**CUDA - Assignments**

1. **What is GPU?**

Ans:

1. A graphics processing unit (GPU) is a specialized electronic device designed

to rapidly manipulate and alter memory to accelerate the creation of images

in a frame buffer intended for output to a display device.

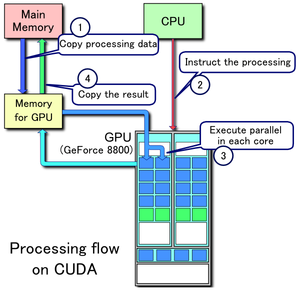
1. It is a programmable logic chip (processor) specialized for display functions.
2. GPUs are located on plug-in cards, in a chipset on the motherboard or in the same chip as the CPU.
3. The GPU renders images, animations and video for the computer's screen.
4. It lets the computer's main processor do its job, allowing the computer to run at full speed.
5. **What is CUDA?**

Ans:

1. CUDA is an abbreviation for Compute Unified Device Architecture.
2. CUDA is a [parallel](http://developer.nvidia.com/cuda/get-started-parallel-computing)computing platform and programming model created by NVIDIA that makes using a GPU for general purpose computing simple and elegant.
3. It allows [software developers](https://en.wikipedia.org/wiki/Software_developer)and [software engineers](https://en.wikipedia.org/wiki/Software_engineer)to use a CUDA-enabled[graphics processing unit](https://en.wikipedia.org/wiki/Graphics_processing_unit) (GPU) for general purpose processing - an approach termed [GPGPU](https://en.wikipedia.org/wiki/GPGPU)(General-Purpose computing on Graphics Processing Units).
4. The CUDA platform is a software layer that gives direct access to the GPU's virtual[instruction set](https://en.wikipedia.org/wiki/Instruction_set)and parallel computational elements, for the execution of [compute kernels](https://en.wikipedia.org/wiki/Compute_kernel).
5. **Explain processing flow in CUDA?**

Ans:

1. CUDA has some specific functions, called kernels.
2. A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads.
3. CUDA also provides shared memory and synchronization among threads.
4. Processing flow in CUDA can be shown diagrammatically as follows:



1. Copy data from main memory to GPU memory.
2. CPU initiates the GPU [compute kernel](https://en.wikipedia.org/wiki/Compute_kernel)
3. GPU's CUDA cores execute the kernel in parallel.
4. Copy the resulting data from GPU memory to main memory.
5. **Explain memory management in CUDA?**

Ans:

1. **Memory Allocation:**

Allocating CPU memory:

1. Allocating memory to be accessed by the CPU is most often accomplished using the C standard library function malloc.
2. An alternative approach is to use CUDA's cudaMallocHostfunction.

Allocating GPU memory:

1. Allocating memory to be accessed by the GPU is most often accomplished using the CUDA function cudaMalloc.
2. There are alternatives, such as cudaMallocPitch and cudaMallocArray.

b) **Memory Deallocation:**

Each memory allocation function has an equivalent deallocation

function: free for malloc, cudaFree for cudaMalloc, and cudaFreeHost for cudaMallocHost.

1. **Explain the concept of grid, block, WARP and thread in CUDA?**

Ans:

1. Thread - This is just an execution of a kernel with a given index. Each thread uses its index to access elements in such that the collection of all threads cooperatively processes the entire data set.
2. Block - This is a group of threads. The threads within a block could execute concurrently or serially and in no particular order.
3. Grid – A group of blocks is called as grid. The blocks in a grid must be able to be executed independently. There’s no synchronization at all between the blocks.
4. WARP - A warp is a unit of thread scheduling in Streaming Multiprocessors. That is, the granularity of thread scheduling is a warp. A block is divided into warps for scheduling purposes.
5. **Explain the following CUDA functions:**
6. Kernel
7. dim3 datatype
8. cudamemcpy
9. cudamalloc
10. cudafree

Ans:

* 1. Kernel - CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA thread.
     + A kernel is defined using the \_\_global\_\_ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax.
  2. Dim3 data type - dim3 is a special CUDA data type with 3 components .x, .y, and .z each initialized to 1.

For example:

* + 1. dim3 gridDim : Dimensions of the grid in blocks (gridDim.z unused below version 2.x)
    2. dim3 blockDim : Dimensions of the block in threads
    3. dim3 blockIdx : Block index within the grid
    4. dim3 threadIdx : Thread index within the block

c. cudamemcpy - It copies count bytes from the memory area pointed to by src to the memory area pointed to by dst.

d. cudamalloc - it allocates size bytes of linear memory on the device and returns in \*devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable.

e. cudafree - It frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc().

1. **Explain the implementation of all the assignments.**

Ans:

**1 A.**

Title: Implement Parallel Reduction using CUDA.

Objective:

1. To implement parallel reduction using CUDA.

2. To analyze time complexity over serial execution of the program.

Outcome:

1. Successful implementation of parallel reduction.

2. Achieve time efficiency.

Problem Statement:

Implement Parallel Reduction using Min, Max, Sum and Average operations.

Theory:

1. Assuming N as the number of the elements in an array, we start N/2 threads, one thread for each two elements.
2. Each thread computes the sum of the corresponding two elements, storing the result at the position of the first one.
3. Iteratively, each step:

a. The number of threads halved.

b. Doubles the step size between the corresponding two elements after some iteration, the reduction result will be stored in the first element of the array.

1. Now, find minimum and maximum elements using respective logic.
2. Take summation of all elements to calculate the sum. Store the sum in resultant sum variable.
3. Divide the resultant sum by total number of elements in an array to find average.

Time complexity:

Total cost of a parallel algorithm is the product of time complexity and the number of processors used in the algorithm.

Total Cost = Time complexity × Number of processors used

Therefore, the efficiency of a parallel algorithm is −

Efficiency = Worst case execution time of sequential algorithm

Worst case execution time of the parallel algorithm

**1 B.**

Title: Implement operations on vector using CUDA.

Objective:

1. To implement vector operations using CUDA.

2. To analyze time complexity over serial execution of the program.

Outcome:

1. Successful implementation of vector operations.

2. Achieve time efficiency.

Problem Statement:

Write a CUDA program that, given an N-element vector, find-

1. The maximum element in the vector
2. The minimum element in the vector
3. The arithmetic mean of the vector
4. The standard deviation of the values in the vector

Test for input n and generate a randomized vector V of length N (N should be large). The program should generate output as the two computed maximum values as well as the time taken to find each value.

Theory:

* Maximum element in the vector -

1. Take input of N elements and generate a randomized vector V of length N.
2. Then compute the maximum value in V on the CPU and on the GPU.
3. Output is two computed maximum values as well as the time taken to find each value.

* Minimum element in the vector -

1. Take input of N elements and generate a randomized vector V of length N.
2. Then compute the minimum value in V on the CPU and on the GPU.
3. Output is two computed minimum values as well as the time taken to find each value.

* The arithmetic mean of the vector -

Take sum of all elements and divide the sum by number of elements.

* The standard deviation of the values of the vector -

For each v in array,

1. Work out the Mean (the simple average of the numbers)
2. Then for each number: subtract the Mean and square the result (the squared difference).
3. Then work out the average of those squared differences. We have calculated Variance here.
4. Take square root of the calculated variance to give standard deviation as output.

Time complexity:

T = n/p

where,

T = total cost

n = number of elements in a matrix

p = number of processors

**1 C.**

Title: Implement vector and matrix operations using CUDA.

Objective:

1. To implement operations on vector and matrix using CUDA.

2. To analyze time complexity over serial execution of the program.

Outcome:

1. Successful implementation of vector and matrix operations.

2. Achieve time efficiency.

Problem Statement:

Vector and Matrix operations -

Design parallel algorithm to

1. Add two large vectors
2. Multiply Vector and Matrix
3. Multiply two N\*N arrays using n2 processors.

Theory:

1. Add two large vectors

i. Take 2 vectors as input.

ii. For parallel processing, divide each vector into partitions based on number of processors.

iii. Perform addition of each corresponding elements.

iv. Copy the result into resultant vectors.

Time complexity:

T = (ts + tw\*m)(n/p)

where

T = total cost

n = number of elements in a matrix

p = number of processors

m = number of messages transferred

ts = starting time

tw = per-word transfer time

1. Multiply Vector and Matrix -

i. The n x n matrix is assigned to an n x n (virtual) processor grid.

ii. The vector is assumed to be on the first row of processors.

Iii. The first step of the product requires a one-to-all broadcast of the vector element along the corresponding column of processors.

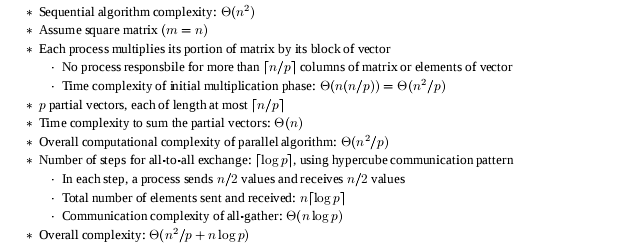
iv. It can be done concurrently for all n columns.

v. The processors compute local product of the vector element and the local matrix entry.

vi. Results of these products are then accumulated on processors in the first column.

vii. Using n concurrent all-to-one reduction operations along the columns.

Time complexity analysis:



1. Multiply two N\*N arrays using n2 processors –
2. Take 2 matrices as input.
3. Loop over all the sub-matrices of A and B required computing the block sub-matrix.
4. Synchronize to make sure the matrices are loaded.
5. Multiply two matrices together; each thread computes one element of sub matrix.
6. Use One-to-all Broadcast communication algorithm to do the multiplication and use All-to-One Reduction communication algorithm to load the resultant one matrix.
7. Synchronize to make sure that the preceding computation is done
8. Write the block sub-matrix to device memory; each thread only writes one element.
9. Load resultant matrix.

Time complexity:

T = n^3/p + ts\*log p + 2tw\* (n^2/sqrt(p)),

where

T = total cost

n = number of elements in a matrix

p = number of processors

ts = starting time

tw = per-word transfer time

**Conclusion:**

Parallel reduction, Vector operations, and Matrix operations are studied and successfully implemented using CUDA interface.