MANIPAL INSTITUTE OF TECHNOLOGY

**Manipal — 576 104**

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

MANIPAL **INSTITUTE OF** YECHNOLOGY

MANIPAL

*{A* r‹»isri/Hrnr i»rir ‹›/ *MAH£, Maitiyal)*

CERTIFICATE

This is to certify that Ms./Mr. .................................................................. Reg. No.

............................. Section: .................. Roll No: has satisfactorily completed

the lab exercises prescribed for Parallel Programming Lab [CSE 3241] of Third Year B. Tech. Degree at MiT, Manipal, in the academic year 2024-2025.

Date:.........................................

Signature Faculty in Charge

Signature

Head of the Department

**CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| LAB NO. | **TITLE** | **PAGE**  NO. |  |
|  | Course Objectives and Outcomes | ii |  |
|  | Evaluation plan | ii |  |
|  | Instructions to the Students | iii |  |
| 1 | Introduction to execution environment of MPI |  |  |
| 2 | Point to Point Communications in MPI | 5 |  |
| 3 | Collective communications in MPI | P |  |
| 4 | Collective communications and Error handling  in MPI | 12 |  |
| 5 | Programs on Arrays in CUDA | 15 |  |
| 6 | Programs on Arrays in CUDA continued. . .. | 24 |  |
| 7 | Programs on Strings in CUDA | 27 |  |
| 8 | Programs on Matrix in CUDA | 32 |  |
| 9 | Programs on Matrix in CUDA continued. . .. | 35 |  |
| lt) | Programs using different CUDA device me O  types and synchronization ’ ’ | 37 |  |
| 11 | References | 41 |  |

Course Objectives

* Learn different APls used in MPI for point to point. collective communications and error handling
* Learn how to write host and kernel code in CUDA for NVIDIA GPU card
* To develop the skills of design and implement parallel algorithms using different parallel programming environment

###### Course Outcomes

At the end of this course, students will be able to

* Write MPI programs using point-to-point and collective communication primitives.
* Design and implement CUDA programs for different parallel applications on NVIDIA GPU architecture.
* Design optimized parallel solutions using thread and memory organization in CUDA.

Evaluation plan

* Internal Assessment Marks : 609a

S Continuous Evaluation and Mid-term exarriination: 6f)\*/o Continuous evaluation component (for each evaluation): 10 marks

The assessment will depend on punctuality, iegularity, program execution, maintaining the observation notebook.

o End semester assessment of 2 hours duration: 4t) 9

INSTRUCTIONS TO THE STUDENTS

**Pre- Lab Session Instructions**

1. Students should carry the Lab Manual Book and the required stationeiy to every lab

session.

1. Be in time and follow the institution dress code.
2. Must Sign in the log register provided.
3. Make sure to occupy the allotted seat and answer the attendance
4. Adhere to the rules and maintain the decorum.
5. Students must come prepared for the lab in advance.

In- Lab Session Instructions

* + Follow the instructions on the allotted exercises.
  + Show the program and results to the instructors on completion of experiments.
  + On receiving approval from the instructor, copy the program and results in the Lab record
  + Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

* + Implement the given exercise individually and not in a group.
  + Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
  + The exercises for each week are divided under three sets: H Solved example

H Lab exercises - to be completed during lab hours

4 Additional Exercises - to be completed outside the lab or in the lab to enhance the skill

o ln case a student misses a lab class, he/ she inner ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).

* Questions for lab tests and examination are not necessarily limited to the questions in the manual, bul may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

* Bring mobile phones or any other electronic gadgets to the lab.
* Cio out of the lab without permission.

**Lab** No 1:

###### Objectives:

Date:

### Ltroduction to execution environment of MPI

In this lab, student will be able to

1. Understand the execution environment of MPI progrnms
2. Learn the various concept of parallel programming
3. Learn and use the Basics API available in MPI

**I. Introduction**

In order to reduce the execution time work is carried out in parallel. Two types of parallel programming are:

* Explicit parallel programming
* Implicit parallel progr;imming

**Explicit pamllel programming** — These are languages where the user has full control and has to

explicitly provide all the details. Compiler effort is minimal.

**Implicit parallel programming** — These are sequential languages where the compiler has full responsibility for extracting the parallelism in the program.

**Parallel Programming Models:**

* Message Pansing Programming
* Shared Memory Programming

**Message Passing Programming:**

* In message passing programming, progr.mmers view their programs (Applications) as a collection of co-operating processes with private (local) variables.
* The only way for an application to share data among processors is for programmer to explicitly code commands to move data from one processor to another.

**Message Passing Libraries:** There are two message passing libraries available. They are:

* PVM — Parallel Virtual Machine
* MPI — Message Passing Interface. lt is a set of parallel APls which can be used with languages such as C and FORTRAN.

Communicators and Groups:

* MPI assumes static processes.
* All the processes are created when the program is loaded.
* No process can be created or terminated in the middle of program execution.
* There is a default process group consisting of all such processes identified by MPI\_COMM\_WORLD.

**III. MPI Environment Management Routines:**

**MPI Init:** Initializes the MPI execution environment. This function must be called in every MPI program. must be called before any other MPI functions and must be called only once in an MPI program.

**MPI\_Init (&argc,&argv),**

**MPI Comm** size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI\_COMM\_WORLD.

**MPI\_Comm\_sizetComm,&size),**

**MPI** Comm **rank:** Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPl\_COMM\_WORLD. This rank is often referred to as a process ID.

**MPI\_Comm\_rank Comm,&ranh),**

MPA **Finalize:** Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

I i 0

Solved Example:

Write a program in MPI to print total number of process and rank of each process.



#include "mpi.h"

#include <stdio.h>

int main(int argc. char \*argv{])

int rank,size;

mi\_c \_rm(mi\_co

MPI\_Init(large,&argv);

\_woaro,+«n›;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); printf("My rank is 9od in total 9‹d processes", rank, size); MPI\_Finalize();

renim 0:

Steps to execute MPI program is provided in the form of video wbich is available in individual systems. However, the basic installation steps are given as follows.

// Follow the following steps to Install. Compile and Run MPI programs in Ubuntu O.S

// To install MPI in Ubuntu, execute the following command in command line

$sudo apt-get update; sudo apt-get install mpich

// To edit MPI program, use any text editor such as vim or gedit and create a file with .c extention.

//To Compile MPI program, execute the following command in command line

$mpicc filename.c •o filename.out

/fTo Run MPI program, execute the following command in command line

$mpirun -np 4 Jfilename.out

Lab Exercises:

1. Write a simple MPI program to find out pow (x, rank) for all the processes where ‘x’ is the integer constant and ‘rank’ is the rank of the process.
2. Write a program in MPI where even ranked process prints “Hello” and odd ranked process prints “World”.
3. Write a program in MPI to simulate simple calculator. Perform each operation using different process in parallel.
4. Write a program in MPI to toggle the character of a given string indexed by the rank of the process. Hint: Suppose the string is HELLO and there are 5 processes, then process 0 toggle ‘H’ to ‘h’, process 1 toggle ‘E’ to ‘e’ and so on.
5. Write a program in MPI where even ranked process prints factorial of the rank and odd ranked process prints ranks Fibonacci number.

Additional Exercises:

1. Write a program in MPI to reverse the digits *or* the following integer array of size 9 with 9 processes. lnitialize the arrny to the following values.

Input array: 18. 523, 301, 1234, 2, 14. 108, 150, 1928

Output array: 8 l, 325, 103, 4321, 2, 41, 801, 5 l. 8291

1. Write a MPI program to find the prime numbers between I and 100 using two processes.

**Lab** No 2:

###### Objectives:

Date:

**Point to Point Communications in MPI**

In this lab, student will be able to

1. Understand the different APIs used for point to point communication in MPI
2. Learn the different modes available in case of blocking send operation

#### Point to Point communication in MPI

* MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
* MPI provides both blocking and non-blocking send and receive operations.

#### Sending message in MPI

* **Bloched** Send sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send.**
* **Send** sends a message and continues without waiting. Also called as Asynchronous send. There are multiple communication modes used in blocking send operation:

P **Standard mode**

h **Synchronous mode**

¥ **Buffered mode**

###### Standard mode

This mode blocks until the message is buffered.

**MPI\_Send(&Msg, Count, Datatype, Destination, Tag, Comm),**

* First 3 parameters together constitute message buffer. The Msg could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The Datatype specifies the message type. Some Data types available in MPI tire: MPI INT, MPI FLOAT, MPI CHAR, MPI DOUBLE, MPI LONG
* Next 3 parameter.s specify message envelope. The Destination specifies the rank of the process to which the message is to be sent.
* **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.
* **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of context and group are combined in a single object called a communicator **(Comm).**
* The default process group is MPI\_COMM\_WORLD.

##### Synchronous mode

This mode requires a send to block until the corresponding receive has occurred.

**MPI\_Ssend(&Msg, Count, Datatype, Destination, Tag, Comer);**

##### Buffered mode

**MPI\_Baend(&Msg, Count, Datatype, Destination, Tag, Comm),**

In this mode a send assumes availability of a certain amount of buffer space. which must be pre- viously specified by the user program through a routine cal! that allocates a user buffer.

**MPI-Buffer attach(buffer, size),**

This buffer can be released by

**MPI\_Buffer\_detach(\*buffer, \*size);**

**Receiv'•Rmessage in MT'I**

**MPI\_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status),**

* Receive a message and block until the requested data is available in the application buffer in the receiving task.
* The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains fol- lowing information: Sender's rank, Sender's tag and number of items received

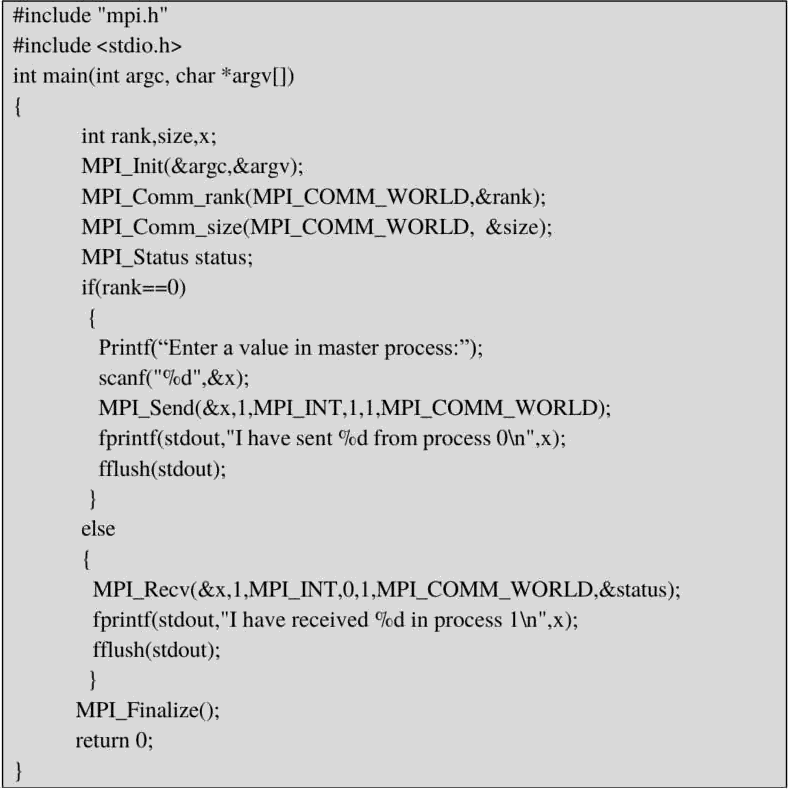
#### Finding execution time in MPI

**MPI Wtime:** Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

6

* MPI\_Wtime () Solved Example:

Write .i MPI yrogrum using standard tool. Thc sender proccss sends a nuinhcr to the rcccivcr. The second process receives the nunzbcr and prints it.



Lab Exercises:

l) Write a MPI pro r:in usinyp synchronous send. The sender process sends a word to this‘ receiver. The second process receives the w'ord, togg•1es e•ich lettei of the word and sends it back to the tii'st pi'ocess. Both processes use synchi'onous send operations.

1. Wi‘ilc a M PI program where the rn‹istcr proccss (proccss 0) sends a nunibcr to cach of the

Slilves itifld the S1iix'c processe s receive thc number arid prints it. Use stiind‹ird send.



1. Write a MPI program to read N elements of the array in the root process (process 0) where N is equttl to the total number of processes. The root process sends one value to each of the slaves. Let even ranked process finds square of the received element and odd ranked process finds cube of received element. Use Buffered send.
2. Write a MPI program to read an integer value in the root process. Root process sends this value to Process 1, Process 1 sends this value to Process2 and so on. Last process sends the value back to root process. When sending the value each process will first increment the received value by one. Write the program using point to point communication routines.

Additional Exercises:

1. Write a MPI program to read N elements of an array in the master process. Let N processes including master process cheek the array values are prime or not.
2. Write a MPI program to read value of N in the root process. Using N processes, including root, find out I ! + ( 1+2 ) +3! +( I +2+3+4 ) **+5! +(1 +2+3+4+5+6)** and print the result in the root process.
3. Implement at least 2 programs to identify deadlock conditions in synchronous send and standard send with multiple point to point communications between two processes

Lab No 3: Date:

Collective Communications in MPI

Objectives:

In this lab, student will be able to

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APls for distributing values from root and gathering the values in the

Collective Communication routines

When **all processes** in a group participate in a global communication operation, the resulting communication is called a collective **communication.**

MPI\_Bcast:

**MPI\_Bcast (Address, Count, Datatype, Root, Comm},**

The process ranked Root sends the same message whose content is identified by the triple (Ad- dress,Count,Datatype) to all processes(including itself) in the communicator **Comm.**

MPI\_Scatter:

**MPI\_Scattert** SendBuff, Sendeount, **SeadDatatype, RecvBuff, Recvcount, RecvDatatype, Root, Comm);**

Ensures that the **Root** process sends out personalized messages, which are in rank order in its send buffer, to all the N processes (including itself).

MPI\_Gather:

**MPI\_Gather(SendAddress,Sendcount, SendDatatype, RecvAddress, RecvConnt, RecvDatatype, Root, Comm);**

The root process receives a personalized message from all N processes. These N received mes- sages are concatenated in rank order and stored in the receive buffer of the root process.

Total Exchange:

In routine **MPI\_Alltoall()** each process sends a personalized message to every other process in- cluding itself. This operation is equivalent to N gathers. each by a different process and in all N' messages are exchanged.

Solved **Example:**

Write a MPI program to read N values of the array in the root process. Distribute these N values among N processes. Every process finds the square of the value it received. Let every process return these values to the root and root process gathers and prints the result. Use collective communication routines.



#include "mpi.h" #include <stdio.h>

int main(int argc, char \*argv[])

int rank,size,N,A[10],B[10], c, i;

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if(rank 0)

N=size;

fprintf(stdout,"Enter °fod values:\n”,N): fflush(stdout);

for(i=0; i<N; i++) scanf{’%d”,&A[i]),

MPI\_Scaner(A,l,MPIplNT,&c,l,MPI\_lNT,0,MPIpCOMM\_WORLD);

fprintf(stdout,"1have received God in process 9od\n",c,rank); fflush(stdout);

c=c\*c;

MPI\_Gather(&c,1,MPI\_lNT,B, l,MPI\_INT,0,MPI\_COMM\_WORLD);

fprintf(stdout,"The Result gathered in the root \n”); ff1ush(stdout);

for(i=0; i<N; i++) fprintf(stdout,"%d \t”,B[i]); ff1ush(stdout);



**Lab Exercises:**

1. Write a MPI pro¿warn to read N v‹ilues in the root process. Root pi occss sends oile X'itlue to each process. Every process receives it and finds the fiictoriill 01‘ thai number and returns it to the matt process. Root process fathers the t‘iictoriill :tnd finds sum oi’ it. Use N number ct’ processes.

 Wi ite ii MPI progi am to read itil integer value M iind NXM eleiiients into an 1 D iiri iiy iii the i oot pi ocess, whci e N ix the **number** ofi |›rnce .ses. Ront |3r‹›ce s sends M elements to each grncess. Euch process kinds ax'crage nt M elements it ieccived and sends these a›'er‹i•e values to ront. Root collcctx .ill the vaiucs und finds the tot.tl nvci ‹igc. Use collective cnmintmic‹itinn routincx.

 Wi ite ‹› MPI pinkranl to mud a string. Using N }3J”(3Cos es T ti”ii›g length is evei Iy di i ihlc fry N›. find the number of non-x’ou'cls in the xtrin\*. In the loot process pi int number of non- x’ov.’cls f”otinJ by cach precise and print the total number ot wow-vowcls.

1. Wi ice a MPI Pi ngl ‹tn to read twn strin•s S i und S2 of same length in the root process. Usin• *N yrs esses* including• the ront Istriny length is evenly divisible by N). pinducc the rcsultant strin\* zs shown hclow. Displ by thc resultant string in thc root process. Use Collective conJlllunication i”outinc .

Extn plc:

String S I: sti5ny Siring S2: 1cqgth Resultant String : slicmipni¿h

**Additional Exercises:**

* 1. Wi ite a MPI progriuii to read a value M and NXM number ot’ elements int‹› 1 D ari ay in the root, where N is the total number ot processes. Find the square of fii st M numbers. the cube of next M nuiiibcl's and so on. Pri nt the rcsults in thc root.

21 W rite a MPI progriim usingu collective communication functions, to rcplacc all even eleiiients of ai’r.iy A to I and i'eplacc .it I odd elements to (1 ot size N. Display the resultant array A, count of all cven and odd numbcrs in root process. Assume N is c venly divisible by number of i'ocesses.

Exam plc

Input Array (A): 1 2.3.4.5.6.7 S 9

Resultant Arriiy (A): t) 1.0.1 t) I (1 1 () Even (Count) — 4

OJJ (C‹›unt) = 3

Lab No 4: Date:



Collective Communications and Error Handlin in MPI

Objectives:

ln this lab, student will be able to

1. Understand the different aggregate functions used in MPI
2. Learn how to write MPI programs using both point to point and collective

communication routines

1. Learn and use the APIS for handling errors in MPI
2. Aggregation **Functions**

MPI provides two forms of aggregation

R Reduction

**Reduction:**

**MPI\_Reduce (SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm),**

This routine reduces the partial values stored in SendAddress of each process into a final result and stores it in RecvAddress of the Root process. The reduction operator is specified by the Op field. Some of the reduction operator available in MPI are: MPI SUM, MPI MAX, MPI MIN, MPI\_PROD

Scan:

**MPI\_Scan (SendAddress, RecvAddress, Count, Datatype, Op, Comm};**

This routine combines the partial values into N final results which it stores in the RecvAddress of the N processes. Note that root field is absent here. The scan operator is specified by the Op field. Some of the scan operator available in MPI are: MPI SUM, MPI\_MAX, MPI\_MlN, MPI\_PROD

**MPI\_Barrier(Comm)** :This routine synchronizes all processes in the communicator **Comm.**

They wait until all N processes execute their respective MPI\_Barrier.

Note: All collective communication routines except MPI\_Barrier, employ a standard blocking mode of poinl-to-point communication.

Error **Handling in MPI:**

* An MPI *communicator i s* more than just a group of process that belong to it. Amongst the items that the communicator hides inside is an error *handler.* The error handler is called every time an MPI error is detected within the communicator.
* The predefined default error handler, which is called **MPI ERRORS\_ARE FATAL,** for a newly created communicator or for MPI\_COMM\_WORLD is to ‹i6‹rrr *the wltole parallel P^«•x•uin* as soon as any MPI error is detected. There is another predefined error handler, which is called **MPI\_ERRORS\_RETURN.**
* The default error handler can be replaced with this one by calling function **MPI\_Er•**

rhandler\_set, for example:

**MPI\_Errhandler set(MPI COMM\_WORLD, MPI\_ERRORS\_RETURN),**

* The only **error code** that MPI standard itself defines is **MPI\_SUCCESS,** i.e.. no error. But the meaning of an error code can be extracted by calling function **MPI\_Error\_string.** On top of the above MPI standard defines the so called *error classes.* The **error class** for a given error code can be obtained by calling function **MPI\_Error\_class.**
* Error classes can be converted to comprehensible error messages by calling the same function that does it for error codes, i.e., **MPI\_Error\_string.** The reason for this is that error classes are implemented as a subset of error codes.

Solved Example:

Write a MPI program using N processes to find l ! + 2! +. +N!. Use collective communication

routines.

#include <stdio.h> #include ”mpi.h"

int main(int argc, char\* argv[])

int rank,size,fact=l, factsum, i;

MPI\_Init(&argc,&argv)i MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank); MP1\_Comm\_size(MPl\_COMM\_WORI-D, &size);

forti=l; i<=rank+1; i++) fact = fact \* i;

MPI\_Reduce (&fact,&factsum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD); if(rank= )

printf(”Sum of all the factorial=9od",factsum);

MPI\_Finalize();

exit(0);

###### Lab Exercises:

l) Write a MPI program using N processes to find I ! + 2! +. +N!. Use scan. Also, handle

different errors using error handling routines.

* 1. Write a MPI program to read a 3 X 3 matrix. Enter an element to be searched in the root process. Find the number of occurrences of this element in the matrix using three processes.
  2. Write a MPI program to read 4 X 4 matrix and display the following output using four processes.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ilp matrix: 1 | 2 | 3 | 4 | O/p matrix: | 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 1 |  | 2 | 4 | 6 | 5 |
| 1 | 1 | 1 | 1 |  | 3 | 5 | 7 | 6 |
| 2 | 1 | 2 | 1 |  | 5 | 6 | 9 | 7 |

* 1. Write a MPI program to read a word of length N. Using N processes including the root get output word with the pattern as shown in example. Display the resultant outpui word in the root.

Example: Input : PCAP Output : PCCAAAPPPP

**Additional Exercises:**

**Lab No 5:**

**Objectives:**

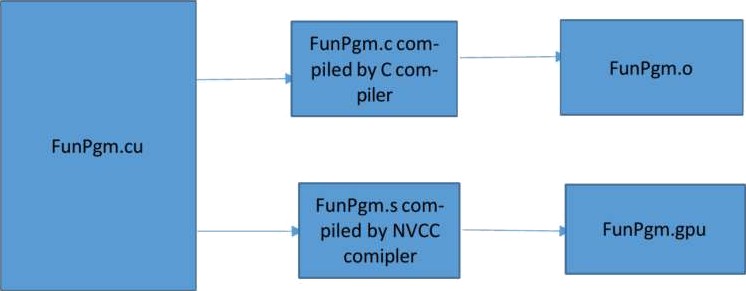
# Programs on arravs in CUDA

Date:

In this lab, student will be able to

1. Know the basics of Computing Unified Device Architecture (CUDA).
2. Learn program structure of CUDA.
3. Learn about CUDA 1D blocks and threads
4. Write simple programs on one dimensional array
5. Learn mathematical functions in CUDA

About CUDA: CUDA is a platform for performing massively parallel computations on graphics accelerators. CUDA was developed by NVIDIA. It was first available with their G8X line of graphics cards. CUDA presents a unique opportunity to develop widely-deployed parallel applications. The CUDA programs are compiled as follows.



FunPgrn.cu is compiled by both C compiler and Nvidia CUDA C compiler (NVCC compiler). If you have both main.c and Funpgm.cu then you can call cuda API's in main.c but keep in mind that you cannot call kernel from main,e. To call the kernel file extension must be .eu.

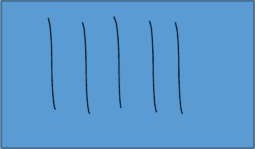
As in OpenCL CPU is the host and its memory the host memory and GPU is the device and its memory device memory. Serial code will be run on host and parallel code will be run on device.

* 1. Copy data from host memory to device memory.
  2. Load device program and execute, caching data on chip for performance.
  3. Copy result from device memory to host memory. CUDA threads, blocks and grid

Thread — Distributed by the CUDA runtime. A single path of execution there can be multiple threads in a program.

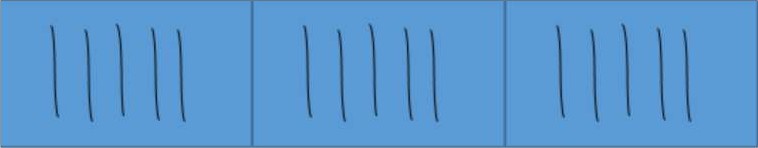
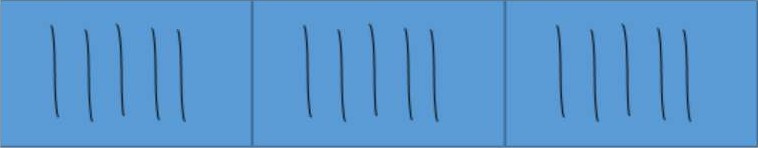
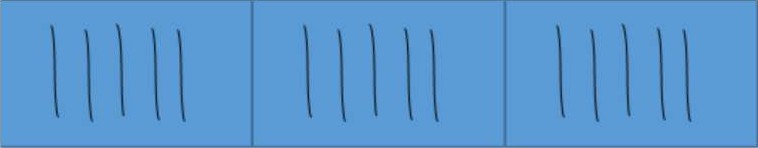
(identii'ied by threadldx)

CUDAThread

Bloch — A user defined group of I to 512 threads. (identified by blockldx)

CUDA Block

Grid — A group of one or more blocks. A grid is created for each CUDA kernel function

 CUDA GRID

de\_vi ce

Some of lhe calculations for indexing the thread is given below.

de\_vi ce

1D grid of lD blocks

int getGloba1Idx\_1D\_l D(){

return blockIdx.x \*blockDim.x + threadIdx.x;

lD grid of 2D blocks device

int getGloba1Idxp1D\_2D(){

return blockIdx.x \* blockDim.x \* blockDim.y

+ tbreadIdx.y \* b1ockDim.x + tbreadldx,x;

lD grid of 3D blocks

int getGlobaIIdx\_1D\_3D(){

return blockldx.x \* blockDim.x \* blockDim.y \* blockDim.z

+ threadIdx.z \* blockDim.y \* blockDim.x

+ threadIdx.y \* blockDiin.x + thieadldx.x;

2D grid of ID blocks

device int getGlobalIdx\_2D\_lD()|

int blockld = bloekIdx.y \* gridDim.x + blockIdx.x; int threadld = blockld \* blockDim.x + threadIdx.x;

de\_vi ce

de\_vi ce

de\_vi ce

return threadld;

2D grid of 2D blocks

int getGloballdx\_2D\_2D(){

int blockld = blockIdx.x + blockIdx.y \* gridDim.x; int threadId = blockld \* (blockDim.x C blockDim.y)

+ (threadIdx.y \* blockDim.x) + threadIdx.x; return threadld;

2D grid of 3D blocks

int getGloba1Idx\_2D\_3D(){

int blockId = blockIdx.x + blockIdx.y \* gridDim.x;

int threadld = blockld \* (blockDim.x \* blockDim.y \* blockDim.z)

+ (thread1dx.z \* (blockDim.K \* b1ockDim.y))

+ (threadIdx.y \* blockDim.x) + threadIdx.x;

return threadId;

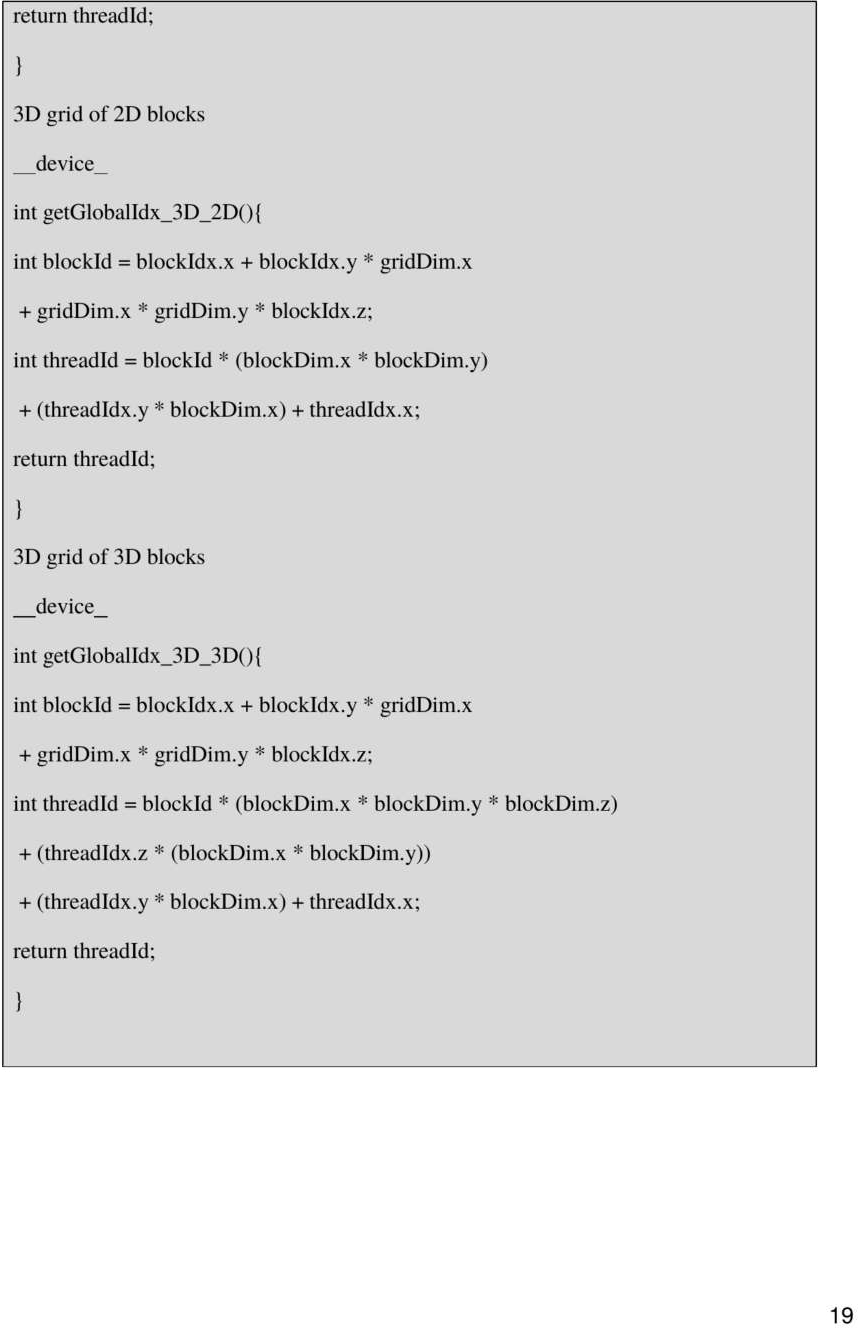
3D grid of lD blocks

int getGloba1Idx\_3D\_l D(){

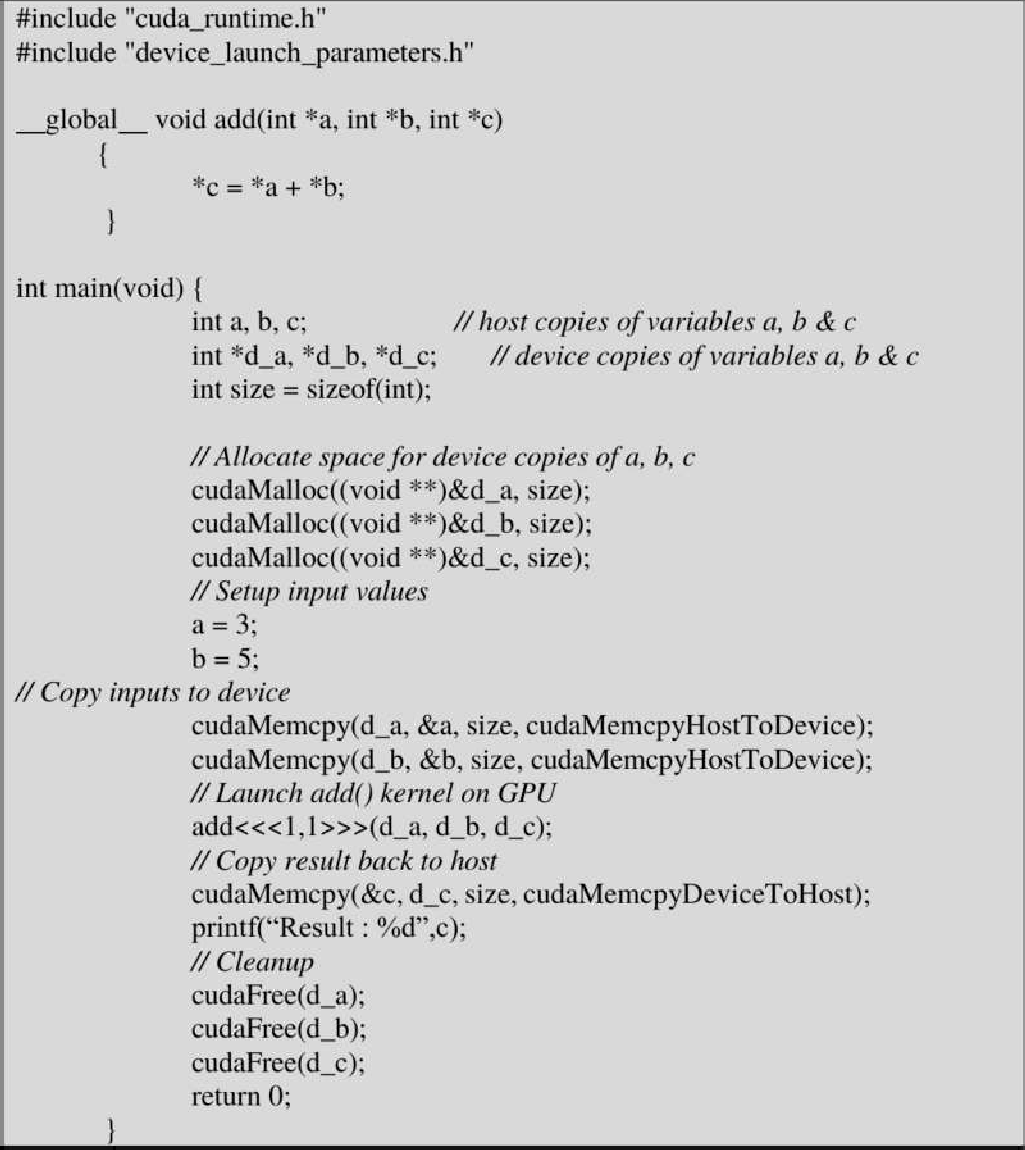
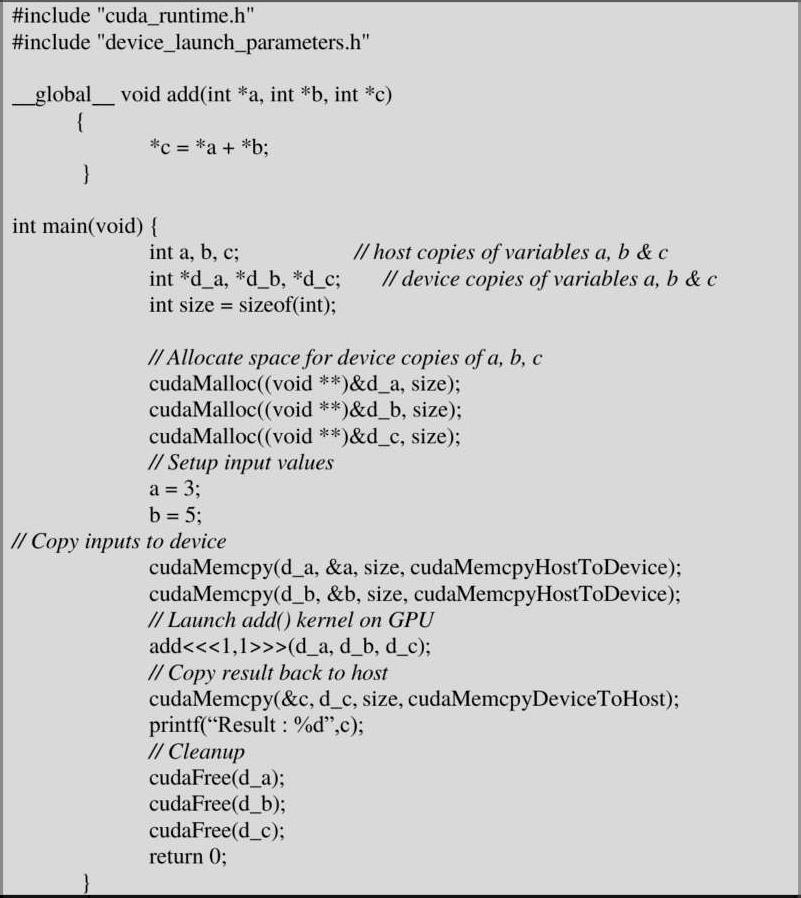
int blockld = blockldx.x + blockldx.y \* gñdDim.x

+ gridDim.x \* gridDim.y \* blockIdx.z;

int threadld = blockld \* blockDim.x + tbreadldx.x;



Solved E xcrcisc: Program to add two nunibcrs.



Explanation:

add is the ł'unction whic h runs on device.



cudaMalloc will allocate memory of size bytes given as second argument to variable passed aS first argument.

ctidaMemcpy (Destination,Source,Size,Direction); cudaMemcpy (d\_a, &a, size, cudaMemcpyHostToDevice); cudaMemcpy (&c, d\_c, size, cudaMemcpyDeviceToHost);

cudaMemcpy copies the variables from host to device or device to host based on the direction which is either eudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Value of size bytes long is copied from source to destination.

cudaFree frees the memory allocated by eudaMalloc.

The add function is called like this add< << 1.1 >>>(d\_a,d\_b,d\_c). The add is followed by three singular brackets then the number of blocks, threads per block then corresponding closing angular brackets then how many arguments the function add takes is enclosed within parenthesis. If you want to add N elements you can achieve it in two ways either having N blocks or having N threads.

That is pass an array with following function calls

add<<< N,1 >>> (d a,d b,d\_c) or add< << 1, N>>> (d a,d b,d c)

Few Mathematical functions in CUDA:

**Major Single-Precision floating point functions:** Single precision functions work on float value (32 bit). A float value is stored in IEEE 754 format.

|  |  |
| --- | --- |
| Function | **Description** |
| sqrtf(x) | Square root function |
| expf(x) | Exponentiation function. Base = e |
| exp2f(x) | Exponentialion function. Base = 2 |
| exp l0f(x) | Exponentiation function. Base = 10 |
| logf(x) | Logarithmic function. Base=e |
| log2f(x) | Logarithmic function. Base=2 |
| log10f(x) | Logarithmic function. Base= 10 |
| sinf(x) | sine function |

|  |  |
| --- | --- |
| cost(x) | cos function |
| tanf(x) | tan function |
| powf(x,y) | power function |
| truncf(x) | tnincation function |
| roundf(x) | round function |
| ceilf(x) | ceil function |
| floorf(x) | floor function |

Major Double-Precision floating point functions: Double precision functions work on double value (64 bit). A double value is stored in IEEE 754 format.

|  |  |
| --- | --- |
|  | Dmcription |
| sqrt(x) | Square root function |
| exp(x) | Exponentiation function. Base = e |
| exp2(x) | Exponentiation function. Base = 2 |
| exp10(x) | Exponentiation function. Base = 10 |
| log(x) | Logarithmic function. Base=e |
| log2(x) | Logarithmic function. Base=2 |
| Iogl0(x) | Logarithmic function. Base=10 |
| sin(x) | sine function |
| cos(x) | cos function |
| tan(x) | tan function |
| pow(x,y) | power function |
| trunc(x) | truncation function |
| round(x) | round function |

|  |  |
| --- | --- |
| ceil(x) | ceil function |
| floor(x) | floor function |

**Steps to** execute **a CUDA program is provided** in the **form of** video which **is made available**

in individual systems.

Lab Exercises:

1. Write a program in CUDA to add two vectors of length N using

a) block size as N b) N threads

1. Implement a CUDA program to add two vectors of length N by keeping the number of threads per block as 256 (constant) and vary the number of blocks to handle N elements.
2. Write a program in CUDA to process a ID array containing angles in radians to generate sine of the angles in the output array. Use appropriate function.

Additional Exercises:

1. Write a program in CUDA to perform linear algebra function of the form y = cx + y, where x and y are vectors and o is a scalar value.
2. Write a program in CUDA to sort every row of a matrix using selection sort.
3. Write a program in CUDA to perform odd even transposition sort in parallel.

**Lab No** 6:

###### Objectives:

Programs on Arrays in CUDA

Date:

In this lab, student will be able to

1. Learn more about CUDA lD blocks and threads
2. Write parallel program applications on one dimensional urray
3. Learn parallel sorting applications in CUDA

lD Sequential Convolution

Convolution is a popular array operation that is used in various forms in signai processing, digital recording, image processing, video processing. and computer vision.

Convolution is often performed as a filter that transforms signals and pixels into more desirable values. For example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

Mathematically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.

The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution mask OR convolution kernel.

The same convolution mask is typically used for aiI elements of the array.

The following example shows a convolution example for lD data where a five-element convolution mask array M is applied to a seven-element input array N.



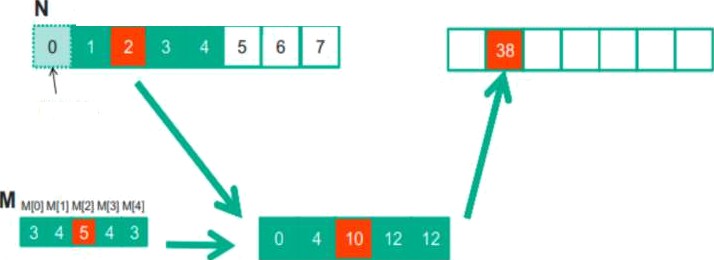
The fact that we use a five-element mask M means that each P element is generated by a weighted sum of the corresponding N element. up to two elements to the lefi and up to two elements to the right.

Each weight value is multiplied to the corresponding N element values before the products are summed together.

In genera), the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated.

Because convolution is defined in terms of neighbouring elements, *buun kms* c'oridiritins naturally exist for output elements that are close to the ends of an array.

For example. when we calculate P|1]. there is only one N element to the left of N[1]. That is, there are not enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such a boundary condition is to define a default vaiue to these missing N elements. For most applications, the default value is 0.

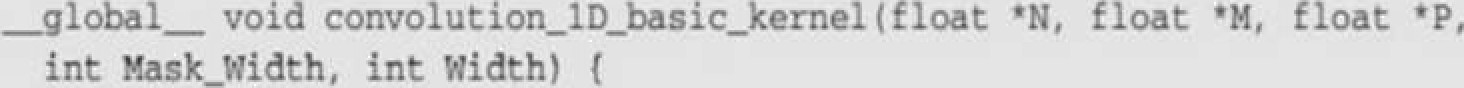


N[0) H[1] N(2] N|3] N|4] NHS] N|6]

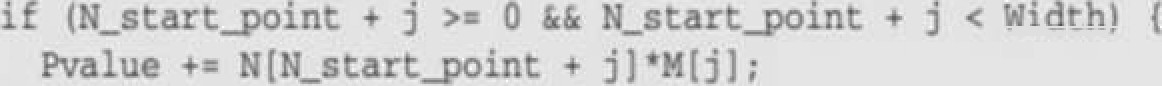
P[0] P[1] P{2] P|3] P[4] I jS] P(6]

Fillad in

* These missing elements are typically referred to as *pIi‹›s’t elements'* in literature.
* The calculation of all output *(P)* elements can be done in pamllel in a ID convolution.
* The first step is to define the major input parameters for the kernel. We assume that the ID convolution kernel receives five arguments: pr›inrer to *ioptif orra¡ N, p‹›inter to iiiytii inn.sk* M, *pointer tn* rniryaiir *arr i) P, si<-e of the mask Mask Width,* and *site of the input rind* oiiryiir *urrcijs Width.* Thus, we have the following set up:
* The second step is to determine and implement the mapping of threads to output elements. Since the output array is one dimensional, a simple and good approach is to organize the threads into a lD grid and have each thread in the grid calculate one output element.

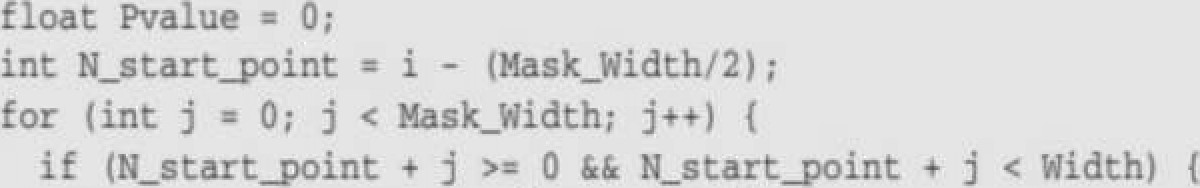








 elobal void umovolutioo\_1D\_basic\_kmnw1(float MB, f\*-# \*a, float \*P,



° •• •• cis-• <t-.e• t • fI”•tfl:

 Write a program in CUDA which performs convolution operation on one-dimensional input array N of size *width* using a mask array M of size *mask\_width* to produce the resultant one- dimensional array P of size *width.*

1. Write a program in CUDA to perform selection sort in parallel.
2. Write a program in CUDA to perform odd even transposition sort in parallel.

**Additional Exercises:**

* 1. Write a program in CUDA which takes N integers as input. It Converts these integers into their corresponding octal values and stores the result in another array in parallel.
  2. Write a **CUDA** program which takes N binary numbers as input and stores the one's complement of each element in another array in parallel.

Lab No 7:

## Programs on strings in CUDA

Date:

Objectives:

In this lab, student will be able to

1. Write simple programs on Strings
2. Learn to compute time of kernel execution
3. Learn about atomic functions
4. Learn to handle errors in the kernel

Arithmetic functions:

In a multithreaded scenario, the issue of data inconsistency will arise, if multiple threads modify a single shared memory variable. To overcome this, atomic functions need to be used. List of atomic functions, their syntax and explanation are provided below.

atomicAddf):



**int atomicAdd (int• address, Int val),**

ftogt atomicAdd(ftoat• address, float val),

double atomicAdd(double\* address, double vat);

Reads the 16-bit, 32-bit or 64-bit word old located at the address in globai or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicSub(1:



**unsigned int atomicSub(unsigned** int\* **address, unsigned** int val),

Reads the 32-bit word old located at the address in global or shared memory. computes, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

**atomicEnch(1:**

**int atomicExch(int\* address, int val),**

**unsigned int atomicExch(unsigned int\* address, unsigned int** val);

**float atomlcExch(float\* address, float** val),

Reads the 32-bit word old located at the address address in g!oba1 or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic trans- action. The tunction returns old.

atomicMinf):

**int atotnicMin(Int\* address, int val);**

**unslgned** Int **atonilcMln(unsigned int\* address, unsigned int val),**

Reads the 32-bitword old located at the address address in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.



**unsigned int atomicMax(unsigned int• address, unsigned int val},**

Reads the 32-bit word old located ai the address address in global or shared memory. computes the maximum of old and val, and xtores the result back to memory at the same address. These three operations are performed in one atomic transaetion. The function returns old.

atomicInc():

**unsigned int atomie1nc(unsigned ints address, unsigned Int val);**

Reads the 32-bit word old located at the address address in global or shared memory, computes I(old >= va1) ? 0 : (o1d+ l)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns oid.





Reads the 32-bil word old located at the address address in global or shared memory. computes (((old 0) I(old > val)) ? val : (old-1) ), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

Solved Example:

A CUDA program which take.s a string as input and determines the number of occurences of a character 'a’ in the string. This program uses atomicAdd() function.

#include "cuda\_nintime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h> #include <std1ib.h> #include<string.h> #include <conio.h> #define N 1024

\_global\_ void CUDACount(char• A, unsigned int \*d\_count){ int i = threadIdx.x;

ii(A[i] ’a’)

atomicAdd(d\_count.1);

int main() { char A[N]; char \*d A;

unsigned int \*count=0,\*d\_count,\*result; printf(“Enter a string");

gets(A);

cudaEvent\_t start, stop, eudaEventCreate(&start), cudaEventCreatet stop), cudaEventRecord(start, 0);

ciidaMalloc((void\*\*)&d\_A, strlen(A)\*sizeof(char)); eudaMalloc((void \*\*)&d\_count,sizeof(unsigned int));

cudaMemcpy(d\_A, A, strlen(A)\*sizeof(char), cudaMemcpyHostToDevice):

cudaMemcpy(d\_count,count,sizeof(unsigned int),cudaMemcpyHostToDevice);



CUDACount<<<1,str1en(A)>>>(d\_A,d\_count);

*ezzor=cuda* /*et£astEzzoz(), i (ezzor I—- c*zrd'*aSuccecs) {*

*pdnIfi(”*C'*UDA Ezzor2• fits\n”, c*zz*daGetEzzorSfifing(ezmr)),•*

cudaEventRecord(stop, 0); cudaEventSynchronlze(stop),

cudaEventElapsedTime(&elapsedTime, start, stop),

cudaMemcpy(result, d\_count, sizeof(unsigned int), cudaMemcpyDeviceToHost); printf{‘Tota1 occurences of a %u”,result);

printf("Time Tahm='7ofi’,elgpsedTime},

cudaFree(d\_A);

cudaFtee(d\_count);

getch(); return 0;

Explanation:

The kernel uses *itomicAJd* function with 1 as the value each time a character ‘a’ occurs in the string. The instructions given in bold are present to find the time. As in OpenCL you need to declare an event, register the event and record the time before kernel execution and after kernel execution. You have to synchronize the event so that main thread can capture the time of execution of kernel. After that cudaEventEIapsedTime(&elapsedTime,start, stop) will give the difference between the recorded stop and start time and store the value in the variable *elupseilTiitie* which is of type fioat. A negative time value means there is something wrong in the CUDA code. To find it out you use the code given in bold and italics. It will display the error message present in CUDA code. Call it once before calling the kernel and once after calling kernel. If first call throws an error message then error is present in CUDA API which precedes the kernel. If second call throws the error message then error is present in the kernel code.

Lab Exercises:

l. Write a program in CUDA to count the number of times a given word is repeated in a sentence. (Use Atomic function)

1. Wriie a CUDA program that reads a string *S* and produces the string *RS* as follows: Inpui string S: PCAP Output string US: PCAPPCAPCP

**Note: Each worh item** copies **required** number of characters **from** *S* in 2tS.

Additional Exercises:

I) Write a CUDA program which reads a string consisting of N words and reverse each word of it in parallel.

1. Write a CUDA program that takes a string Min as input and one integer value N and produces an output string , Seri, in parallel by concatenating input string Stat, N times as shown below.

Input: Sin = “Hello” N = 1 Ouput: Sout = “HelloHelloHello”

**Note:** Every thread copies the same **character from** the **Input string** S, N **times to** the re- quired **position.**

1. Write a CUDA program which reads a string *Sin* and produces an output string T as shown below.

Input: Sin: “Hai”

Ouput: *T:* “Haaiii “

Note:Every thread stores a character from input string Min, required number of times into output strap r.

Lab No 8:

Objectives:

## Programs on Matrix using CUDA

Date:

ln this lab, student will be able to

* 1. Understand how to write kernel code in CUDA to perform operations on matrix
  2. Learn about CUDA 2D blocks/threads
  3. Write simple program on two dimensional arrays

Solved Exercise:

Write a program in CUDA to find transpose of a matrix in parallel.



4include "cuda\_nintime.h"

#include "device\_launch\_parameters.h" #include <stdio.h>

#include <stdlib.h>

global void transpose(int \*a, int \*t)

int n=threadIds.x, m=blockIdx.x, size=blockDim.x, sizel=gridDim.x:

t[n\*sizel +m)m[m\*size+n]:

int main(void)

int \*a,\*t, m,n,ij;

int \*d\_a,\*d\_t;

printf("Enter the value of m: ");scanf(”9»d“,&m): printf("Enter the value of n: "):scanf(”9od",&n); int size=sizeof(int)\*m\*n; a=(int\*)ma1loc(m\*n\*sizeof(int)); c=(int\*)malloc(m\*n\*sizeof(int));

printf("Enter input matrix:\n"); for(i=0;i<m\*n;i++)

cudaMalloc((void\*\*)&d\_a,size);



cudaMalloc((void\*\*)&d\_t,size);

cudaMemcpy(d\_a,a,size,cudaMemcpyHostToDevice); transpose<<<i **,n>>>(d\_a,d\_t);** cudaMemcpy(t,d\_t,size,cudaMemcpyDeviceToHost); printf("Result vector is:\n");

for(i=0:i<n;i--r)

**for(j=0;j<m ++)**

**printf("9od\t”,t[i\*m+j)); printf("\n");**

getchar(); cudaFree(d\_a); cudaFree(d\_t);

Lab Exercises:

1. Write a program in CUDA to add two Matrices for the following specifications:
   1. Each row of resultant matrix to be computed by one thread.
   2. Each column of resultant matrix to be computed by one thre‹id.
   3. Each element of resultant matrix to be computed by one thread.
2. Write a program in CUDA to multiply two Matrices for the following specifications:
   1. Each row of resultant matrix to be computed by one thread.
   2. Each column of resultant matrix to be computed by one thread.
   3. Each element of resultant matrix to be computed by one thread.

**Additional** Exercises:

1. Write a CUDA program that reads it MXN matrix A and produces a resultant matrix B of same size as follows: Replace all the even numbered matrix elements with their row sum and odd numbered matrix elements with their column sum.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Example: |  | A |  |  |  | B |  |
|  | \/p: 1 | 2 | 3 | O/p: | 5 | 6 | 9 |
|  | 4 | 5 | 6 |  | 15 | 7 | 15 |

1. Write a CUDA program to read a matrix A of size NXN. It replaces the principal diagonal elements with zero. Elements above the principal diagonal by their factorial and elements below the principal diagonal by their sum of digits.

Lab No 9: **Date:**

### Programs on Matrix using CUDA(continued...)

**Objectives:**

In this lab, student will be able to

1. Understand how to write kernel code in CUDA to perform operations on matrix.
2. Learn how to represent sparse matrix using CSR format and perform parallel operations using it.

Compressed Sparse Row (CSR) format:

* + A sparse matrix is a matrix where the majority of the elements are zero. Sparse matrices are stored in a format that avoids storing zero elements.
  + The compressed spar.se row (CSR) format i.s a popular, general-purpose spar.se matrix representation. CSR explicitly stores column indices and nonzero values in arrays col\_index and data as shown below.
  + A third array of row pointers, row\_ptr. give the starting location of every row in the compressed

.storage.

A 0 B

q= C D E

###### D•ts{{l

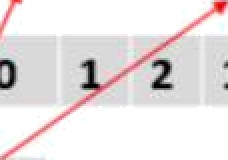
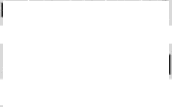






0 1 2 3 4 5

[A



B C D E Fj

2 a

g y]

o

=non zero elements

=coIumn indices of elements

(0 2 sj —pointers to the first element in each row

Lab Exercises:

l. Write a program in CUDA to perform parallel Sparse Matrix - Vector multiplication using com- pressed sparse row (CSR) storage format. Represent the input sparse matrix in CSR format in the host code.

1. Write a program in CUDA to read MXN matrix A and replace I' row of this matrix by same elements, 2“d row elements by square of each element and 3“ row elements by cube of each element and so on.
2. Write a CUDA program that reads a matrix A of size MXN and produce an output matrix B of same size such that ii replaces all the non-border elements (numbers in bold) of A with its equivalent l ’s complement and remaining elements same as matrix A.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | A |  |  |  |  | B |  | |
| 1 | 2 |  | 3 | 4 | 1 | 2 |  | 3 | 4 |
| 6 | 5 |  | 8 | 3 | 6 | 10 |  | 111 | 3 |
| 2 | 4 |  | 10 | 1 | 2 | 11 |  | 101 | I |
| 9 | 1 |  | 2 | 5 | 9 | 1 |  | 2 | 5 |

**Additional** Exercises:

* 1. Write a CUDA program which reads an input matrix A of size MXN and produces an output matrix B of size MXN such that, each element of the output matrix is calculated in parallel. Each element. B[i][j], in the output matrix is obtained by adding the elements in i" row and j"' column of the input matrix A.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Example: |  | A |  |  | B |  |
|  | 1 | 2 | 3 | O/p: 11 | 13 | 15 |
|  | 4 | 5 | b | 20 | 22 | 24 |

* 1. Write a CUDA program that reads a character type matrix A and integer type iriatrix B of size MXN. It produces an output string 5Ffi such that, every character ot *A* is repeated r times (where *r* is the integer value in matrix B which is having the same index as that o( the character taken in A). Write the kernel such that every value of input matrix must be produced required number of times by one thread.

Example: A

p C a P

e X a M

B

1 2 4 3

2 4 3 2

Output String STR: pCCaaaaPPPeeXXXXaaaMM

**Lab No 10: Date:**

# Programs using different CUDA Device memorv tvpes and

**synchronization**

Objectives:

ln this lab, student will be able to

1. Learn about CUDA 2D grid and 2D block
2. Implement the optimized parallel program applications using constant memory and shared memory

**Synchronization:** CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, **syncthreads().** When a kernel function calls **syncthreads(),** the thread that executes the function call will be held at the calling location until every thread in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase.

**Shared Variables:** Accessing shared memory is extremely fa.st and highly parallel. If a variable declaration is preceded by the keyword **shared** , it declares a shared variable in CUDA. Such declarations typically reside within a kernel function or a device function. The scope of a shared variable is within a thread block means all threads in a block see the same version of a shared variable. The lifetime of a shared variable is within the duration of the kernel. Shared variables are an efficient means for threads within a block to collaborate with each other. Shared memory is fast but it is small. A common strategy is partition the data into subsets called tiles so that each tile fits into the shared memory.

**Constant Variables:** If a variable declaration is preceded by the keyword **\_constant\_,** it de- clares a constant vai iable in CUDA. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execu- tion. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at G5,536 bytes. One may need to break up the input data volume to fit within this limitation.

**Device Variables: A** variable whose declaration is preceded only by the keyword  **device** is a global variable and will be placed in global memory. Accesses to a global variable are slow. However, global variables are visible to all threads of all kennels. Their contents also persist through the entire execution. Thus, global variables can be used as a means for threads to collab- orate across blocks. Global variables are often used to pass information from one kernel invocation to another kernel invocation.

Solved Exercise:

Write a program in CUDA to perform tiled matrix multiplication using 2D Grid and 2D Block



//Matrix multiplication of 4x4 matrix

#include <stdio.h> #include <std1ib.h> #include <unistd.h> #include <cuda\_runtime.h>

#define BLOCK\_WIDTH 2

#define TILE\_WIDTH 2

#define WIDTH 4

global void MatMu1ElementThreadShared(int \*a, int \*b, int \*c) { shared int MDs[TlLE\_WIDTH][TILE\_WIDTH];

\_shared\_ int NDs[TILE\_WlDTH][TH F WIDTH]; int m;

int bx=blockldx.x; iut by=blockldx.y; irit tx=threadIdx.x; int ty=thieadIdx.y;

int Row=by\*TII.F WIDTH + ty; ini Col= bx\*TILE\_WlDTH + n;

int Pvalue=0;

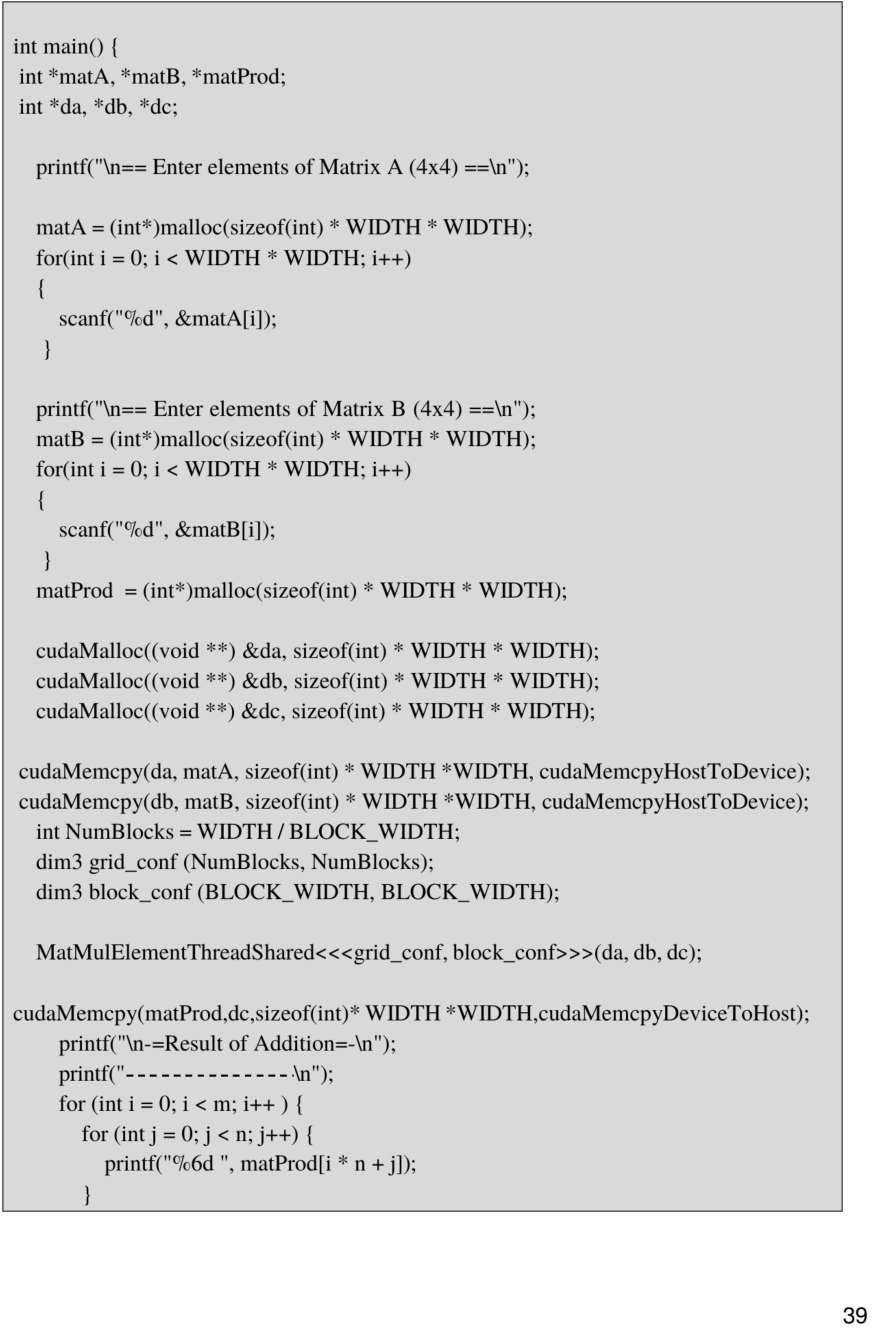
for(m=0; m<WIDTH/TILE\_WIDTH; m++)

MDs[ty][tx]=atRow\*WIDTH+m\*TiLE\_WIDTH+n],

NDs[ty)[tx]=b[(m\*TILE\_WIDTH+ty)\*WIDTH+Col];

for (int k = 0; k < TILE\_WIDTH; k++) Pvalue += MDs[ty][k]\*NDs[k][tx];

c[Row\*WIDTH-i-Col] = Pvalue;





Lab Eaerciaes:

1. Write a program in CUDA to perform matrix multiplication using 2D Grid and 2D Block.
2. Write a program in CUDA to improve the performance of ID parallel convolufion using constant

Memory.

1. Write a program in CUDA to perform inclusive scan algorithm.

Additional Enerdses:

J. Write a program in CUDA which displays a shopping mall item menu with its price. The N number of friends are allowed to purchase as many items they want. Calculate the total purchase done by N friends.

1. Write a program in CUDA to perform tiled ID convolution operation on the ñiput array N of size *width* using the mask array, M of size *mask\_widtJ* to produce the resultant array P of size *width.*

REFERENCES

l. Michael J. Quinn, *“Parallel Pro$ruinming iii C with MPI nut OperiMP”,* McGraw Hill

Edition, 2003.

2. D. Kirk and W. Hwu , *“Pru¿ramming Muse ivrl j’ Parallel Prof essoi s —A Hands-un up-*

*prou‹h”,* Elsevier Inc.,2" Edition, 2013.