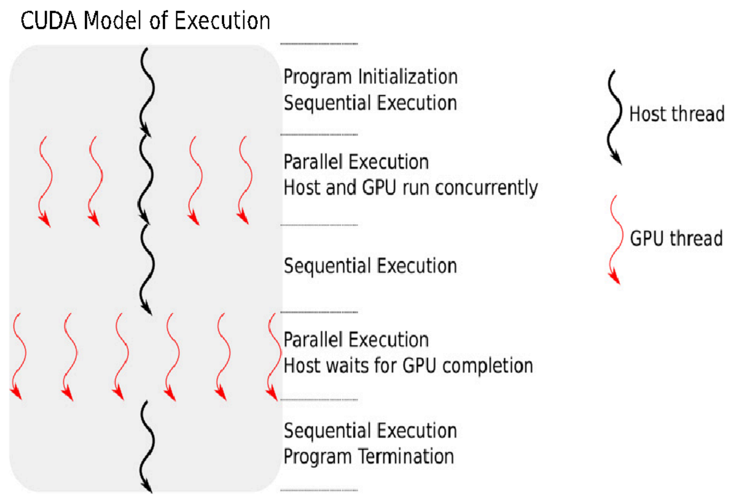
Part – A(1*10=10 Marks)
Answer All the Questions

Q. No	Questions	Marks	BL	CO	PO	PI Code
1	Calling a kernel is typically referred to as ----------- a) Kernel Thread b) Kernel initialization c) Kernel termination d) Kernel invocation Ans: D	1	1	CO 3	2 & 3	**4.2.1**
2	The CUDA architecture consists of ----------- parallel computing kernels and functions a) RISC b) CISC c) PTX d) ZISC Ans: C	1	1	CO 3	2 & 3	**4.2.1**
3	The maximum number of threads that can be launched in a specific block is a) 8 b) 32 c) 256 d) 1024 Ans: D	1	1	CO 3	2 & 3	**4.2.1**
4	NVDIA CUDA warp is made up of how many threads? a) 512 b) 1024 c) 312 d) 32 Ans: D	1	1	CO 3	2 & 3	**4.2.1**
5	For a vector addition, assume that the vector length is 2000, each thread calculates one output element and the thread block size is 512 threads. How many threads will be in the grid? a) 2000 b) 2024 c) 2048 d) 2096 Ans: C	1	1	CO 3	2 & 3	**4.2.1**

Register number ______________________________

6	NVDIA calls occupancy as the ratio of a) Resident-warps/maximun-warps b) Maximum-warps/resident-warps c) Resident-grids/maximum-grids d) Maximum-blocks/resident-grids Ans: A	1	1	CO 3	2 & 3	**4.2.1**
7	The page -locked memory is also referred to as a) Constant memory b) Zero-copy memory c) Pinned memory d) Texture memory Ans: B	1	1	CO 3	2 & 3	**4.2.1**
8	If a variable a is host variable and dev_a is a device variable to allocate a memory to dev_a. Select the correct statement. a) Cudamalloc(&dev_a,sizeof(int)) b) Malloc(&dev_a,size of (int)) c) Cudamalloc (void **)&dev_a,size of (int)) d) Malloc (void **)&dev_a size of (int)) Ans: C	1	1	CO 3	2 & 3	**4.2.1**
9	Threads are scheduled to run on an SM as a a) Thread b) Block c) Warp d) Grid Ans: B	1	1	CO 3	2 & 3	**4.2.1**
10	Which one is not an off-chip memory in GPU? a) Cache memory b) Global memory c) Texture memory d) Constant memory Ans: A	1	1	CO 4	3	**4.2.1**

Part – B (4*4=16 marks)
Answer any four Questions

Q. No	**Question**	**Marks**	**BL**	**CO**	**PO**	**PI Code**
11	With a neat sketch showcase the steps involved in the GPU program execution model.  CUDA Model of Execution Program Initialization Sequential Execution Parallel Execution Host and GPU run concurrently Sequential Execution Parallel Execution Host waits for GPU completion Sequential Execution Program Termination Host thread GPU thread **FIGURE 6.2** The GSLP CUDA execution model. The host machine may continue execution (default behavior) or may block, waiting for the completion of the GPU threads.	4	2	CO 2	2 & 3	**4.2.1**

Register number ______________________________

12	What is zero-copy memory in GPU? What is the need for it? • Zero-copy memory is a term used to convey that **no explicit memory transfer between the host and t**he device needs to be initiated. • Another, less fashionable term used for the same concept is **mapped memory.** • Mapped memory is page-locked memory that can be mapped to the address space of the device. • So, we have a memory region with two addresses: one for access from the host and one for access from the device. • A transfer across the PCIe bus will be initiated by the CUDA run-time upon the first attempt to access a region of memory that is designated as mapped memory, stalling the active kernel while it is taking place. This process may sound inefficient, but there is still justification for using mapped memory: • It makes the program logic simpler because there is no need to separately allocate device memory and transfer the data from the host. This can be a viable option for early development phases that involve porting CPU code to CUDA. Devoting lengthy parts of the code for memory transfers may be a distraction that can be reserved for the later stages, when the core logic of the program behaves as expected. • The CUDA run-time can automatically overlap kernel-originating memory transfers with another kernel execution. This can boost performance without the need for using streams. • For low-end systems where the CPU and the GPU share the same physical RAM, no transfer ever takes place, making the use of mapped memory in such cases a no-brainer.	4	3	CO 2 & CO 3	2 & 3	**4.2.1**
13	Identify and state the tools used for profiling the CUDA programs.	4	3	CO 3	2 & 3	**4.2.1**
14	List the CUDA functions used in memory hierarchy.	4	3	CO 3	2 & 3	**4.2.1**
15	What is shared memory in GPU? What is the uses of it? • *Shared memory* is a block of fast on-chip RAM that is shared among the cores of an SM. • Each SM gets its own block of shared memory, which can be viewed as a user-managed L1 cache. • In Fermi and Kepler architectures, shared memory and L1 cache are actually part of the same on-chip memory that can be programmatically partitioned in different ways. • Currently, each SM in both architectures sports 64 KB RAM that can be partitioned as 16 KB/48 KB or 48 KB/16 KB to serve these two roles. • Compute-Capability 3.x devices also have the option of a 32 KB/32 KB split. • Compute-Capability 1.x devices come with only 16 KB shared memory and no L1 cache. • A programmer can specify the *preferred* device-wide arrangement by calling the cudaDeviceSetCacheConfig function. • A *preferred* kernel-specific arrangement can be also set with the cudaFuncSetCacheConfig function. • Both of these func- tions set only a preference. • Shared memory can be used in the following capacities: • As a holding place for very frequently used data that	4	3	CO 3	2 & 3	**4.2.1**

Register number ________________________________

	would otherwise require global memory access • As a fast *mirror* of data that reside in global memory, if they are to be accessed multiple times • As a fast way for cores within an SM, to share data • Shared memory can be statically or dynamically allocated. • *Static allocation* can take place if the size of the required arrays is known at compile time. • **Dynamic allocation** is needed if shared memory requirements can be only calculated at run-time, i.e., upon kernel invocation.					

Part – C (2*12=24 marks)
Answer any four Questions

16	Write a CUDA program to illustrate the vector addition using single thread block.	12	3	CO 3	2 & 3	**4.2.1**

```
#include <stdio.h>
#include<time.h>

#include<math.h>

__global__ void vecAdd_kernel_UsingManyBlocks(int
*d_a, int *d_b, int *d_c, int N)
{

  //int tid = threadIdx.x;// threadIdx.x returns
thread IDs local to each thread block

 int gid = threadIdx.x + blockIdx.x * blockDim.x;

   c[gid] = a[gid] + b[gid];
  if(tid < N) // To avoid out of order memory access
by thread

  {

   d_c[tid] = d_a[tid] + d_b[tid];

  }

}

void vecAddCPU(int *h_a, int *h_b, int *h_c, int N)

{

  int i;

  for(i = 0; i < N; i++)

  {

    h_c[i] = h_a[i] + h_b[i];

  }

}

int main()

{

 int i,N = 16;

  /********************** Memory Allocation on CPU
**********************************/
```

Register number ________________________________

```
  int *h_a = (int *)malloc(N * sizeof(int)); //
Memory allocation on CPU for vector h_a  input

  int *h_b = (int *)malloc(N * sizeof(int)); //
Memory allocation on CPU for vector h_b  input

  int *h_c = (int *)malloc(N * sizeof(int)); //
Memory allocation on CPU for vector h_c  output

  int *hr_c = (int *)malloc(N * sizeof(int));//
Memory allocation on CPU for vector hr_c  output from
GPU

  int *d_a, *d_b, *d_c; // Decleration of GPU
variables

  /************************* Data initialization on
CPU ****************************/


   for(i = 0; i < N; i++)

   {

     h_a[i] = 2;

     h_b[i] = 2;

     h_c[i] = 0;

     hr_c[i] = 0;

   }

   /*

   for(i = 0; i < N; i++)

   {

     printf("\n h_c[%d] = %d",i,h_c[i]);

   }

   */

 /************************* 2. Memory allocation on
GPU ******************************/

   cudaMalloc((void **)&d_a, N*sizeof(int)); //
Allocate memory on GPU for variable d_a  input

   cudaMalloc((void **)&d_b, N*sizeof(int)); //
Allocate memory on GPU for variable d_b  input

   cudaMalloc((void **)&d_c, N*sizeof(int)); //
Allocate memory on GPU for variable d_c  output

  /******************* 3. Transfer data from Host to
Device **************************/

  cudaMemcpy(d_a, h_a, N*sizeof(int),
cudaMemcpyHostToDevice); // Copy data from Host to
Device for vector a
```

Register number ______________________________

```
  cudaMemcpy(d_b, h_b, N*sizeof(int),
cudaMemcpyHostToDevice); // Copy data from Host to
Device for vector b

  cudaMemcpy(d_c, h_c, N*sizeof(int),
cudaMemcpyHostToDevice); // Copy data from Host to
Device for vector c

  /******************** 4. Kernel lauch to execute
vector addition on Device
****************************/

  /*Vector Addition using many thread blocks*/
  vecAdd_kernel_UsingManyBlocks<<<2, 8>>>(d_a, d_b,
d_c, N);

 /************************** 5. Copy results back from
GPU to CPU****************************************/
cudaMemcpy(hr_c, d_c, N*sizeof(int),
cudaMemcpyDeviceToHost);//Copy data from GPU to Host
for vector c



/**************** Validate whether CPU and GPU
results are matching or NOT
****************************/

  for(i = 0; i < N; i++)

  {

   printf("\n hr_c[%d] = %d", i, hr_c[i]);

   }
  /* 6. Free CPU memory*/

  free(h_a);

  free(h_b);

  free(h_c);

  free(hr_c);

  /* 6. Free GPU memory*/

  cudaFree(d_a);

  cudaFree(d_b);

  cudaFree(d_c);

return(0);

}
```

17	Relate and discuss streaming multiprocessors and warps in the process of execution of a kernel. • GPU cores are essentially vector processing units, capable of applying the same instruction on a large collection of operands. So, when a kernel is run on a GPU core, the same instruction sequence is synchronously executed by a large collection of processing units **called streaming processors, or SPs.** • A group of SPs that execute under the control of a single control unit is **called a streaming multiprocessor, or SM.**	12	3	CO 3	2 & 3	**4.2.1**

Register number ______________________________

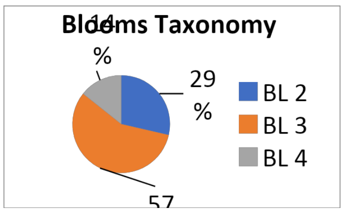
	• A GPU can contain multiple SMs, each running each own kernel. Since each thread runs on its own SP, we will refer to SPs as cores (Nvidia documentation calls them CUDA cores), although a more purist approach would be to treat SMs as cores. • Nvidia calls this execution model Single-Instruction, Multiple Threads (SIMT). • SIMT is analogous to SIMD. • The only major difference is that in SIMT the size of the "vector" on which the processing elements operate is determined by the software, i.e., the block size. • The computational power of a GPU (and consequently, its target market) is largely determined, at least within the members of a family, by the number of SMs available. • As an example, Table 6.2 lists a number of legacy, Fermi, Kepler, and Maxwell-class GPU offerings • Threads are scheduled to run on an SM as a block. • The threads in a block do not run concurrently, though. • Instead they are executed in groups called warps. • The size of a warp is hardware-specific. • The current CUDA GPUs use a warp size of 32. • At any time instance and based on the number of CUDA cores in an SM, we can have 32 threads active (one full active warp), 16 threads active (a half-warp active), or 8 threads active (a quarter-warp active). • The benefit of interleaving the execution of warps (or their parts) is to hide the latency associated with memory access, which can be significantly high. • An SM can switch seamlessly between warps (or half- or quarter-warps) as each thread gets its own set of registers. • Each thread actually gets its own private execution context that is maintained on-chip. Each SM can have multiple warp schedulers, e.g., in Kepler there are four. This means that up to four independent instruction sequences from four warps can be issued simultaneously • Once an SM completes the execution of all the threads in a block, it switches to a different block in the grid. • In reality, each SM may have a large number of resident blocks and resident warps, i.e., executing concurrently. • In compute capability 3.x devices, each SM has 192 CUDA cores. • So, how can we maximize the utilization of an SM if only 32 threads are active at any time? Obviously the answer is that we cannot, unless we have multiple warps running on the SM. • Each SM of a compute capability 3.x device has four warp schedulers, which means it can direct four different instruction sequences. • This would still leave 192 − 4 ∗ 32 = 64 cores unused. These are utilized by the SM for running more warps in case there is a stall.					
18	Describe the characteristics of constant memory as (i) constant memory as caching (ii) constant memory as broadcast • Constant memory is a form of virtual addressing of global memory. • There is no special reserved constant memory block. • **Constant memory has two special properties** you might be interested in. • First, it is cached,	12	3	CO 3	2 & 3	**4.2.1**

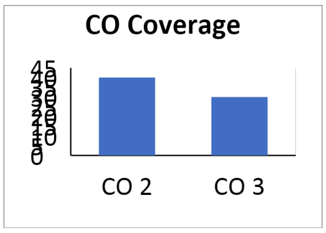
Register number ______________________________

    - and second, it supports broadcasting a single value to all the elements within a warp.
- Constant memory, as its name suggests, is for **read-only memory**.
- This is memory that is either declared at compile time as read only or defined at runtime as read only by the host.
- On compute 1.x devices (pre-Fermi), constant memory has the property of being cached in a small 8K L1 cache, so subsequent accesses can be very fast.
- This is providing that there is some potential for **data reuse** in the memory pattern the application is using. It is also highly optimized for **broadcast access** such that threads accessing the same memory address can be serviced in a single cycle.
- With a 64 K segment size and an 8 K cache size, you have an 8:1 ratio of memory size to cache, which is really very good.
- If you can contain or localize accesses to 8 K chunks within this constant section you'll achieve very good program performance. On certain devices you will find localizing the data to even smaller chunks will provide higher performance.
- With a nonuniform access to constant memory a cache miss results in N fetches from global memory in addition to the fetch from the constant cache.
- Thus, a memory pattern that **exhibits poor locality and/or poor data reuse** should not be accessed as constant memory.
- Also, each **divergence in the memory fetch** pattern causes serialization in terms of having to wait for the constant memory.
- Thus, a warp with 32 separate fetches to the constant cache would take at least 32 times longer than an access to a single data item. This would grow significantly if it also included cache misses.
- Single-cycle access is a huge improvement on the several hundred cycles required for a fetch from global memory.
- However, the several hundred–cycle access to global memory will likely be hidden by task switches to other warps, if there are enough available warps for the SM to execute.
- **Thus, the benefit of using constant memory for its cache properties relies on the time taken to fetch data from global memory and the amount of data reuse the algorithm has.**
- As with shared memory, the low-end devices have much less global memory bandwidth, so they benefit proportionally more from such techniques than the high-end devices.
- **As broadcast**
- Constant memory has one very useful feature. It can be used for the purpose of distributing, or broadcasting, data to every thread in a warp.
- This broadcast takes place in just a single cycle, making this ability very useful.
- In comparison, a coalesced access to global memory on compute 1.x hardware would require a memory fetch taking hundreds of cycles of latency to complete. Once it has arrived from the memory subsystem, it would be distributed in the same manner to all threads, but only after a significant wait for the memory subsystem to provide the data. Unfortunately, this is an all too common problem, in that memory speeds have failed to keep pace with processor clock speeds.
- By using the broadcast mechanism, which is also present on

Register number ________________________________

	Fermi for L2 cache–based accesses, you can distribute data very quickly to multiple threads within a warp. • This is particularly useful where you have some common transformation being performed by all threads. • Each thread reads element N from constant memory, which triggers a broadcast to all threads in the warp. • Some processing is performed on the value fetched from constant memory, perhaps in combination with a read/write to global memory. You then fetch element N + 1 from constant memory, again via a broadcast, and so on. As the constant memory area is providing almost L1 cache speeds, this type of algorithm works well. •					





Approved by Audit Professor/ Course Coordinator

Register number ______________________________