Register Number									

Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam – 603 110 (An Autonomous Institution, Affiliated to Anna University, Chennai)						
Computer Science and Engineering **Continuous Assessment Test -2** **Question Paper**						
Degree & Branch	B.E				**Semester**	VII
Subject Code & Name	UCS1727– GPU Computing				**Regulation:**	**2018**
Academic Year	2023-2024 ODD	**Batch**	2020-2024	**Date**	**.10.2023**	**FN**
Time: 08:10 - 09:40 AM (90 Minutes)	**Answer All Questions**				**Maximum: 50 Marks**	

(K1: Remembering, K2: Understanding, K3: Applying, K4: Analyzing, K5: Evaluating)

CO1:	Understand GPU architecture (K2)
CO2:	Write programs using CUDA, identify issues and debug them (K3)
CO3:	Implement efficient algorithms in GPUs for common application kernels such as matrix multiplication (K3)
CO4:	Write simple programs using OpenCL (K3)
CO5:	Write an efficient parallel program for a given problem(K3).

Part – A (6×2 = 12 Marks)

1	K1	What are eager evaluation and lazy evaluation? Ans: In the eager evaluation model used by CPUs we stall at the first read into a1, and on each subsequent read. With the lazy evaluation model used by GPUs we stall only on consumption of the data, the additions in the third code segment, if that data is not currently available	CO2	2.1.2 2.2.2 13.1.1
2	K1	What is speculative execution? Ans: The CPU will have predicted a branch correctly, it makes sense to start executing the instruction stream at that branch address. However, this adds to the cost of branch misprediction, as now the instruction stream that has been executed has to be discarded. The optimal model for both branch prediction and speculative execution is simply to execute both paths of the branch and then commit the results when the actual branch is known.	CO2	1.4.1 2.1.2 2.2.2
3	K3	Identify the CUDA API call used to retrieve GPU device properties?.	CO2	2.1.2

		cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp properties, int device);* *struct cudaDeviceProp device_0_prop;* *CUDA_CALL(cudaGetDeviceProperties(&device_0_prop, 0));*		13.3.2
4	K1	What is loop invariant analysis? Loop invariant analysis looks for expressions that are constant within the loop body and moves them outside the loop body.	CO2	2.1.2 2.2.2
5	K1	What does the term "cache coherence" refer to, and what are the available approaches to address cache coherence? Ans: Suppose two cores need to update x, because one core is assigned a debit processing task and the other a credit processing task. Both cores must have a consistent view of the memory location holding the parameter x. This is the issue of cache coherency. Write invalidate and write update protocols.	CO3	1.3.1 1.4.1 2.2.2 13.1.1 13.1.2
6	K2	A CUDA kernel performs the following operation C[z] = A[y] * B[x] Identify whether the kernel is memory bound or arithmetic bound? Justify your answer ANS: Kernel that fetches two values from memory, multiplies them, and stores the result back to memory has very low arithmetic density. C[z] = A[y] * B[x] The real work being done is the multiplication. With only one operation being performed per three memory transactions (two reads and one write), the kernel is very much memory bound	CO3	1.4.1 2.1.2 13.3.1

Part – B (3×6 = 18 Marks)

7	K3	Explain the role and application of CUDA ballots in GPU programming? Develop a code snippet in which CUDA ballots prove advantageous for coordinating threads?
		ANS:
		_ballot():
		unsigned int __ballot(int predicate);
		This function evaluates the predicate value passed to it by a given thread.
		A predicate, in this context, is simply a true or false value. If the predicate value is nonzero, it returns a value with the Nth bit set, where N is the value of the thread (threadIdx.x).
		This atomic operation can be implemented as
		C source code as follows:
		__device__ unsigned int __ballot_non_atom(int predicate)
		{
		if (predicate != 0)
		return (1 << (threadIdx.x % 32));
		else
		return 0;
		}
		The usefulness of ballot may not be immediately obvious, unless you combine it with another atomic operation, atomicOr.
		The prototype for this is
		int atomicOr(int * address, int val);
		It reads the value pointed to by address, performs a bitwise OR operation (the j operator in C) with the contents of val, and writes the value back to the address.
		It also returns the old value
		It can be used in conjunction with the __ballot function as follows:
		volatile __shared__ u32
		warp_shared_ballot[MAX_WARPS_PER_BLOCK];
		// Current warp number - divide by 32
		const u32 warp_num = threadIdx.x >> 5;
		atomicOr(&warp_shared_ballot[warp_num],
		__ballot(data[tid] > threshold));
		we use an array that can be either in shared memory or global memory, but obviously
		shared memory is preferable due to it's speed.
		We write to an array index based on the warp number, which we implicitly assume here is 32.
		Thus, each thread of every warp contributes 1 bit to the result for that warp.
		For the predicate condition, if the value in data[tid], our source data, is greater than a given threshold.
		Each thread reads one element from this dataset.
		The results of each thread are combined to form a bitwise OR of the result where thread 0 sets (or not) bit 0, thread 1 sets (or not) bit 1, etc.
			CO2	1.4.1
				2.2.2
				13.1.2
				13.3.1

8	K3	Apply the concept of loop fusion for the following code snippet and demonstrate how it improves performance.
		unsigned int i,j;
		a = 0;
		for (i=0; i<100; i++)
		{
		a += b * c * i;
			CO2	1.4.1
				2.2.2
				13.1.1

```
}
d = 0;
for (j=0; j<200; j++)
{
   d+= e * f
}
```

ANS:

loop fusion:

```
void loop_fusion_example_unfused(void)
{
unsigned int i,j;
a = 0;
for (i=0; i<100; i++) /* 100 iterations */
{
a += b * c * i;
}
d = 0;
for (j=0; j<200; j++) /* 200 iterations */
{
D+= e * f
}
}

void loop_fusion_example_fused_01(void)
{
unsigned int i; /* Notice j is eliminated */
a = 0;
d = 0;
for (i=0; i<100; i++) /* 100 iterations */
{
a +=b * c * i;
d += e * f * i;
}
for (i+100; i<200; i++) /* 100 iterations */
{
d += e * f * i;
}
}

void loop_fusion_example_fused_02(void)
{
unsigned int i; /* Notice j is eliminated */
a = 0;
d =0;
for (i=0; i<100; i++) /* 100 iterations */
   {
   a += b * c * i;
   d += e * f * i;
   d += e * f * (i*2);
   }
}
```

we have two independent calculations for results a and d.

The number of iterations required in the second calculation is more than the first.

However, the iteration space of the two calculations overlaps.

		You can, therefore, move part of the second calculation into the loop body of the first, as shown in function loop_fusion_example_fused_01. This has the effect of introducing additional, independent instructions, plus reducing the overall number of iterations, in this example, by one-third.		

9	K3	Apply the concept of loop unrolling for the following loop structure demonstrate how loop unrolling will improve performance in parallel programming.
		for (i=1; i<=1000; i++)
		X[i] = X[i] + S;
		X is an array whose starting address is stored in the register R1
		R2 contains the terminal address of an array x
		S is a constant stored in the register F2
		ANS:
		Loop: LD F0, 0(R1)
		ADDD F4, F0, F2
		SD 0(R1), F4 #1
		
		LD F6, -8(R1)
		ADDD F8, F6, F2
		SD -8(R1), F8 #2
		
		LD F10,-16(R1)
		ADDD F12,F10,F2
		SD -16(R1), F12 #3
		
		LD F14,-24(R1)
		ADDD F16,F14,F2
		SD -24(R1),F16 #4
		
		SUBI R1, R1, #32
		BENZ R1, Loop
		
		Loop: LD F0, 0(R1) 1
		stall 2
		ADDD F4, F0, F2 3
		stall 4
		stall 5
		SD 0(R1), F4 6 ;drop SUBI &BNEZ #1
		LD F6, -8(R1) 7
		stall 8
		ADDD F8, F6, F2 9
		stall 10
		stall 11
		SD -8(R1), F8 12 ;drop SUBI &BNEZ #2
		LD F10,-16(R1) 13
		stall 14
		ADDD F12,F10,F2 15
		stall 16
		stall 17
		SD -16(R1), F12 18 ;drop SUBI &BNEZ #3
		LD F14,-24(R1) 19
		stall 20
		ADDD F16,F14,F2 21
		stall 22
			CO2	1.3.1
				1.4.1
				2.2.2
				13.3.1

		stall 23 SD -24(R1),F16 24 #4 SUBI R1, R1, #32 25 BENZ R1, Loop 26 Stall 27 Loop Unrolling :Without any scheduling • It takes 27 cycles for 4 iterations 27/4 = 6.8 clock cycles per iteration • CPI =6.8/3 =2.2 Instruction cycles Loop: LD F0, 0(R1) 1 LD F6, -8(R1) 2 LD F10,-16(R1) 3 LD F14,-24(R1) 4 ADDD F4, F0, F2 5 ADDD F8, F6, F2 6 ADDD F8, F6, F2 7 ADDD F16, F14, F2 8 SD 0(R1), F4 9 SD -8(R1), F8 10 SD -16(R1), F12 11 SUBI R1, R1, #32 12 BENZ R1, Loop 13 SD 8(R1), F16 14 clock cycles per 4 iterations 14/4 = 3.5 clock cycles per iteration • CPI=3.5/3 =1.16		

Part – C (2×10 = 20 Marks)

10	K2	Explain the approach to error handling in CUDA programming.
		ANS:
		CUDA provides a set of error handling mechanisms.
		The most common macros are:
		cudaGetErrorString:
		Converts a CUDA error code into a human-readable string representation.
		cudaGetLastError:
		Returns the last error that occurred.
		cudaSuccess:
		A special value indicating that CUDA call succeeded.
		cudaError_t cudaStatus;
		cudaStatus = cudaSomeFunction();
		
		if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "CUDA error: %s\n",
		
		cudaGetErrorString(cudaStatus));
		}
		Asynchronous Error Checking:
			CO3	1.3.1
				1.4.1
				2.2.2
				13.3.1
				13.3.2

		When using CUDA APIs that return error codes, you need to ensure that any previous asynchronous operations are completed before checking for errors. you can use cudaDeviceSynchronize() or other appropriate synchronization methods before checking for errors.
		// Example kernel launch
		myKernel<<<gridSize, blockSize>>>(input, output);
		
		// Synchronize the device to ensure the kernel execution is completed.
		cudaDeviceSynchronize();
		// Check for errors after synchronization.
		cudaError_t cudaStatus = cudaGetLastError();
		if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "CUDA error after kernel execution: %s\n", cudaGetErrorString(cudaStatus));
		}
		Error Reporting and Debugging:
		When developing CUDA applications, it's essential to keep track of error messages.
		You can use logging libraries or simply write the error messages to standard output or error streams.
		You can make use of NVIDIA's debugging tools like NVIDIA Nsight or CUDA-GDB to analyze and debug your CUDA code effectively.
		Graceful Resource Management:
		In case of any CUDA error, it's crucial to perform proper cleanup and resource management.
		This involves releasing any allocated memory on the device, resetting the device, and freeing resources before exiting the application.
		// Release resources and clean up
		cudaFree(devicePtr);
		cudaDeviceReset();		

(OR)				
11	K2	Describe the challenges related to algorithmic aspects in CUDA programming. ANS: Thread Divergence: In CUDA, work is divided into threads, and threads are grouped into blocks. When threads within a block follow different execution paths (e.g., if-else statements), it can lead to thread divergence. Thread divergence negatively impacts performance because the GPU needs to execute all the branches taken by different threads, serializing the process Memory Access Patterns: Memory access patterns play a crucial role in GPU performance. Irregular memory accesses, such as non-coalesced reads/writes or random memory accesses, can result in lower memory throughput and increased memory latency. Optimizing memory access patterns can significantly improve the overall performance Workload Imbalance: Load balancing is essential to keep all GPU cores busy. If some threads within a block finish their work earlier than others, they will be idle until all threads have completed their tasks. Identifying and minimizing workload imbalances can lead to better GPU utilization	CO3	1.3.1 1.4.1 2.2.2 13.3.1 13.3.2

		Overhead of Kernel Launches: Kernel launches on CUDA GPUs come with some overhead. It is essential to design algorithms that minimize the number of kernel launches and the data transfers between the host (CPU) and the device (GPU) to achieve better performance. Synchronization: Synchronization points, such as barriers or locks, can introduce performance bottlenecks in GPU computation. Efficiently managing synchronization in CUDA programs is essential to avoid unnecessary stalls and maximize parallelism. Data Dependencies: Dependencies between data elements can hinder parallelism and stall GPU execution. Identifying and minimizing data dependencies through techniques like loop unrolling and loop tiling can improve GPU performance Thread/Block Granularity: Choosing the appropriate thread and block granularity is critical for achieving optimal performance. If the granularity is too fine, the overhead of managing threads and blocks can outweigh the computational benefits. On the other hand, if it's too coarse, some GPU resources may be underutilized. Memory Allocation and Deallocation: Frequent memory allocation and deallocation on the GPU can impact performance. Reusing memory buffers whenever possible can reduce the overhead associated with memory management. Precision of Computations: GPUs often support different precisions (e.g., single-precision and half-precision floating-point arithmetic). Choosing the right precision for computations can impact both performance and numerical accuracy. Scalability: While GPUs offer massive parallelism, not all algorithms are inherently parallelizable. Ensuring scalability of algorithms to fully utilize the available GPU resources is crucial for achieving optimal performance.		
12	K3	Create a CUDA program that employs the Binary Search technique to locate an element within an array. ANS: The search we have two options: a binary search.	CO2	1.4.1 2.1.2 2.2.2 13.3.1 13.3.2

A binary search takes advantage of the fact we have a sorted list of samples from the previous step.

It works by dividing the list into two halves and asking whether the value it seeks is in the top or bottom half of the dataset.

It then divides the list again and again until such time as it finds the value.

The worst case sort time for a binary search is log2(N).

```
#include <iostream>

#include <cstdio>


// Kernel function for binary search

__global__ void binarySearchKernel(int *arr, int target, int left, int right, int *result) {

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

while (left <= right) {

      int mid = left + (right - left) / 2;

      if (arr[mid] == target) {

         atomicExch(result, mid); // Store the result in a thread-safe manner

         return;

      } else if (arr[mid] < target) {

         left = mid + 1;

      } else {

         right = mid - 1;

      }

   }

}

int main() {

   const int arraySize = 1024;

   const int target = 42;
```

```
    int *hostArray, *deviceArray, *deviceResult;


    // Allocate memory on host and device
    hostArray = new int[arraySize];
    cudaMalloc(&deviceArray, arraySize * sizeof(int));
    cudaMalloc(&deviceResult, sizeof(int));


// Initialize the sorted array on the host
    for (int i = 0; i < arraySize; ++i) {
        hostArray[i] = i * 2;
    }
// Copy data from host to device
    cudaMemcpy(deviceArray, hostArray, arraySize * sizeof(int),
cudaMemcpyHostToDevice);


    // Set up grid and block dimensions
    int threadsPerBlock = 256;
    int blocksPerGrid = (arraySize + threadsPerBlock - 1) / threadsPerBlock;
// Launch kernel
    binarySearchKernel<<<blocksPerGrid, threadsPerBlock>>>(deviceArray,
target, 0, arraySize - 1, deviceResult);
// Copy result from device to host
    int result;
    cudaMemcpy(&result, deviceResult, sizeof(int),
cudaMemcpyDeviceToHost);
if (result != -1) {
        std::cout << "Element " << target << " found at index " << result <<
std::endl;
    } else {
        std::cout << "Element " << target << " not found in the array." <<
std::endl;
```

		}
		// Clean up
		delete[] hostArray;
		cudaFree(deviceArray);
		cudaFree(deviceResult);
		return 0;
		}		

(OR)				

13	K3	Develop a CUDA program that populates an array with values ranging from 0 to num_elements. Additionally, set up four streams to run concurrently on four different GPU devices? The objective is to measure the following metrics for each of the four GPU devices:
		• The duration it takes to transfer data from the CPU to the GPU.
		• The time required to execute the kernel operation.
		• The time it takes to copy the results back from the GPU to the CPU
		• The total execution time of the entire operation
		ANS:
		Streams are virtual work queues on the GPU.
		They are used for asynchronous operation i.e, when you would like the GPU to operate separately from the CPU.
		By creating a stream you can push work and events into the stream which will then execute the work in the order in which it is pushed into the stream.
		Streams and events are associated with the GPU context in which they were created.
		void fill_array(u32 * data, const u32 num_elements)
		{
		for (u32 i=0; i< num_elements; i++)
		{
		data[i] = i;
		}
		}
		void check_array(char * device_prefix, u32 * data, const u32 num_elements)
		{
		bool error_found = false;
		for (u32 i=0; i< num_elements; i++)
		{
		if (data[i] !=(i*2))
		{
		printf("%sError: %u %u", device_prefix, i, data[i]);
		error_found = true;
		}
		}
		if (error_found ==false)
		printf("%sArray check passed", device_prefix);
		}
		// Define maximum number of supported devices
		#define MAX_NUM_DEVICES (4)
		// Define the number of elements to use in the array
			CO2	1.4.1
				2.1.2
				2.2.2
				13.3.1
				13.3.2

```
        #define NUM_ELEM (1024*1024*8)
// Define one stream per GPU
        cudaStream_t stream[MAX_NUM_DEVICES];
// Define a string to prefix output messages with so
// we know which GPU generated it
        char device_prefix[MAX_NUM_DEVICES][300];
// Define one working array per device, on the device
        u32 * gpu_data[MAX_NUM_DEVICES];
// Define CPU source and destination arrays, one per GPU
        u32 * cpu_src_data[MAX_NUM_DEVICES];
        u32 * cpu_dest_data[MAX_NUM_DEVICES]
// Generate a prefix for all screen messages
        struct cudaDeviceProp device_prop;
        CUDA_CALL(cudaGetDeviceProperties(&device_prop,
device_num));
        sprintf(&device_prefix[device_num][0], "\nID:%d %s:", device_num,
                                                device_prop.name);

// Create a new stream on that device
        CUDA_CALL(cudaStreamCreate(&stream[device_num]));

// Allocate memory on the GPU
        CUDA_CALL(cudaMalloc((void**)&gpu_data[device_num],
                                        single_gpu_chunk_size));
// Allocate page locked memory on the CPU
         CUDA_CALL(cudaMallocHost((void                                **)
&cpu_src_data[device_num],
                                                single_gpu_chunk_size));
        CUDA_CALL(cudaMallocHost((void
**)&cpu_dest_data[device_num],
                                        single_gpu_chunk_size));

// Fill it with a known pattern
fill_array(cpu_src_data[device_num], NUM_ELEM);

// Copy a chunk of data from the CPU to the GPU  asynchronous
        CUDA_CALL(cudaMemcpyAsync(gpu_data[device_num],
                cpu_src_data[device_num], single_gpu_chunk_size,
                cudaMemcpyHostToDevice, stream[device_num]));
// Invoke the GPU kernel using the newly created  stream - asynchronous
invocation
                gpu_test_kernel<<<num_blocks,
                num_threads,
                shared_memory_usage,
                stream[device_num]>>>(gpu_data[device_num]);

        cuda_error_check(device_prefix[device_num],
                "Failed to invoke gpu_test_kernel");
// Now push memory copies to the host into the streams
// Copy a chunk of data from the GPU to the CPU asynchronous
        CUDA_CALL(cudaMemcpyAsync(cpu_dest_data[device_num],
                        gpu_data[device_num],
                        single_gpu_chunk_size,
                        cudaMemcpyDeviceToHost,
                        stream[device_num]));
```

```
}
// Process the data as it comes back from the GPUs  Overlaps CPU execution
with GPU execution
        for (int device_num=0;device_num < num_devices;device_num++)
{
// Select the correct device
CUDA_CALL(cudaSetDevice(device_num));
//Wait for all commands in the stream to complete
CUDA_CALL(cudaStreamSynchronize(stream[device_num]));
// GPU data and stream are now used, so  clear them up
        CUDA_CALL(cudaStreamDestroy(stream[device_num]));
        CUDA_CALL(cudaFree(gpu_data[device_num]));

// Data has now arrived in  cpu_dest_data[device_num]
        check_array(                                device_prefix[device_num],
cpu_dest_data[device_num], NUM_ELEM);

// Clean up CPU allocations
        CUDA_CALL(cudaFreeHost(cpu_src_data[device_num]));
        CUDA_CALL(cudaFreeHost(cpu_dest_data[device_num]));

// Release the device context
        CUDA_CALL(cudaDeviceReset());
}
}
```

it checks the contents and then frees the GPU and CPU resources associated with each stream.

we need to add some timing code to see how long each kernel takes in practice.

Add events to the work queue.

Now events are special in that we can query an event regardless of the currently selected GPU

we need to declare a start and stop event:

```
// Define a start and stop event per stream
        cudaEvent_t kernel_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_to_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_from_start_event[MAX_NUM_DEVICES];
        cudaEvent_t memcpy_from_stop_event[MAX_NUM_DEVICES];
```

Finally, we need to get the elapsed time and print it to the screen:

```
// Wait for all commands in the stream to complete
        CUDA_CALL(cudaStreamSynchronize(stream[device_num]));

// Get the elapsed time between the copy  and kernel start
        CUDA_CALL(cudaEventElapsedTime(&time_copy_to_ms,memcpy
_to      _start_event[device_num], kernel_start_event[device_num]));

// Get the elapsed time between the kernel start and copy back start
        CUDA_CALL(cudaEventElapsedTime(&time_kernel_ms,
        kernel_start_event[device_num],memcpy_from_start_event[device
        _num]));
Get the elapsed time between the copy back start  and copy back start
        CUDA_CALL(cudaEventElapsedTime(&time_copy_from_ms,memc
py_from_       start_event[de vice_num],
memcpy_from_stop_event[device_num]));

// Get the elapsed time between the overall start and stop events
```

```
        CUDA_CALL(cudaEventElapsedTime(&time_exec_ms,
        memcpy_to_start_event[device_num],
        memcpy_from_stop_event[device_num]));

// Print the elapsed time
        const float gpu_time = (time_copy_to_ms þ time_kernel_ms +
        time_copy_from_ms);
        printf("%sCopy To : %.2f ms",
        device_prefix[device_num], time_copy_to_ms);
        printf("%sKernel : %.2f ms",
        device_prefix[device_num], time_kernel_ms);
printf("%sCopy Back : %.2f ms",  device_prefix[device_num],
                time_copy_from_ms);
        printf("%sComponent Time : %.2f ms", device_prefix[device_num],
                        gpu_time);
        printf("%sExecution Time : %.2f ms", device_prefix[device_num],
        time_exec_ms);
        printf("\n");
When we run the program we see the following result:
ID:0 GeForce GTX 470:Copy To : 20.22 ms
ID:0 GeForce GTX 470:Kernel : 4883.55 ms
ID:0 GeForce GTX 470:Copy Back : 10.01 ms
ID:0 GeForce GTX 470:Component Time : 4913.78 ms
ID:0 GeForce GTX 470:Execution Time : 4913.78 ms
ID:0 GeForce GTX 470:Array check passed

ID:1 GeForce 9800 GT:Copy To : 20.77 ms
ID:1 GeForce 9800 GT:Kernel : 25279.57 ms
ID:1 GeForce 9800 GT:Copy Back : 10.02 ms
ID:1 GeForce 9800 GT:Component Time : 25310.37 ms
ID:1 GeForce 9800 GT:Execution Time : 25310.37 ms
ID:1 GeForce 9800 GT:Array check passed
When we run the program we see the following result:
ID:0 GeForce GTX 470:Copy To : 20.22 ms
ID:0 GeForce GTX 470:Kernel : 4883.55 ms
ID:0 GeForce GTX 470:Copy Back : 10.01 ms
ID:0 GeForce GTX 470:Component Time : 4913.78 ms
ID:0 GeForce GTX 470:Execution Time : 4913.78 ms
ID:0 GeForce GTX 470:Array check passed

ID:1 GeForce 9800 GT:Copy To : 20.77 ms
ID:1 GeForce 9800 GT:Kernel : 25279.57 ms
ID:1 GeForce 9800 GT:Copy Back : 10.02 ms
ID:1 GeForce 9800 GT:Component Time : 25310.37 ms
ID:1 GeForce 9800 GT:Execution Time : 25310.37 ms
ID:1 GeForce 9800 GT:Array check passed
```

Prepared By **PAC Members** **Approved By**

(HOD/CSE)